Lightweight finite field operators for public key cryptography on resource-constrained devices

By:

Luis Armando Rodríguez Flores

Thesis submitted as the partial fulfillment
of the requirement for the degree of:

DOCTOR OF SCIENCE IN COMPUTER SCIENCE

at

Instituto Nacional de Astrofísica, Óptica y Electrónica

April, 2019
Tonantzintla, Puebla

Advisor(s):

Dr. René Armando Cumplido Parra*

*Coordinación de Ciencias Computacionales, INAOE

Dr. Miguel Morales Sandoval+

+ Cinvestav Unidad Tamaulipas

©INAOE 2019
All rights reserved
The author grants to INAOE the right to
reproduce and distribute copies of this dissertation

Acknowledgments

Firstly, I would like to express my sincere gratitude to my wife Jannet who always support me in the hard moments, and my child Isaac who always has helped me to smile the life. To my sisters: Guadalupe, Veronica and Angelica. To my parents: Guadalupe and José Luis for their love and support.

Beside my family, I would like to thank to my advisors René Cumplido (INAOE) and Miguel Morales Sandoval (Cinvestav Tamaulipas) for the continuous support in my PhD. studies, for their patience, motivation, immense knowledge, and above all for that great friendship that they offered me. Their guidance helped me during all the time of research and writing of this thesis. I could not have imagined to realize this research without their support.

To my graduate committee: PhD. Alicia Morales, PhD. Gustavo Rodríguez, PhD. Alejandro Medina and PhD. Miguel Arias for their encouragement, insightful comments, and inputs to improve this research.

My sincere thanks also goes to PhD. Claudia Feregrino-Uribe, PhD. Hayde Peregrina-Barreto, PhD. Lázaro Bustio-Martinez, PhD. Mariano Vargas-Santiago, MSc. José Valdés-Rayón, MSc. Jonathan Serrano-Cuevas, MSc. Miguel A. Alvarez-Carmona, MSc. Claudia Cruz-Martinez for their support, help and friendship.

Knowing that there will be no way to thank for a life of sacrifice, effort and love, I want you to feel that the goal reached is yours and that the strength that helped me to achieve it was their great support.

Finally, I thank CONACyT (Scholarship Grant 402861) and INAOE for their support during this PhD. studies. I thank the project thesis completion scholarship: Fondo Sectorial de Investigación para la Educación, CONACyT Proyecto de Ciencia Básica No. 281565, directed by Dr. Miguel Morales Sandoval.

Luis Armando Rodríguez Flores.

"They got the guns, but we have the numbers."

The Doors, Five to one.

Abstract

Nowadays, computational power is being used in various activities of human life. New computing paradigms such as the Internet of Things, wireless sensor networks, ubiquitous computing, and ambient intelligence make use of computing power to improve the quality of human life. Devices used in these new computing paradigms do not work in isolation, they communicate with each other or with the Internet to offer services. Also, it is expected that a large number of small devices are connected to the Internet in the near future.

Since a wide range of applications require that devices store, transmit and receive sensitive information, it is necessary to provide security services so that attackers do not compromise data and devices. Security services required by applications are confidentiality, authentication, integrity, and non-repudiation. These security services can be provided through cryptography.

Cryptography is divided in two areas: symmetric cryptography (or private key cryptography) and asymmetric cryptography (or public key cryptography, PKC). In private key cryptography, the sender and the receiver must agree to use specific information (key) to encrypt and decrypt messages. Public key cryptography proposes to use a pair of keys for each user, one public and the other one private. The public key can be known by anyone and can be used to encrypt a message for the owner of the key pair, which can decrypt the message with his private key, which presumably only he knows. Public key cryptography provides the four aforementioned security services, while private key cryptography only provides the confidentiality service. The main disadvantage with public key cryptography is that it requires a more considerable amount of computational power than private key cryptography, this is because it bases its security on mathematical problems defined in groups and finite fields. The core and most time consuming operations in any PKC system are the ones related to group and finite field operations (multiplication, inversion and exponentiation). Therefore, both types of cryptography are commonly used together, for example, public key

cryptography can be used to perform a key exchange that is subsequently used in a private key cryptography system.

Unlike traditional devices such as servers, personal computers, laptops, smartphones, etc., devices used in ubiquitous computing paradigms commonly are resource-constrained devices in terms of memory, computational power, and power consumption. Consequently, the processing time (in the order of seconds) of software implementations of public key cryptography in these devices is unacceptable for some applications. Therefore, specialized hardware is required to accelerate the most demanding (computationally) operations of public key cryptography. Works proposed in the literature accelerate public key cryptography operations at the expense of a large amount of hardware resources often not available in devices with limited resources commonly used in wireless sensor networks or the Internet of Things.

Being group and finite field arithmetic the bottleneck in PKC, this thesis research proposes new hardware finite field operators implementations for PKC realizations on resource-constrained devices. As a key distinctive approach, hardware operators are realizations on novel algorithms that performs the group and field operation in a digit-by-digit fashion. In this approach, the operands and the partial results are divided into digits and processed one digit at a time, similar to a software implementation, but making use of the parallelism and the resources available in hardware. The proposed operators have been developed for reconfigurable hardware Field-Programmable Gate Array (FPGA).

The contributions of this thesis are: *i)* Novel algorithm for Montgomery multiplication, *ii)* Novel algorithm for binary field $\mathbb{F}_{2^m}$ multiplication, *iii)* Compact hardware architectures for multiplication in the prime field $\mathbb{F}_p$ and binary field $\mathbb{F}_{2^m}$, *iv* compact hardware exponentiation in the prime field $\mathbb{F}_p$ and in the additive group of elliptic curves $\mathbb{E}(\mathbb{F}_{2^m})$. A hardware-software co-design of two of the most representative schemes in PKC, key exchange and digital signature, was done to evaluate the proposed operators which require fewer hardware resources than state of the art works reported in the literature. The obtained results establish an area-performance trade-off that helps to choose the most appropriate hardware architecture to accelerate operations required in public key cryptography in resource-constrained devices.

Resumen

Actualmente el poder computacional esta siendo utilizado en diversas actividades de la vida humana. Nuevos paradigmas de computación como el Internet de las cosas, redes de sensores inalámbricos, computo ubicuo, y ambientes inteligentes hacen uso del poder computacional para mejorar la calidad de la vida de la humanidad. Los dispositivos usados en estos nuevos paradigmas de computación no trabajan aisladamente, se comunican entre ellos o con Internet para poder ofrecer sus servicios. Además, se espera que una gran cantidad de dispositivos estén conectados a Internet en el futuro próximo.

Ya que en gran cantidad de aplicaciones se requiere que los dispositivos almacenen, transmitan y reciban información sensible es necesario proveer servicios de seguridad para que los atacantes no comprometan los datos o los dispositivos. Los servicios de seguridad que requieren las aplicaciones informáticas son: confidencialidad, autenticación, integridad y no-repudio. Estos servicios de seguridad pueden proveerse a través de la criptografía.

La criptografía se divide en dos áreas: criptografía simétrica (o de llave privada) y criptografía asimétrica (o de llave pública). En la criptografía de llave privada el emisor y el receptor deben ponerse de acuerdo en usar cierta información (llave) para cifrar y descifrar mensajes. La criptografía de lleve pública propone usar un par de llaves para cada usuario, una pública y otra privada. La llave pública la puede conocer cualquiera y la pueden usar para cifrar un mensaje para el dueño del par de llaves, el cual puede descifrar el mensaje con su llave privada, que presumiblemente solo él conoce. La criptografía de llave pública provee los cuatro servicios de seguridad antes mencionados, mientras que la criptografía de llave privada solo provee el servicio de confidencialidad. La principal desventaja con la criptografía de llave pública es que requiere mayor cantidad de poder computacional que la criptografía de llave privada, esto se debe a que basa su seguridad en problemas matemáticos definidos en grupos y campos finitos. Las operaciones base y más demandantes en tiempo

en cualquier sistema de criptografía de llave pública son operaciones aritméticas en grupos y campos finitos (multiplicación, inversión y exponenciación). Por lo cual, comúnmente se utilizan los dos tipos de criptografía en conjunto, por ejemplo, se puede usar la criptografía de llave pública para realizar un intercambio de llave que posteriormente es usada en un sistema de criptografía de llave privada.

A diferencia de los dispositivos tradicionales como servidores, computadoras personales, computadoras portátiles, teléfonos inteligentes, etc., los dispositivos utilizados en los paradigmas de computación ubicua comúnmente tienen recursos limitados en cuanto a memoria, poder computacional y consumo de energía. Esto ocasiona que el tiempo de procesamiento (en el orden de segundos) de implementaciones en software de la criptografía de llave pública sea inaceptable para algunas aplicaciones. Por lo cual se requiere de hardware especializado para acelerar las operaciones computacionalmente más demandantes de la criptografía de llave pública. Trabajos propuestos en la literatura aceleran las operaciones de criptografía de llave pública a costa de gran cantidad de recursos hardware muchas veces no disponibles en dispositivos con recursos limitados comúnmente usados en redes de sensores inalámbricos o Internet de las cosas.

Siendo la aritmética en grupos y campos finitos el cuello de botella de la criptografía de llave pública, este trabajo de investigación propone nuevos operadores hardware en campos finitos para la realización de criptografía de llave pública en dispositivos con recursos limitados. Como un enfoque distintivo clave, los operadores hardware son implementaciones de algoritmos que ejecutan operaciones en grupos y campos con un enfoque dígito a dígito. En este enfoque los operandos y los resultados parciales son divididos en dígitos y procesados un dígito a la vez, similar a una implementación en software, pero haciendo uso del paralelismo y los recursos disponibles en hardware. Los operadores propuestos han sido desarrollados para hardware reconfigurable Field-Programmable Gate Array (FPGA).

Las aportaciones de esta tesis son: *i)* Algoritmo novedoso para la multiplicación Montgomery, *ii)* Algoritmo novedoso para la multiplicación en el campo binario $\mathbb{F}_{2^m}$, *iii)* Arquitecturas hardware compactas para la multiplicación en el campo primo $\mathbb{F}_p$ y el campo binario $\mathbb{F}_{2^m}$, *iv* Implementación hardware compacta para la exponenciación en el campo primo $\mathbb{F}_p$ y en el grupo aditivo de curvas elípticas $\mathbb{E}(\mathbb{F}_{2^m})$. Un co-diseño hardware-software de los esquemas de criptografía de llave pública más representativos, intercambio de llaves y firma digital, fueron implementados para evaluar los operadores propuestos, los cuales requieren menor cantidad de recursos hardware que los trabajos reportados en el estado del arte. Los resultados obtenidos establecen

un compromiso área-desempeño que ayudan a elegir la arquitectura hardware más adecuada para acelerar las operaciones que se requieren en la criptografía de llave pública en dispositivos con recursos limitados.

Contents

1 Introduction 1

1.1 Pervasive computing, constrained devices and security threats . . . 1

1.2 Data security with public key cryptography . . . 4

1.3 Challenges of public key cryptography in constrained environments . . 7

1.4 Finite field operators in public key cryptography . . . 8

1.5 Research problem . . . 10

1.6 Hypothesis . . . 11

1.7 Research objectives . . . 11

1.7.1 General objective . . . 11

1.7.2 Specific objectives . . . 11

1.8 Methodology . . . 12

1.9 Thesis outline . . . 14

2 Preliminaries 15

2.1 Public key cryptography (PKC) . . . 15

2.1.1 Groups and cyclic groups . . . 18

2.1.2 Finite fields (F_q) as groups construction . . . 20

2.1.3 The prime field $\mathbb{F}_p$ and the multiplicative group $\mathbb{F}_p^*$ . . . 21

2.1.4 Discrete Logarithm Problem in multiplicative groups (DLP-$\mathbb{F}_p^*$) . 23

2.1.5 DLP-F_p* based cryptosystems (DH, RSA, RSA-Digital Signature) 23

2.1.6 The role of finite field operators in DLP-$\mathbb{F}_p^*$ based cryptosystems 26

2.2 Curve based Cryptography: a case of PKC . . . 27

2.2.1 Additive group of elliptic curve points . . . 29

2.2.2 Elliptic curves over a finite field ($E(F_q)$) . . . 30

2.2.3 Elliptic curves over prime field ($E(F_p)$) . . . 30

2.2.4 Elliptic curves over binary field ($E(F_{2^m})$) . . . 31

2.2.5 Discrete Logarithm Problem in additive elliptic curve groups (DLP-$E(F_q)$) . . . 32
2.2.6 DLP-$E(F_q)$ based cryptosystems (Elliptic curve cryptography - ECC) . . . 33
2.2.7 Elliptic curves ElGamal public key cryptosystem . . . 33
2.2.8 ECDH key exchange . . . 33
2.2.9 Finite field operators role in DLP-$E(F_q)$ based cryptosystems . . . 35
2.3 Finite field operators as key elements in PKC realizations for constrained environments . . . 36
2.3.1 Multiplication, inversion and squaring operators in F_q . . . 37
2.3.2 Exponentiation operator in F_q^* . . . 39
2.3.3 Exponentiation (scalar multiplication) operator in $E(F_q)$ . . . 40
2.4 Summary . . . 41

3 State of the art 43
3.1 Finite field operators in hardware . . . 43
3.2 Hardware architectures for the multiplication operator over $\mathbb{F}_p$ . . . 44
3.3 Hardware architectures for exponentiation operator over $\mathbb{F}_p^*$ . . . 48
3.4 Hardware architectures for the multiplication operator over $\mathbb{F}_{2^m}$ . . . 50
3.5 Hardware architectures for the scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$ . . . 52
3.6 Summary . . . 55

4 Proposed lightweight finite field operators 57
4.1 Digit-digit computation approach . . . 57
4.2 Compact digit-digit Montgomery multiplication in $\mathbb{F}_p$ . . . 60
4.2.1 A variant of the digit-digit Montgomery implementation . . . 69
4.3 Compact exponentiation in $\mathbb{F}_p^*$ . . . 72
4.4 Operators over $\mathbb{F}_{2^m}$ . . . 76
4.5 Compact digit-digit $\mathbb{F}_{2^m}$ multiplication . . . 79
4.5.1 Hardware architecture 1 (Compact implementation) . . . 80
4.5.2 Hardware architecture 2 (Karatsuba version) . . . 81
4.5.3 Hardware architecture 3 (One partial multiplier version) . . . 82
4.6 Compact exponentiation in $\mathbb{E}(\mathbb{F}_{2^m})$ . . . 83
4.7 Summary . . . 90

5 Implementation and Results 91
5.1 Tools and evaluation metrics . . . 91

5.2 Area performance trade-off approach 93
5.3 Compact FPGA operators in $\mathbb{F}_p$ 93
5.4 Compact FPGA exponentiation in $\mathbb{F}_p^*$ 96
5.5 Compact FPGA operators in $\mathbb{F}_{2^m}$ 98
5.6 Compact FPGA $\mathbb{E}(\mathbb{F}_{2^m})$ exponentiator 101
5.7 Comparisons 102
5.7.1 $\mathbb{F}_p$ multiplication 104
5.7.2 $\mathbb{F}_p$ exponentiation 105
5.7.3 $\mathbb{F}_{2^m}$ multiplication 108
5.7.4 $\mathbb{E}(\mathbb{F}_{2^m})$ scalar multiplication 110
5.8 In-circuit verification of FPGA finite field operators 112
5.8.1 HW/SW co-design for operators over $\mathbb{F}_p$ 112
5.8.2 HW/SW co-design for operators over $\mathbb{E}(\mathbb{F}_{2^m})$ 114
5.9 Summary 116

6 Conclusions and directions **119**
6.1 Summary of contributions 120
6.2 Publications 121
6.3 Future work 121

A Test vectors **123**
A.1 RSA digital signature scheme 123
A.2 Elliptic Curve Diffie-Hellam key exchange 123

B List of Acronyms **127**

List of Figures

1.1 User-computational device relation. a) In the past, scarce computers were used by many people, b) In the 1980s it was possible that each user poses its own computer, c) Nowadays, several low-cost computers are available for a single user in the form of personal and familiar objects. 2
1.2 Pervasive computing devices in cyber-physical systems. 3
1.3 Communication in a Wireless Sensor Network. 5
1.4 Intruder node in a Wireless Sensor Network. 6
1.5 Intruder node breaks the communication between two nodes in a Wireless Sensor Network. 6
1.6 A Hierarchical Layer Model for Information Security Applications. . . . 9
1.7 FPGA design flow. 13

2.1 Public key cryptography. 16
2.2 Digital signature. 16
2.3 One-way trapdoor function. 18
2.4 Group properties. 18
2.5 A finite cyclic group generated by a. 20
2.6 RSA digital signature scheme. D = Document, s = *signing exponent*, S = signature, v = *verification exponent*, V = Verification 27
2.7 Graphical view of point double/addition in elliptic curves. 29
2.8 Elliptic curve over real numbers and over the prime field. 31
2.9 Elliptic Curve Diffie-Hellman key exchange protocol. 34
2.10 Square of the element $A \in \mathbb{F}_{2^m}$. 38

3.1 Lightweight hardware architecture for scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$ presented in [1]. 53
3.2 Hardware-software co-design for scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$ proposed in [2]. 54

4.1 Graphical view of a full parallel multiplier. 58
4.2 Graphical view of a digit-serial multiplier. One digit of B is processed at a time. In this step the digit B_1 is processed. 59
4.3 Graphical view of a bit-serial multiplier. One bit of B is processed at a time. In this step the bit b_1 is processed. 59
4.4 $A^{<i+1>}$ computation in a digit by digit approach. 65
4.5 Dataflow in variable A acording to algorithm 4 67
4.6 New digit-digit Montgomery multiplier architecture (Algorithm 4), memory and result reside in memory blocks. 68
4.7 $A^{<i+1>}$ computation using different digit sizes for X and Y. 70
4.8 Montgomery multiplier digit-digit different digit size approach. 72
4.9 Digit-digit Montgomery Powering Ladder architecture (Algorithm 7). . 75
4.10 Computation of P_j and R_j. 80
4.11 Partial computation of $S_j^{<i>}$. 81
4.12 Hardware architecture 1. 82
4.13 Hardware architecture 3. 84
4.14 Memory blocks for partial result operations. 86
4.15 Schedule for the point addition and point double operations in the context of elliptic curves. 87
4.16 Graphical view of the fast reduction Algorithm 13. 88
4.17 General view of the ECC scalar multiplier architecture (Algorithm 9). . 89

5.1 Implementation results of the Montgomery multiplier (Figure 4.6) in the Virtex-7 FPGA. 94
5.2 Implementation results of the $\mathbb{F}_p$ multiplier with different digit sizes (figure 4.8) in the Virtex-7 FPGA for 1024 bits operand size. 97
5.3 Implementation results for the $\mathbb{F}_p^*$ exponentiator (MPL) architecture for a Virtex-7 Xilinx FPGA. 98
5.4 Implementation results for the $\mathbb{F}_{2^m}$ multiplier architecture in the Virtex-7 Xilinx FPGA. 100
5.5 Implementation results for the $\mathbb{E}(\mathbb{F}_{2^m})$ scalar multiplier in the Zynq-7000 Xilinx FPGA. 103
5.6 Hardware-software co-design for exponentiation over $\mathbb{F}_p$. 113
5.7 Hardware-software co-design for $\mathbb{E}(\mathbb{F}_{2^m})$ scalar multiplier. 115

List of Tables

2.1 Multiplication modulo 5. 21
2.2 Addition modulo 5. 21
2.3 RSA public key cryptosystem. 25
2.4 FPGA sensor key creation 26
2.5 Elliptic curves cryptography, ElGamal encryption and decryption 33
2.6 Required finite field operations in point addition and point double in elliptic curves cryptography. 35
2.7 Finite field operations required in a point multiplication in elliptic curve cryptography over $\mathbb{F}_p$ with 256-bits in size. 35

3.1 Approaches taken in hardware finite field operator $\mathbb{F}_p$ multiplier. 47
3.2 Hardware approaches for finite field exponentiation over $\mathbb{F}_p$. 49
3.3 Hardware approaches for multiplication over $\mathbb{F}_{2^m}$. 51
3.4 Hardware approaches for exponentiation over $\mathbb{E}(\mathbb{F}_{2^m})$. 55

4.1 Notation. 62
4.2 Microcode for the control module. 69
4.3 NIST recommended irreducible polynomials for binary fields. 83

5.1 Implementation results for the datapath block (1024 bits operand size). 95
5.2 Implementation results for the control block, using the microprogramming and FSM approaches (1024 bits operand size). 95
5.3 Implementaciones compactas del algoritmo Montgomery. 104
5.4 Results and comparison for a 1024-bit exponentiation. 106
5.5 Supply power (W) of the MPL architecture. 107
5.6 $\mathbb{F}_p$ exponentiation in software vs. proposed $\mathbb{F}_p$ exponentiation compact hardware architecture. 108
5.7 State of the art works for finite field hardware operators over $\mathbb{F}_{2^m}$. . . . 108

5.8 State of the art works for finite field hardware operator over $\mathbb{E}(\mathbb{F}_{2^m})$. . . 110
5.9 Implementation results for MPL exponentiation over $\mathbb{F}_p$ in the MicroZed. 114
5.10 Implementation results for the RSA digital signature hw-sw co-design in the MicroZed. 114
5.11 Scalar multiplier over $\mathbb{E}(\mathbb{GF}(2^m))$ implementation results in the MicroZed board. 115
5.12 ECDH hw-sw co-design implementation results. 116

A.1 Selected parameters for in-circuit verification of the RSA digital signature scheme. 124
A.2 Test vectors for the RSA digital signature 125
A.3 Test vectors for the RSA digital signature verification 125
A.4 Test vector for ECDH in the hw-sw co-design (hexadecimal base) 126

List of Algorithms

1 Montgomery multiplication algorithm (**MMA**) 61
2 Iterative digit-digit MMA algorithm presented in [3] 64
3 Iterative Montgomery Multiplication [4] 65
4 New iterative Montgomery Multiplication algorithm 66
5 Montgomery digit-digit multiplication with different digit size. 71
6 MPL method for exponentiation in $\mathbb{GF}(p)$ 73
7 Digit-digit MPL algorithm 73
8 Digit-digit $\mathbb{F}_{2^m}$ multiplier algorithm. 81
9 Montgomery Scalar Multiplication 85
10 Add Algorithm 85
11 Doubling Algorithm 85
12 Coordinates conversion projective-affine. 86
13 Fast reduction modulo $f(z) = z^{233} + z^{74} + 1$ (With $W = 32$) 87

Chapter 1

Introduction

This chapter presents an introduction to the field of study. It describes the new computing paradigms such as the Internet of Things (IoT), Ambient Intelligence (AmI), and ubiquitous computing as well as their security threats. Furthermore, it presents the context in which the research problem arises, the specific and general aims of this research and a summary of the main contributions achieved in this dissertation. Finally, it is presented the organization of this dissertation at the end of the chapter.

1.1 Pervasive computing, constrained devices and security threats

At the beginning of the computer era, computers were enormous and very expensive. At that time, computers were scarce and owned only by big enterprises or educational institutions. Computer resources had to be shared among many users, motivating the creation of distribute computing mechanisms [5].

In the next step in computers evolution, a group of people could connect to the same computer to work simultaneously. Computers became popular, there were more computers than before, and some of them were interconnected, often in a wired way.

However, in the 1980s the Personal Computer (PC) revolution changed the world [5]. The development of the microprocessor caused that personal computers were cheap enough thus accessible to ordinary people.

Today, due to the progress in technology for high integration of circuits and systems, people have a variety of computational resources at their disposal unlike some few decades ago. Not only PCs as initially thought ideated are available for users. Now, people often have some mobile devices like cell phones, tablets, etc., see Figure 1.1. Furthermore, computational resources often are embedded in familiar objects such as cars, TV's, refrigerators, washing machines, etc. Also, these new computational

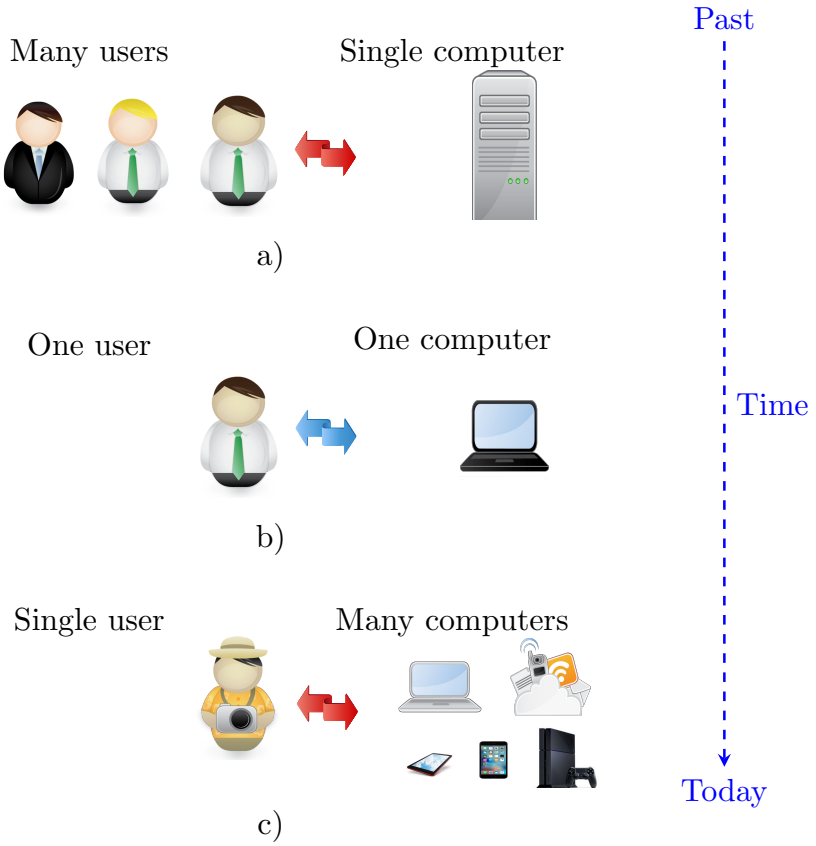

Figure 1.1: User-computational device relation. a) In the past, scarce computers were used by many people, b) In the 1980s it was possible that each user poses its own computer, c) Nowadays, several low-cost computers are available for a single user in the form of personal and familiar objects.

resources often are interconnected wirelessly.

As it was envisioned by Weiser [6], the computational power is spread today in all aspects of the human life with new paradigms of Pervasive/Ubiquitous computing [6, 7], Internet of Things (IoT) [8] and Ambient Intelligence (AmI) [9]. Pervasive Computing also referred sometimes as Ubiquitous Computing, deals with the idea of making computing power available anyplace, anytime in a uniform way so that it may be exploited for meeting challenges faced by society [10]. In the same way, Ambient Intelligence has been defined as a potential future in which people will be surrounded by intelligent objects and in which the environment will recognize the presence of persons and will respond to it in an undetectable manner [11].

Pervasive applications are in many fields such as military, scientist, medicine, econ-

omy, agriculture, etc. New computing paradigms offer a lot of benefits by customizing the environment to meet people needs. Some pervasive computing applications are Smart home, Health monitoring, Health assistance, Smart transportation, Smart hospitals, etc [9]. For example, in smart livings, sensors as small computing devices are distributed in the environment to collect data and create a context-aware application.

Computing platforms, i.e., pervasive computing devices, have generally limited computing capabilities if compared with traditional desktop or server machines. Some typical computing devices in pervasive applications include Radio-frequency identification (RFID) tags, wireless sensors, and smart cards, see Figure 1.2.

Almost all devices used in pervasive applications are resource constrained devices concerning memory, computational power, and energy consumption. Additionally to the computational load that these devices face due to end user application, computation and communication overhead is added if a security layer is required.

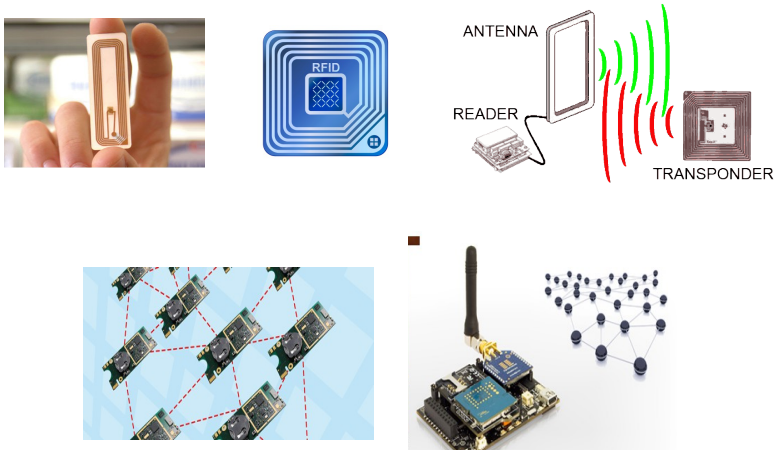

Figure 1.2: Pervasive computing devices in cyber-physical systems.

However, a critical point for development and practical realization of previously mentioned applications, is concerned with data and communication links security. In the new computing paradigms, battery-driven devices are commonly deployed in unfriendly or risk areas. As a result of this, devices become easy-targets of attacks compared to home and enterprise computers systems [12]. Because devices are physically accessible, they are vulnerable to many attacks. Cyber-physical systems create new classes of risks resulting from their interaction between cyberspace and the physical world. For example, Linux.Darlloz worm was discovered in 2013 by Symantec researchers. The worm was able to propagate to IoT devices such as home routers, TV set-top boxes, security cameras, printers, and industrial control systems.

In January 2014, a variant of the worm was found to include a cryptocurrency mining tool. Another real-life example is the IoT botnet built from Mirai malware that in September 2016 was responsive for a 600-Gbps attack targeting Brian Krebs's security blog [13].

Some of the most common security attacks on ubiquitous computing environments are [14]: man-in-the-middle attack, illegal connection attack, capturing sensitive data, stealing an intermediary device, data manipulation, impersonating and insiders. Despite a large number of existing attacks, often, security-related components are absent in end-user pervasive applications, or are added as an afterthought as an extra *feature* [15].

1.2 Data security with public key cryptography

Typically, an end-user pervasive application uses several small computing devices or nodes that establish communication with each other, and interactions between them and other systems. Since nodes could store and transmit sensitive information, it is necessary to provide them with security services of confidentiality, authentication, integrity, and non-repudiation (security services are defined in Section 2).

As an example, consider the graphical view (Figure 1.3) of a set of networked mobile computing devices in pervasive computing (i.e., a military Wireless Sensor Network (WSN) or a Body Area Network (BAN)). A security layer to protect the sensitive information transmitted under this scenario should guarantee that:

1. Nodes are correctly authenticated each other.
2. Integrity of transmitted data between two nodes is guaranteed. That is, it will be perceived if data is maliciously modified.
3. All transmitted data between two nodes is confidential.

From an origination point, information may pass through several nodes before arriving at the final destination. For example, communicating data from A to E in Figure 1.3 requires that information goes through path A-C-D-E. Regardless of the information pass through nodes C and D, data must be only revealed in an understandable form to E (**confidentiality**), and E can verify that the received information comes from A (**authenticity**) without suffering any modification during transmission (**integrity**). Once E receives the information, A cannot deny it was the node which initiated the communication (**non-repudiation**).

The previous security services allow thwarting some common attacks to the model shown in Figure 4. Some of them include:

1. Interception, when an unauthorized node listens and access in plain form the information that passes between A and E, see Figure 1.4.
2. Interruption, when an intruder node breaks the communication between two nodes, see Figure 1.5.
3. False node, involves the addition of a node by an adversary and causes the injection of malicious data [16], see Figure 1.5.
4. Modification, when a malicious node catches a message and change it. It adds wrong data (about receiver, sender or information itself) or deletes some packets. The message becomes corrupted [17].
5. Fabrication, when a malfunctioning node will generate inaccurate data that could expose sensor network integrity [16].

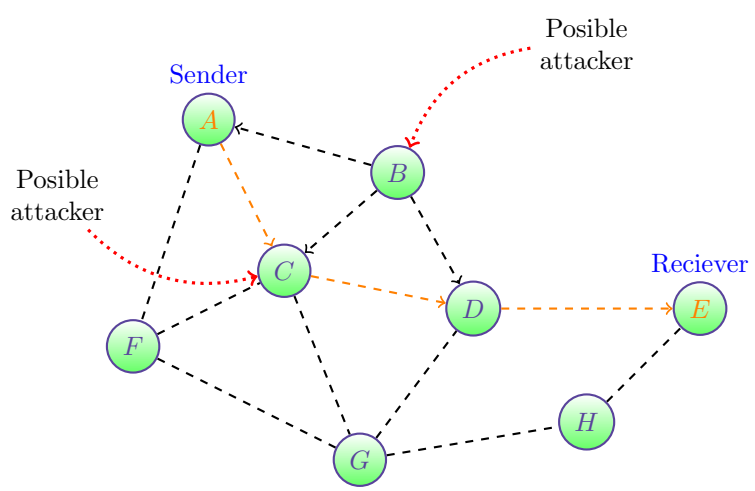

Figure 1.3: Communication in a Wireless Sensor Network.

The guarantee of security services has been extensively studied, and many secure and effective mechanisms are used today to protect computing systems and their assets. Most of these known security solutions are based on cryptography, either symmetric (private key cryptography) or asymmetric (public key cryptography). Symmetric Key Cryptography (SKC) only provides confidentiality service, while Public Key Cryptography (PKC) provides confidentiality, authentication, integrity, and non-repudiation

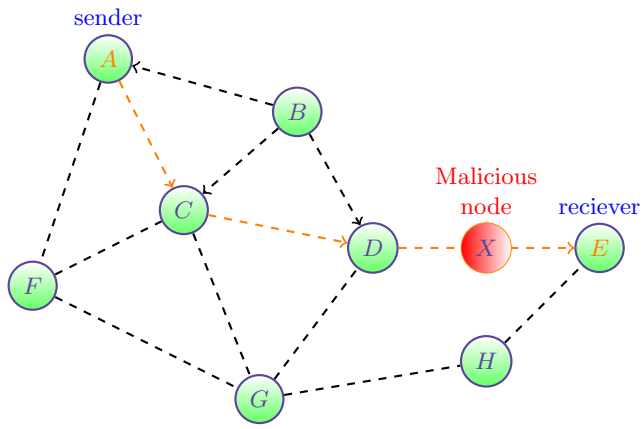

Figure 1.4: Intruder node in a Wireless Sensor Network.

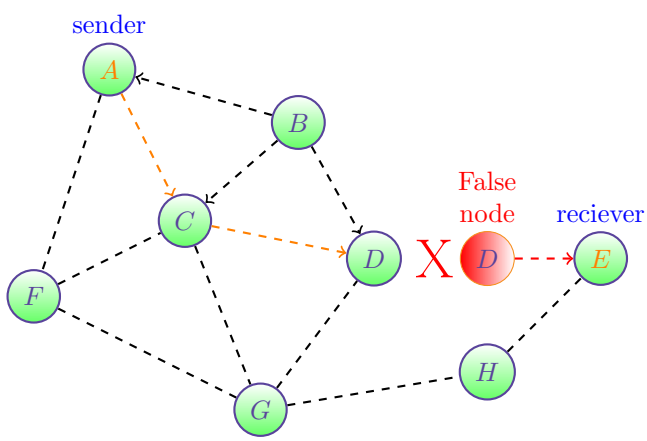

Figure 1.5: Intruder node breaks the communication between two nodes in a Wireless Sensor Network.

services. SKC and PKC have been widely implemented in servers, routers, personal computers, etc. However, in the next section, challenges to implement a PKC algorithm in resource-constrained devices and in envisioned IoT applications are drawn.

1.3 Challenges of public key cryptography in constrained environments

It is known that public key cryptography algorithms are 100-1000 slowly than private key algorithms [1, 18]. In addition, PKC algorithms require most resources than SKC algorithms. In practice, both of them are used in conjunction: a public key algorithm is used to securely establish a private key between two parties, and then that key is used by a symmetric cipher to encrypt and protect all data exchange between two parties.

PKC is slower than SKC since PKC is based on mathematical problems defined over algebraic structures where elements are large (i.e, of hundreds or thousands bits). Under these structures, i.e., finite fields, an encryption operation demands thousands of arithmetic operations, such as inversion, multiplication, exponentiation, squaring and addition. That is why, most of the known solutions for information security based on cryptography fail to be used in pervasive computing applications, the main reasons are:

1. PKC operations are very costly. A typical software implementation of public key cryptosystems (e.g., RSA) with a key length of 1024-bits in mobile devices demands more than 22 seconds and consumes above 726 mWs [19]. Furthermore, it is known that the most critical operation in the Rivest–Shamir–Adleman (RSA) cryptosystem, modular exponentiation, is the most time-consuming operation, requiring more than 20 seconds of the total computation in low-resource devices [20].

2. PKC requires a considerable amount of memory space for storing parameters and temporary results in the computation. For example, the MICAz mote only has 4KB RAM, which is the total space for data and program stack. Since operands in 1024-bit RSA are mostly 128 8-bit integers, subroutines, to support the underlying arithmetic (i.e., field multiplication, modular reduction, etc.) have to reserve a considerable amount of memory space for storing temporary results [20].

3. A typical hardware implementation of public key cryptography algorithms is focused on high-speed computation and often is not aware of energy and memory

requirements. However, using these solutions in pervasive application devices is not possible, due to the high area resources and power consumption.

As it has been pointed out, PKC algorithms can provide security services demanded in IoT applications, but they must be carefully implemented in that domain due to the scarce computational resources. During that implementation phase, the main challenge is to efficiently design the underlying arithmetic modules. The next section give a brief introduction to the arithmetic required in PKC, and particularly, it introduces the concept of **finite field operators**.

1.4 Finite field operators in public key cryptography

A PKC based solution for providing security services can be viewed in several layers, such as it is shown in Figure 1.6. The security protocols used to guarantee confidentiality, integrity and authentication in pervasive computing applications rely on PKC schemes (encryption/decryption, digital signatures). These schemes are implemented as a series of finite field operations. In this thesis, a finite field operator is referred as a module (either in hardware or software) to execute one finite field operation. Finite Fields are algebraic structures built over the group concept. A group is a nonempty set of elements G together with a binary operation over G that satisfies the properties of: Closure, Associativity, Identity and Inverse element [21]. Security of most known PKC relies on the group structure. Finite Fields have a finite number of elements $n = p^m$. This number of elements is a prime power p^m. When $m = 1$, finite fields are namely the prime field, and is denoted as $\mathbb{F}_p$. Another interesting finite field for cryptographic applications is the binary field commonly represented as $\mathbb{F}_{2^m}$. An extended description for group and finite field es presented in Chapter 2.

According to Figure 1.6, finite field arithmetic is an essential aspect for public key cryptography deployment. For example, in the RSA cryptosystem, the most critical and costly operation is modular exponentiation over prime field $\mathbb{F}_p$. Usually, numbers in the RSA cryptosystem are 1024-4096 bits in size. Another example is elliptic curve cryptography where arithmetic operations are performed over elements of 163-571 bits in size, according to the National Institute of Standards and Technology (NIST) government use recommendations [22]. Modular exponentiation with big numbers is computationally expensive, hence the motivation to speed up its computation with customized hardware. A modular exponentiation required in a RSA implementation with an exponent with 1024-bits in size may require up 2048 modular multiplications with algorithms such as binary exponentiation.

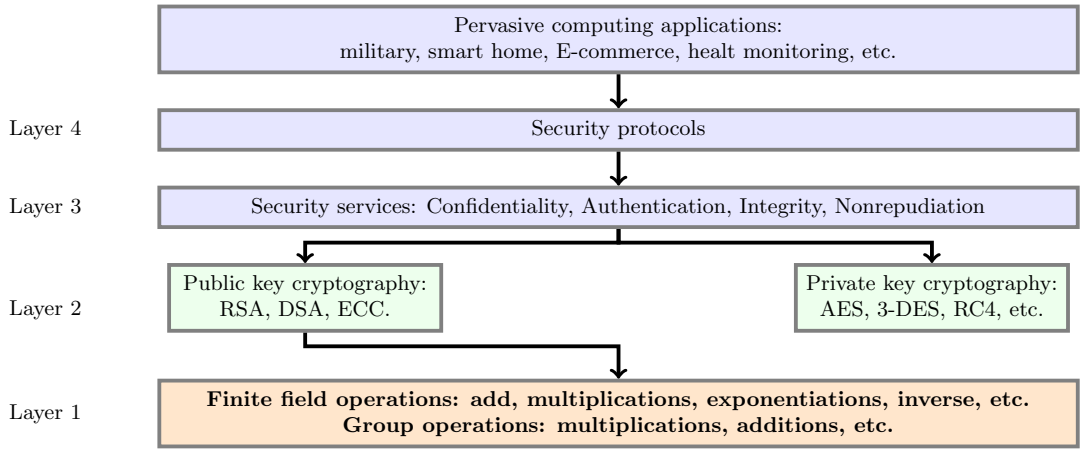

Figure 1.6: A Hierarchical Layer Model for Information Security Applications.

In the context of elliptic curve cryptography, the most used operation is point addition and point multiplication. However, since elliptic curves used in cryptography are defined over finite fields $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$, required operations are additions/subtraction, square, inversion, and multiplication in the corresponding field.

Finite field operations in RSA and Elliptic Curve Cryptography (ECC) are high and are the most time-consuming operations in those PKC cryptosystems. In consequence, in the literature, several works have proposed different approaches to construct finite field hardware accelerators but mainly motivated to achieve faster computations at the cost of higher hardware resources. Even at present, motivation remains on finding the most appropriate architecture for this operation depending on the context of application and underlying computing platforms.

In the context of pervasive computing, hardware implementations are restricted to use few hardware resources. So, it is necessary to keep hardware resources as low as possible and achieve a better performance than software implementations.

In the literature, the main aim of known hardware solutions is to provide high levels of security, without considering requirements of resource-constrained devices. A new field called lightweight cryptography (LWC) is an emerging research field, and focuses on designing schemes for devices with constrained capabilities [23]. So, the security in pervasive computing paradigms is where the research problem of this thesis work arise.

1.5 Research problem

In recent times, it is evident that new paradigms of Pervasive/Ubiquitous computing, Internet of Things (IoT), Ambient Intelligence (AmI), Wearable computing, etc., are a reality. These paradigms spread the computational power in the form of little-interconnected computing devices to all aspect of human life.

However, the lack of security in these domains is a real threat when these devices acquire, store, process and communicate sensitive information. High integration of pervasive computing applications in people's life, as for example in health applications, demands effective and practical security mechanisms to ensure confidentiality, authentication, integrity, and non-repudiation. Particularly, these security services can be provided by Public Key Cryptography (PKC) (e.g., RSA and ECC cryptosystems). However, since PKC algorithms rely on number theory, efficient operations in the underlying algebraic structures of PKC are demanded. These operations are considered computationally expensive since they are computed over large integers (163-4096 bits). For practical realizations of PKC-based protocols and security schemes, specialized hardware for finite field operators is desired, but in the context of pervasive computing applications, to develop such hardware architectures is challenging, the main reasons are:

- PKC algorithms based on abstract algebraic structures defined over large a set of numbers, with order around 2^{163} to 2^{4096} or either greater.
- A PKC scheme demands thousands of arithmetic operations such as inversion, multiplication, exponentiation, squaring and addition in algebraic structures, as groups and finite fields.
- PKC-based security solution requires more energy, memory or computing power than the one available in pervasive applications.
- Timing requirements of security solutions are usually achieved through hardware support. However, current hardware architectures for PKC have been designed to achieve high speed without considering the limited computing capabilities of the target devices.

The main approaches for hardware implementations of PKC have focused on speeding up the underlying finite field operations at the expense of high amounts of hardware resources. The implementation of PKC-based security solutions in resource-constrained devices using a straightforward approach is not viable.

The problem addressed in this research work is the design and implementation of efficient finite field operators for PKC that can be viable to operate in the domain of computing-constrained applications, as the ones found in Pervasive Computing, IoT and AmI. On one side, a PKC-based security layer implemented solely in software on a computing constrained device will induce a high computing overhead that will increase energy waste, a precious resource in Pervasive Computing applications. Additionally, a software-based solution for security could not meet timing requirements as underlying finite field operations are highly time-consuming. On other side, available approaches to build hardware accelerators for PKC are not viable in the context of Pervasive applications, because the primary design goal has been to achieve the fastest implementation at the cost of higher computing hardware resources.

1.6 Hypothesis

The digit-digit approach for hardware implementations of finite field operators in FPGAs requires less hardware area than other processing approaches. These hardware architectures for finite field operators would allow compact hardware implementations of PKC based security layer well suited for low-resources devices, such as the ones commonly found in pervasive computing applications. The proposed hardware architectures must outperform execution time of state of the art software implementations while using reduced area resources than other implementation approaches.

1.7 Research objectives

The objectives that guide this thesis work are presented in this section. The general objective is presented first, following with the specific objectives.

1.7.1 General objective

The general objective of this research work is:

- To develop novel algorithms for arithmetic operations in finite fields and their corresponding compact hardware architectures (finite field operators) well suited to implement public key cryptographic algorithms (e.g., RSA and ECC) in resource-constrained devices.

1.7.2 Specific objectives

To achieve the general objective, the following specific objectives are proposed:

- To identify finite field operators that make up most critical basic building blocks in public key cryptosystems.
- To propose algorithms for finite field arithmetic used in public key cryptography that can yield compact hardware implementations.
- To define the most appropriate approach for designing compact hardware architectures for the proposed algorithms for finite field operators.

1.8 Methodology

Design and implementation of lightweight hardware for finite field operators for PKC is a complex task because there are several algorithms for arithmetic operations and a considerable number of possibilities to their hardware implementations. So, it is necessary to consider, evaluate and select the most adequate. The main design goal in this work is a low area hardware architecture design that leads to obtain low energy consumption.

General strategy

In order to achieve the proposed objectives of this thesis, the following methodology is proposed:

1. State of the art revision.
   (a) Elaborate a critical study of reported public key cryptography algorithms for RSA and ECC, and their underlying arithmetic.
   (b) Study of finite field operators in $\mathbb{F}_p$: multiplication and exponentiation.
   (c) Study of finite field operators in $\mathbb{F}_{2^m}$: multiplication.
   (d) Study of Elliptic Curves operators $\mathbb{E}(\mathbb{F}_{2^m})$.
   (e) Select adequate finite field algorithms for lightweight hardware implementation.
2. Lightweight hardware architecture design.
   (a) Define lightweight hardware/software architectures over selected operators.
   (b) Apply digital design techniques (pipelining, critical path reduction) to improve efficiency of the hardware architectures and reduce the amount of used hardware resources.

(c) Generate test vectors.

(d) Describe lightweight hardware modules and perform hardware simulations. Hardware architectures will be described in VHDL and implemented in FPGAs according to the Xilinx design flow shown in Figure 1.7.

(e) Explore the space design (full-parallel, digit-serial, bit-serial, digit-digit, systolic arrays).

(f) Evaluate the proposed architectures with test vectors.

3. Hardware-software co-design and evaluation.

(a) Develop a hardware-software co-design of proposed lightweight architectures for finite field operators required in RSA and ECC.

(b) Evaluate algorithms and lightweight hardware from full implementation of a PKC security protocol (i.e., authentication in sensor networks).

(c) Report the obtained results.

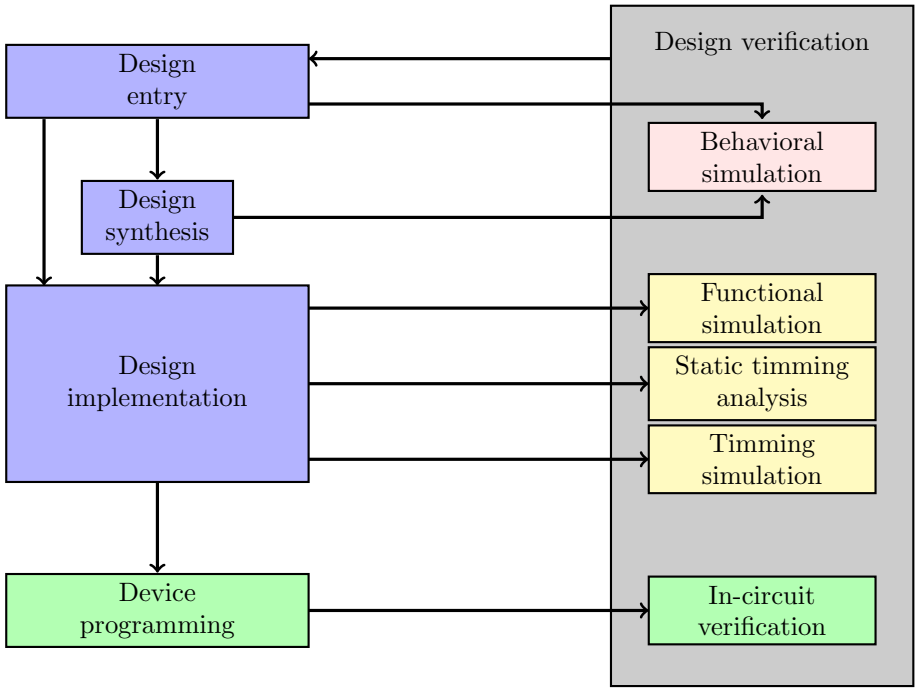

Figure 1.7: FPGA design flow.

1.9 Thesis outline

This thesis document is composed of six chapters divided into several sections and subsections. Following an Introduction, theoretical basis that supports this research are presented in Chapter 2 (Preliminaries). Chapter 3 (State of the Art) presents a review of state of the art works for hardware architectures in FPGAs for finite field operators over $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$. The proposed finite field operators for $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$ are presented in chapter 4 (Proposed lightweight finite field operators). Chapter 5 presents and discuss obtained results, while conclusion are given in Chapter 6. Test vectors for finite field operator in $\mathbb{F}_p$, $\mathbb{F}_{2^m}$, RSA digital signature and ECC key interchange protocol are presented in Appendices.

Chapter 2

Preliminaries

In this chapter, PKC basics needed to frame the research context of this proposal dissertation is presented. PKC algorithms are presented highlighting the importance of efficient realization of finite field operators for deploying high-level security mechanisms in pervasive computing applications.

2.1 Public key cryptography (PKC)

Public key cryptography allows to implement confidentiality through encryption (see Figure 2.1), and integrity, authentication, and non-repudiation by means of the digital signature concept (see Figure 2.2).

Confidentiality

Confidentiality in the context of information security is about protecting information from disclosure to unauthorized parties. Encryption is the conventional technique used in cryptography to ensure information confidentiality. The goal of encryption is to ensure that only authorized parties (in possession of a private key) can read, view or use the information. In public key cryptography, each user has a pair of keys (Kpub, Kpriv), the first one called the public key, and the other the private key. The public key is available to all, while the private key is kept secret. So, assuming that entity A wants to send a message to entity B, A uses the public key from B (B_{Kpub}) to encrypt the message. In this way, the message can be sent over an insecure channel, with certainty that nobody can read the message. Only user B who knows the private key can decrypt the message with B_{Kpriv}. If someone tries to decrypt the message ciphered other than B_{Kpriv}, decryption algorithm's output should be *unreadable* [24].

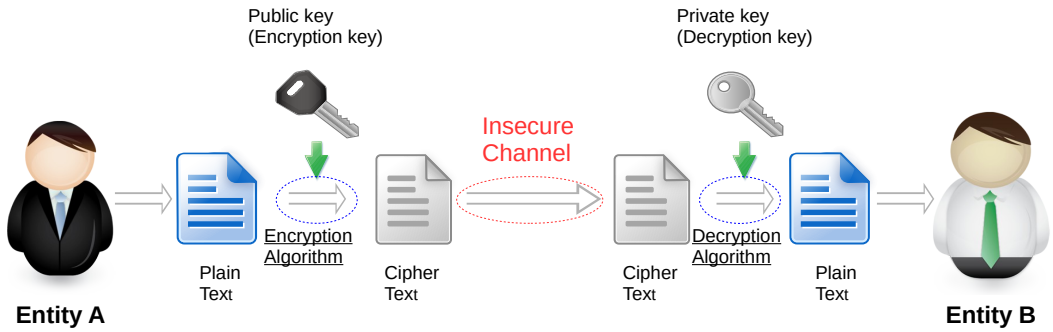

Figure 2.1: Public key cryptography.

Integrity

Hash function only provides data integrity. As it can be seen in Figure 2.2, sender uses a key and an algorithm to sign the information and thus ensure its integrity. Consequently, using a validation key and a validation algorithm, any other entity can verify the authenticity of the signature over the information. If the information is modified, the validation process fails, revealing that the integrity of information has been violated [25].

Authentication (and Nonrepudiation)

As explained in previous security service, once information has been signed, receptor can ensure that the sender is originator of the message only if signature is correctly verified, this guarantees authentication. Furthermore, the sender can not deny message origin. Thus, non-repudiation service is also ensured [25].

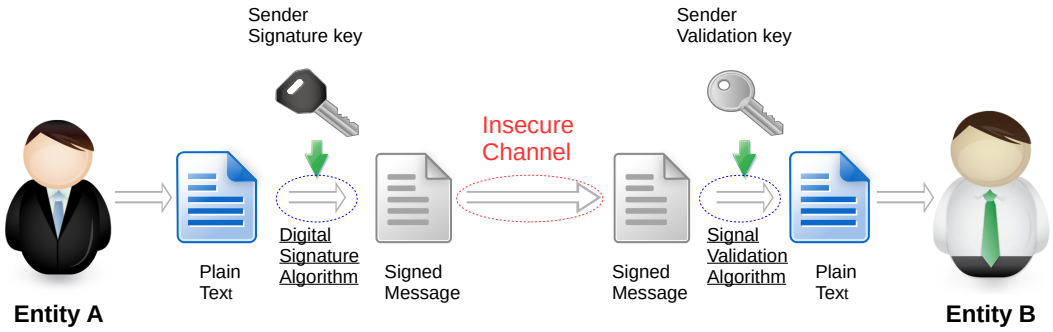

Figure 2.2: Digital signature.

Formal definition of PKC

The formal definition of a PKC scheme is:

Definition 1 *Let $k \in \mathbb{N}$ be a **security parameter**. A **PKC encryption scheme** is defined by the following spaces (all depending of security parameter k) and algorithms [26].*

M_k	space of all possible messages;
PK_k	space of all possible public keys;
SK_k	space of all possible private keys;
C_k	space of all possible ciphertexts;
KeyGen	a randomised algorithm that takes a security parameter k, runs in expected polynomial-time (i.e., $O(k^c)$ bit operations for some constant $c \in N$) and outputs $\{pk, sk\}$ pair with $pk \in PK_k$ and $sk \in SK_k$;
Encrypt	a randomised algorithm that takes as input $m \in M_k$ and pk, runs in expected polynomial-time (i.e., $O(k^c)$ bit operations for some constant $c \in N$) and outputs a ciphertext $c \in C_k$;
Decrypt	an algorithm (not usually randomised) that takes $c \in C_k$ and sk, runs in polynomial-time and outputs either $m \in M_k$ or invalid ciphertext symbol $\perp$.

It is required that Decrypt(Encrypt(m, pk), $sk) = m$ if (pk, sk) is a matching key pair.

Diffie and Hellman [27] proposed the realization of *Public Key Cryptosystem* using one-way functions and trapdoor information. A one-way function is an invertible function that is easy to compute, but its inverse is difficult to compute [28]. In this context, difficult means (informally) that any algorithm that attempts to compute the inverse will take a lot of time, e.g., the age of the universe. With trapdoor information, computation of one-way function inverse is considered easy to perform, see Figure 2.3. Two of the most popular public key cryptography schemes are RSA and ECC, which are explained later.

The discrete logarithm problem (DLP), elliptic curve discrete logarithm problem (ECDLP) and the integer factorization problem are the mathematical problems used as one-way trapdoor functions in today known public key cryptography algorithms. These mathematical problems and their basis are explained in the next sections.

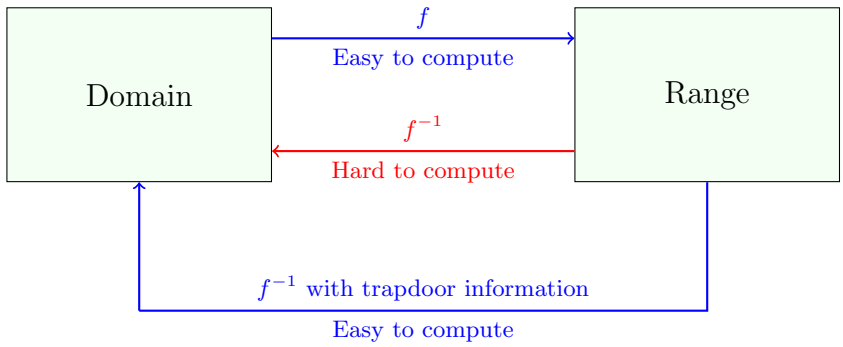

Figure 2.3: One-way trapdoor function.

2.1.1 Groups and cyclic groups

Public key cryptosystems rely on the hardness of some mathematical problems, defined over abstract algebraic structures such as groups and finite fields. Generally, a PKC system implementation is mainly related to arithmetic operations defined in those abstract algebraic structures. This section presents the main concepts of groups and finite fields related to cryptography applications, as well as the definition of the discrete logarithm problem.

Definition 2 *A group is a set of elements* G *together with a binary operation* $\diamond$*. Properties that a group obeys are shown in Figure 2.4.*

Group	
1. Closure:	• If $a, b \in G$, then $a \diamond b \in G$.
2. Associativity:	• $a \diamond (b \diamond c) = (a \diamond b) \diamond c$ for all $a, b, c \in G$.
3. Identity element:	• $\exists e \in G$ such that $a \diamond e = e \diamond a = a$ for all $a \in G$.
4. Inverse element:	• $\forall a \in G$, $\exists y \in G$ such that $a \diamond y = y \diamond a = e$.

Figure 2.4: Group properties.

A group G is said to be an **abelian** (or **commutative**) group if:

$$a \diamond b = b \diamond a, \text{ for all } a, b \in G$$

A group G is said to be **finite** if it contains only finitely number of elements. Otherwise, it is said that the group G is **infinite**. The group G is of **order** n if it

contains exactly n elements. In the context of public key cryptography, finite groups are commonly used.

As examples:

1. The pair $\{\mathbb{Z}, +\}$ is an abelian group in which the identity element is 0, and inverse of $a \in \mathbb{Z}$ is $-a$.

2. The pair $\{\mathbb{R}^+, \times\}$ is an abelian group in which the identity element is 1 and inverse of $a \in \mathbb{R}^+$ is a^{-1}.

A typical operation in groups structures is to apply the group operation of an element "g" with itself n times. Let g be an element of a group G, and let n be a positive integer. Then g^n means that the operation $\diamond$ is applied to g, n times:

$$g^n = \underbrace{g \diamond g \diamond \cdots \diamond g}_{\text{n times}} \tag{2.1}$$

For some groups, notation ng is commonly used to detonate that the group operation is applied n times to g, but this is just a matter of notation.

Definition 3 *Let $\{G, \diamond\}$ be a group, and let H be a nonempty subset of G such that [29]:*

- $\forall a, b \in H$, $a * b \in H$
- $\forall a \in H$, $a^{-1} \in H$

Then H is called a subgroup of G.

H is itself a group using exactly the same binary operation to that in the larger ("supergroup") G.

As example:

1. Let $\{G, \diamond\}$ be a group, let $m > 0$ be an integer, and define

$$H = \{g^m | g \in G\}. \tag{2.2}$$

Then H is a subgroup of G.

Finite cyclic groups

Let $\{G, \diamond\}$ be a finite group, and let $a \in G$. The set $\{a^m | m \in \mathbb{Z}\}$ form a subgroup. This subgroup is called subgroup generated by a, and is commonly denoted by [29].

$$< a >= \{e, a, a^2, a^3, ...\}. \tag{2.3}$$

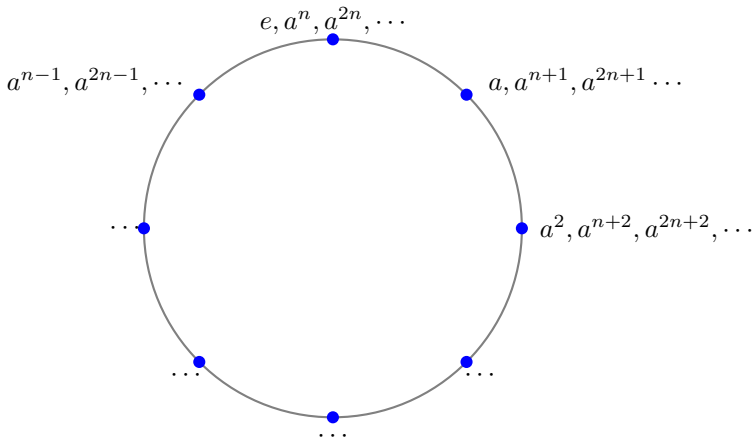

Figure 2.5: A finite cyclic group generated by a.

Then elements $\{e, a, a^2, a^3, \cdots, a^n, \cdots\}$ cannot be all different for $m > n$. The smaller positive integer n such that $a^n = e$ is called the order of a. If n is the order of a, then the finite group generated by a is:

$$< a >= \{e, a, a^2, a^3, \cdots, a^{n-1}\}. \tag{2.4}$$

A graphical representation of a cyclic group generated by the element a is shown in figure 2.5.

2.1.2 Finite fields ($\mathbb{F}_q$) as groups construction

Definition 4 *A field is a set $\mathbb{F}$ with two operations $+$ and $\times$ satisfying the following properties:*

- $\mathbb{F}$ *is an Abelian group under $+$ with identity element 0 (zero).*
- $\mathbb{F}^*$ *(the nonzero elements of $\mathbb{F}$) form an Abelian group under $\times$, with identity element 1.*
- $\times$ *distributes over $+$, i.e., $a \times (b + c) = a \times b + a \times c$ for any $a, b, c \in \mathbb{F}$.*

The number of elements in a field is called the *order* of the field. If the field has a finite number of elements, then it is called a finite field, otherwise it is called an infinite field. Infinite fields include real numbers, rational numbers, complex numbers, etc. A Finite field denoted as $\mathbb{F}_q$ is usually referred as a Galois Field denoted as $\mathbb{GF}(q)$. A Finite field of order q exists if and only if q is the power of a prime. Furthermore, if q is the power of a prime number, the finite field of order q is unique. Finite fields are of special interest for cryptographic applications since they are used in most widely used cryptosystems RSA and ECC. In next section, the finite field $\mathbb{F}_p$ is presented.

2.1.3 The prime field $\mathbb{F}_p$ and the multiplicative group $\mathbb{F}_p^*$

Before describing the finite field $\mathbb{F}_p$, it is required to give some useful definitions.

If a is an integer and n is a positive integer, we define $a \bmod n$ to be the remainder when a is divided by n. The integer n is called the **modulus** [21].

Definition 5 *Two numbers a and b are **congruent modulo n** (written $a \equiv b \bmod n$) if $a - b$ is a multiple of n. The integer n is called the modulus of the congruence and is assumed to be positive.*

Let be a an integer with $a < n$, **multiplicative inverse** of a mod n is the number a^{-1} such that $a \times a^{-1} \equiv 1 \bmod n$. It is well studied that a has a unique multiplicative inverse $\bmod n$ if and only if $\gcd(a, n) = 1$ [30]. If n is a prime number p then all numbers $a < p$ have a multiplicative inverse since $\gcd(a, p) = 1$ for all a.

Definition 6 *Set $\mathbb{Z}/n\mathbb{Z}$ is defined as the set of integers modulo n [28]:*

$$\mathbb{Z}/n\mathbb{Z} = \{0, 1, 2, \ldots, n-1\}. \tag{2.5}$$

It is possible to add and multiply $\mathbb{Z}/n\mathbb{Z}$ elements as integers and then divide the result by n and take the remainder in order to obtain an element in $\mathbb{Z}/n\mathbb{Z}$ [28].

Table 2.1: Multiplication modulo 5.

x	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

Table 2.2: Addition modulo 5.

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Tables 2.1 and 2.2 illustrate addition and multiplication modulo 5 over $\mathbb{Z}/5\mathbb{Z}$.

Element $a \in \mathbb{Z}/n\mathbb{Z}$ has inverse modulo n if and only if gcd(a, n) = 1. Numbers that have inverses are called units. Set of all units is denoted by:

$$(\mathbb{Z}/n\mathbb{Z})^* = \{a \in \mathbb{Z}/n\mathbb{Z} | \gcd(a, m) = 1\} \tag{2.6}$$

$$= \{a \in \mathbb{Z}/n\mathbb{Z} | a \text{ has inverse modulo } n\} \tag{2.7}$$

The set $(\mathbb{Z}/n\mathbb{Z})^*$ is called **group of units modulo** n [28]. When $n = p$ is a primer number, all numbers $0 < a < p - 1$ are relative primes to p and belong to the set $(\mathbb{Z}/p\mathbb{Z})^*$.

As example:

- The group of units modulo 7 is:

$$(\mathbb{Z}/7\mathbb{Z})^* = \{1, 2, 3, 4, 5, 6\} \tag{2.8}$$

- But, if n is not a prime number, there are elements that do not belong to $(\mathbb{Z}/n\mathbb{Z})^*$. For example, the group of units modulo 24 is:

$$(\mathbb{Z}/24\mathbb{Z})^* = \{1, 5, 7, 11, 13, 17, 19, 23\} \tag{2.9}$$

The quantity of elements in the group of units $\mathbb{Z}/n\mathbb{Z}$ is essential for some public key cryptosystems. The function that computes the quantity of units elements in $\mathbb{Z}/n\mathbb{Z}$ is:

Definition 7 *Euler's phi function is the function $\phi(n)$ defined by the rule:*

$$\phi(n) = \#(\mathbb{Z}/n\mathbb{Z})^* = \#\{0 \leqslant a < n | \gcd(a, n) = 1\}. \tag{2.10}$$

Division in $\mathbb{Z}/n\mathbb{Z}$ can be a problem, since it is possible divide by a in $\mathbb{Z}/n\mathbb{Z}$ only if $\gcd(a, n) = 1$. However, if $n = p$ is a prime number, then it is possible divide by every non-zero element in $(\mathbb{Z}/p\mathbb{Z})^*$. If zero is removed from $\mathbb{Z}/p\mathbb{Z}$, remaining elements are units and closed under multiplication. Then, if p is prime, the set $\mathbb{Z}/p\mathbb{Z}$ of integers modulo p together with addition, subtraction, multiplication and division rules, is an example of a field.

In fact, if p is prime, then the set $\mathbb{Z}/p\mathbb{Z}$ is a *field*. The field $\mathbb{Z}/p\mathbb{Z}$ of integers modulo p has only finite elements. It is a finite field, and it is often denoted as $\mathbb{F}_p, \mathbb{GF}(p)$ or $\mathbb{Z}_p$. In the rest of this document, the multiplicative group will be denoted as $\mathbb{F}_p^*$ and the prime finite field will be denoted as $\mathbb{F}_p$.

Definition 8 *Let p be a prime number. Then there exists an element $g \in \mathbb{F}_p^*$ whose powers give every element of $\mathbb{F}_p^*$ [28].*

$$\mathbb{F}_p^* = \left\{1, g, g^2, g^3, \ldots, g^{p-2}.\right\} \tag{2.11}$$

Element or elements with this property are called *primitive roots* or *generators* of $\mathbb{F}_p^*$.

2.1.4 Discrete Logarithm Problem in multiplicative groups (DLP-$\mathbb{F}_p^*$)

The Discrete Logarithm Problem (DLP) is defined as:

Definition 9 *Let G be a finite group with group operation denoted by $\star$. The Discrete logarithm problem is to determine, for any two given elements g and h in G, an integer x satisfying [28]:*

$$\underbrace{g \star g \star g \star \ldots g}_{x \text{ times}} = h \tag{2.12}$$

The discrete logarithm problem for the multiplicative $\mathbb{F}_p^*$

The DLP for the multiplicative group is defined as:

Definition 10 *Let g be a primitive root for $\mathbb{F}_p^*$ and let h be a non-zero element of $\mathbb{F}_p^*$. The Discrete Logarithm Problem (DLP) in $\mathbb{F}_p^*$ is the problem of finding an exponent x such that [31, 28]:*

$$g^x \equiv h \mod p. \tag{2.13}$$

It is believed that the discrete logarithm problem is hard to solve [32]. However, there is no proof of this statement. There are works that try to solve the discrete logarithm problem in an efficient way, but no one has succeeded, for example [33, 34].

2.1.5 DLP-F_p* based cryptosystems (DH, RSA, RSA-Digital Signature)

In this section, cryptosystems based in multiplicative group $\mathbb{F}_p^*$ are presented.

Diffie-Hellman (DH)

The dilemma of interchanging a key over an insecure channel was resolved by Diffie-Hellam (DH) key exchange algorithm [27]. The key interchange between parts can be used in some other private key cryptosystem such as AES or 3DES. As a traditional example, Alice and Bob want to interchange a key over an insecure channel that is monitored by Eve. Eve can read all the information that Alice and Bob interchange. Diffie and Hellman, in a brilliant insight, proposed that discrete logarithm problem difficulty over $\mathbb{F}_p^*$ provides a possible solution.

First, Alice and Bob need to agree on a prime number p and a non-zero primitive root g. Values p and g are made of public knowledge by Alice and Bob, so Eve knowns them. Second step is that Alice chooses a secret a that she does not reveal to anyone,

and Bob must do the same, choosing a secret b that he does not reveal to anyone. Now, Bob and Alice can compute:

$$\underbrace{A \equiv g^a \bmod p}_{\text{Alice's computation}} \qquad \text{and} \qquad \underbrace{B \equiv g^b \bmod p}_{\text{Bob's computation}} \tag{2.14}$$

In the next step, Alice sends A to Bob, and Bob sends B to Alice. The values A and B are sent over an insecure channel, so Eve knowns these values. Finally, Alice and Bob use their secret values to compute:

$$\underbrace{A' \equiv B^a \bmod p}_{\text{Alice's computation}} \qquad \text{and} \qquad \underbrace{B' \equiv A^b \bmod p}_{\text{Bob's computation}} \tag{2.15}$$

The values computed by Alice and Bob are the same, since:

$$A' \equiv B^a \equiv (g^b)^a \equiv g^{ab} \equiv (g^a)^b \equiv A^b \equiv B' \bmod p \tag{2.16}$$

The common value $A' = B'$ is the secret key exchanged. If Eve wants to know that secret key she needs to solve the Discrete Logarithm Problem (DLP) [28].

RSA

The RSA public key cryptosystem was developed in 1977 [35]. RSA is named after its (public) inventors, Ron Rivest, Adi Shamir, and Leonard Adleman. This algorithm is used to encryp/decrypt a message as well as to implement digital signatures. Security of RSA is based on solving the integer factorization problem. Table 2.3 summarizes main process in the RSA cryptosystem.

The security of RSA depends on the following dichotomy [28].

Setup. Let p and q be large primes, let $N = pq$, and let e and c be integers.

Problem. Solve the congruence $x^e \equiv c \mod N$ for the variable x.

Easy. Bob, who knows p and q values, can easily solve for x.

Hard. Eve, who does not know p and q values, cannot easily find x.

Dichotomy. Solving $x^e \equiv c \mod N$ is easy for a person who possesses certain extra information, but it is apparently hard for all other people.

Security of RSA algorithm is based on the problem of factoring large numbers. If it is possible to factor N, then it is possible to use p and q to compute d. Knowing d

Table 2.3: RSA public key cryptosystem.

Bob	Alice
Key Creation	
Choose secret primes p and q. Choose encryption exponent e with $\gcd(e,(p-1)(q-1))=1$ Publish $N=pq$ and e.	
Encryption	
	Choose plaintext m. Use Bob's public key (N,e) to compute $c \equiv m^e \mod N$. Send ciphertext c to Bob.
Decryption	
Compute d satisfying $ed \equiv 1 \mod (p-1)(q-1)$. Compute $m' \equiv c^d \mod N$, Then m' equals the plaintext m.	

is computationally equivalent to factoring N. Integer factorization has been a topic of research for hundreds of years. Nowadays, for enough large numbers, there is still missing a polynomial time algorithm that can factor a large number into its two prime factors.

RSA digital signature

RSA encryption and RSA digital signature schemes have been described in [35]. In this section the RSA digital signature scheme is presented. Examples presented here are in the context of embedded devices, so FPGAs are used as entities instead of using Bob and Alice as in the previous examples.

A sensor node implemented with an FPGA needs to send a message to a server in the network, implemented with an FPGA too. In this example we call them FPGA Sensor and FPGA Server. FPGA sensor sends a message to FPGA server, but how can FPGA server verify that FPGA sensor is who send the message? The answer is with RSA digital signature scheme [35]. The FPGA sensor must publish a parameter N that is the product of the to secret primes p and q. A verification exponent v must be published too for FPGA sensor. Since the FPGA sensor knows factorization of N it can

solve the congruence:

$$sv \equiv 1(\mathrm{mod}\,(p-1)(q-1)). \tag{2.17}$$

So s is FPGA sensor's *signing exponent* and v is its *verification exponent*.

If D is a digital message to be signed D need to be encoded as an integer in the range $1 < D < N$ and FPGA sensor can compute S signature of D document.

$$S \equiv D^s \bmod N. \tag{2.18}$$

So, the FPGA server can validate S signature of digital message D by computing:

$$S^v \equiv D^{sv} \equiv D \bmod N \tag{2.19}$$

Table 2.4 resume steps to create the public and private keys for RSA digital signature explained previously.

Table 2.4: FPGA sensor key creation

1	• Choose secret primes p and q.
2	• Choose verification exponent v $\gcd(v(p-1)(q-1)) = 1$
3	• Compute s $sv \equiv 1(\mathrm{mod}\,(p-1)(q-1))$
4	• Publish $N = pq$ and v

Figure 2.6 shows a graphical view of an RSA digital signature in which the sensor FPGA create a digital signature S using its signing exponent s for the digital message D and send them to the FPGA server. The FPGA server uses the FPGA sensor's verification exponent v and the digital signature S to obtained the verification value V. If V and D are the same, the FPGA server can assure that the FPGA sensor signed the digital document D.

2.1.6 The role of finite field operators in DLP-$\mathbb{F}_p^*$ based cryptosystems

Finite field arithmetic is the basis of public key cryptosystems such as Diffie-Hellman, RSA, ElGamal, RSA digital signature, etc. The principal operation in the DLP-$\mathbb{F}_p^*$ based cryptosystems is the modular exponentiation, and due to the significant size numbers (1024-4096 bits) used in these public key cryptosystems, this operation is costly. There have been different approaches to implement modular exponentiation for

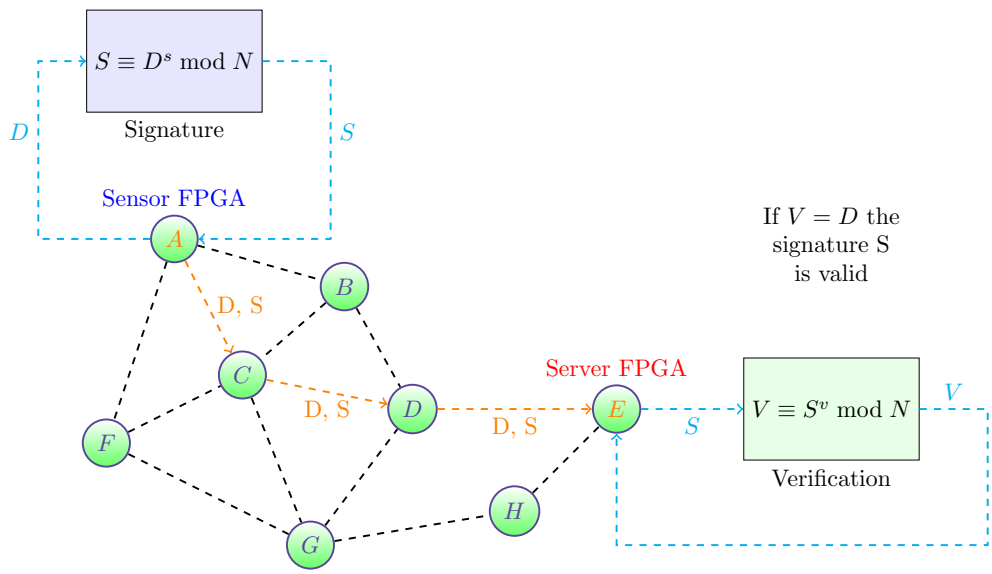

Figure 2.6: RSA digital signature scheme. D = Document, s = *signing exponent*, S = signature, v = *verification exponent*, V = Verification

public key cryptosystems. Finite field operators have been proposed in software and hardware. However, a typical software finite field operator for modular exponentiation in 8-bit microprocessors takes 20 seconds approximately [36]. Consequently, in time restricted applications it commonly uses custom hardware to accelerate its computation. Design and implementation for finite field operators is a crucial task when public key cryptosystems are deployed.

2.2 Curve based Cryptography: a case of PKC

Curve based cryptography has become increasingly popular in the research community in recent years [37]. The most popular curve based cryptography is Elliptic Curve Cryptography (ECC) and most recently Hyperelliptic Curve Cryptosystems (HECC). The use of elliptic curves for cryptography was independently proposed by Miller [38] and Koblitz [39]. ECC is commonly used for key exchange over an insecure channel and for digital signatures [37]. Furthermore, ECC has become the preferred public key cryptosystem for applications where resource-constrained devices are used. Mainly because ECC uses relatively short operand length compared to other public key cryptosystems such as RSA. In this section a brief introduction to elliptic curves is presented; for a more detailed study in ECC the reader can consult [28, 40].

The set of solutions (points) to an equation of the form:

$$Y^2 = X^3 + AX + B \tag{2.20}$$

is called an **elliptic curve** E [28]. Graphical representation of two different elliptic curves are shown in Figure 2.7.

Two points P and Q of an elliptic curve E can be "added" to produce a third point $R \in E$. Figure 2.7 shows the "addition" operation of two points in an elliptic curve over real numbers. Geometry is the most natural to describe the addition operation of points $P, Q \in E$ on elliptic curves (see Figure 2.7):

- **Adding P and Q when $P \neq Q$.**
  A line L that intersect P and Q can be drawn. The line L intersects E in a third point R'. The point R that is a reflect of R' through x-axis is the sum $R = P + Q$.

- **Adding P and Q when $P = Q$.**
  If $P = Q$ then the addition is called a "double", and is computed as follow: the line L can be computed as the tangent line to E at the point P. Then, L intersect E at P and another point R', so $P + P$ sum is R point that reflects R' through x-axis, $R = P + P$.

- **Adding P and Q when $Q = P'$.**
  If L is the line that intersect P and P' then L do not intersect E in a third point. The solution has been create an extra point O that lives at "infinity"[28]. Point O does not exist in XY-plane, but it is pretended that it lies on every vertical line. So, it is possible to add $P + P' = O$, furthermore, $P + O = O + P = P$.

For two distinct points, $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ belong to an elliptic curve E , with $P \neq -Q$, the slope of the line L that joins them is $\Delta = (y_Q - y_P)/(x_Q - x_P)$. There is exactly one other point where L intersects the elliptic curve, and is the negative of P and Q sum. After some algebraic manipulation, sum $R = P + Q$ is expressed as [21]:

- When $P \neq Q$:

  Elliptic curves point addition:

$$R = (x_R, y_R) = \tag{2.21a}$$

$$x_R = \Delta^2 - x_P - x_Q \tag{2.21b}$$

$$y_R = -y_P + \Delta(x_P - x_R) \tag{2.21c}$$

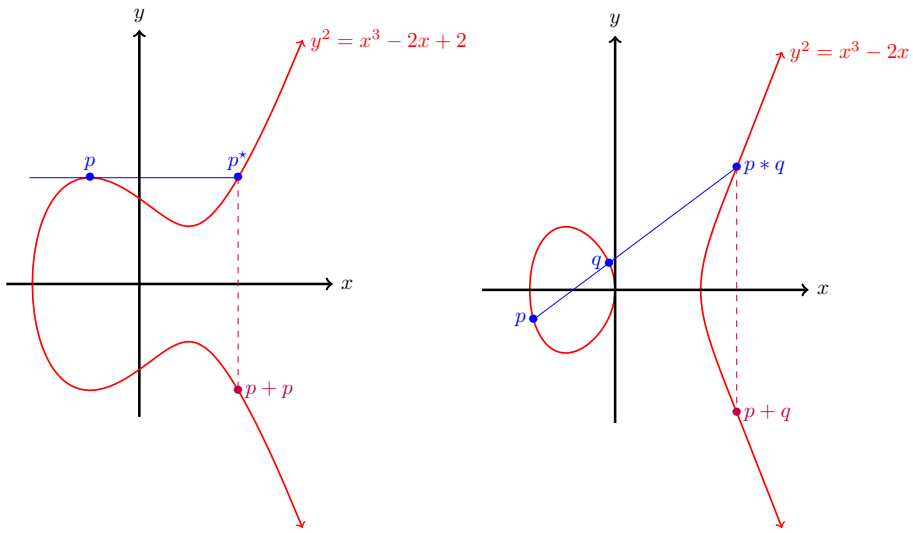

Figure 2.7: Graphical view of point double/addition in elliptic curves.

- When $P = Q$: $P + Q = P + P = 2P = R$. When $y_P \neq 0$, R is:

  Elliptic curves point double:

$$R = (x_R, y_R) = \tag{2.22a}$$

$$x_R = \left(\frac{3x_p^2 + a}{2Y_p}\right)^2 - 2x_p \tag{2.22b}$$

$$y_R = \left(\frac{3x_p^2 + a}{2Y_p}\right)(x_p - x_R) - y_P \tag{2.22c}$$

Scalar point multiplication $Q = kP$ is the main operation in ECC. Q is computed by k-times point addition operation [41]:

$$Q = kP = \underbrace{P + P + \cdots + P}_{k-\text{times}}. \tag{2.23}$$

2.2.1 Additive group of elliptic curve points

Let E be an elliptic curve. Then addition law on E has the following properties [28]:

- Identity. $P + O = O + P = P$, for all $P \in E$.
- Inverse. $P + (-P) = O$, for all $P \in E$.
- Associative $(P + Q) + R = P + (Q + R)$ for all $P, Q, R \in E$
- Commutative $P + Q = Q + P$ for all $P, Q \in E$

In other words, points of an elliptic curve E together with the addition law form an Abelian group.

2.2.2 Elliptic curves over a finite field $(E(F_q))$

Elliptic curves are defined over a field K. In the previous section, "addition" operations were presented for elliptic curves over real numbers, which show a graphical view of the addition points in an elliptic curve. However, for practical cryptographic applications, elliptic curves are defined over a finite field F_q, $q = p^m$. When $m = 1$, finite field $\mathbb{F}_p$ is known as the prime field. When $p = 2$ the finite field $\mathbb{F}_{2^m}$ is known as the binary field. Prime field $\mathbb{F}_p$ and binary field are commonly used for ECC applications.

2.2.3 Elliptic curves over prime field $(E(F_p))$

Elliptic curves over finite fields $\mathbb{F}_{q^m}$ are used in real applications. In this section, elliptic curves over prime field $\mathbb{F}_p$ are presented.

Definition 11 *Let $p \geqslant 3$ be a prime. An* elliptic curve *over $\mathbb{F}_p$ is the set of solutions for next equation [28]:*

$$E : Y^2 = X^3 + AX + B \qquad \textit{with } A, B \in \mathbb{F}_p \textit{ satisfying } 4A^3 + 27B^2 \neq 0. \tag{2.24}$$

The set of points on E with coordinates in $\mathbb{F}_p$ is the set:

$$E(\mathbb{F}_p) = \{(x, y) | x, y \in \mathbb{F}_p \text{ satisfy } y^2 = x^3 + Ax + B\} \cup \{O\} \tag{2.25}$$

When elliptic curves are defined over finite fields, they have only finite points, and they are hard to draw in a geometrical way. Figure 2.8 show a comparison of two elliptic curves, first in real numbers $\mathbb{R}$ and second in prime field $\mathbb{F}_p$.

Point addition and double formulas can geometrically be derived as with elliptic curves over real numbers $\mathbb{R}$, that leads to a field of mathematics called algebraic geometry. However, for practical purposes, explicit formulas generated for real numbers are

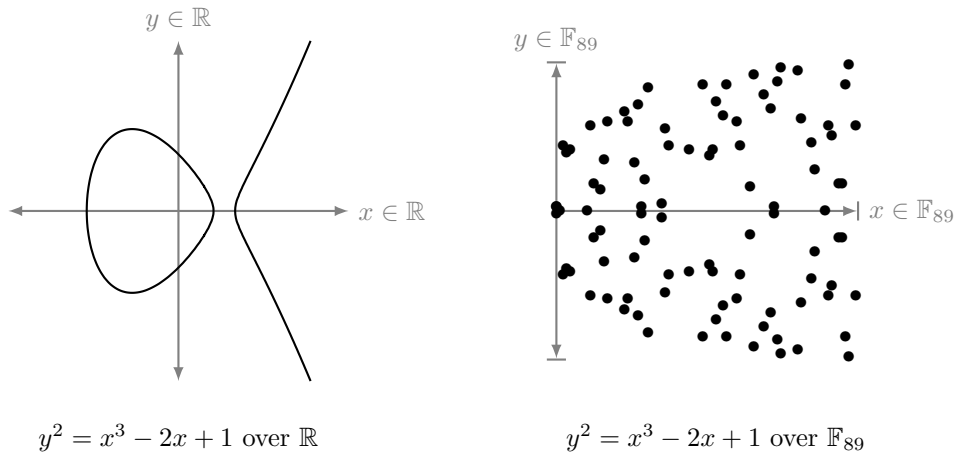

Figure 2.8: Elliptic curve over real numbers and over the prime field.

used but realizing all operations in finite field $\mathbb{F}_p$. Next rules determine elliptic curve point addition over $\mathbb{E}(\mathbb{F}_p)$. For all points $P, Q \in E$:

1. $P + O = P$.
2. If $P = (x_p, y_p)$ then $-P = (x_p, -y_p)$, and $P + (-P) = O$.
3. If $P = (x_p, y_p)$ and $Q = (x_q, y_q)$ with $P \neq -Q$, then $R = P + Q = (x_r, y_r)$ is defined by the following rules.

$$\begin{aligned} x_r &= (\lambda^2 - x_p - x_q) \mod p \\ y_r &= (\lambda(x_p - x_r) - y_p) \mod p \end{aligned}$$

Where:

$$\lambda = \begin{cases} (\frac{y_q - y_p}{x_q - x_p}) \mod p & \text{if } P \neq Q \\ (\frac{3x_p^2 + a}{2y_p}) \mod p & \text{if } P = Q \end{cases}$$

2.2.4 Elliptic curves over binary field $(E(F_{2^m}))$

Elliptic curves over binary fields are the set of solutions of an equation of the form:

$$E : Y^2 + XY = X^3 + AX^2 + B. \tag{2.26}$$

For a binary curve defined over $\mathbb{F}_{2^k}$, variables and coefficients all take on values in $\mathbb{F}_{2^k}$ and calculations are performed over $\mathbb{F}_{2^k}$. Rules for addition of elliptic curves over $\mathbb{F}_{2^k}$ are determinated by the following formulas. For all points $P, Q \in E(\mathbb{F}_{2^k})$:

1. $P + O = P$.

2. If $P = (x_p, y_p)$ then $-P = (x_p, x_p + y_p)$, and $P + (-P) = O$.

3. If $P = (x_p, y_p)$ and $Q = (x_q, y_q)$ with $P \neq -Q$ and $P \neq Q$, then $R = P + Q = (x_r, y_r)$ is defined by following rules.

$$\begin{aligned} x_r &= \lambda^2 + \lambda + x_p + x_q + a \\ y_r &= \lambda(x_p + x_r) + x_r + y + p \end{aligned}$$

   Where:

$$\lambda = \frac{y_q + y_p}{x_q + x_p}$$

4. If $P = (x_p, y_p)$ then $R = 2P = (x_r, y_r)$ is determined by the following rules:

$$\begin{aligned} x_r &= \lambda^2 + \lambda + a \\ y_r &= x_p^2 + (\lambda + 1)x_r \end{aligned}$$

2.2.5 Discrete Logarithm Problem in additive elliptic curve groups (DLP-$E(F_q)$)

The discrete logarithm problem was defined in an alternative group by Miller and Koblitz [39, 42] in the mids 80s. The new group was formed by points of an elliptic curve, defined over a finite field.

Definition 12 *Let* E *be an elliptic curve (over a finite field), and let* P *and* Q *be points with* $P, Q \in E$. *The* Elliptic Curve Discrete Logarithm Problem (ECDLP) *is the problem of finding an integer* n *such that* $Q = nP$ *[28].* n *is called the elliptic discrete logarithm of* Q *with respect to* P.

Some times, n is denoted by:

$$n = \log_P(Q). \tag{2.27}$$

There may be points $P, Q \in E$ such that Q is not a multiple of P. In this case, $\log_P(Q)$ is not defined. So for practical applications P generally is a generator of E, so, the discrete logarithm of Q always exists.

Table 2.5: Elliptic curves cryptography, ElGamal encryption and decryption

Public Parameter Creation	
A trusted party chooses and publishes a (large) prime p, an elliptic curve E over $\mathbb{F}_p$, and a point P in $E(\mathbb{F}_p)$.	
Bob	**Alice**
Key Creation	
Chooses a private key n_B. Computes $Q_B = n_B P$ in $\mathbb{E}(\mathbb{F}_p)$. Publishes public key Q_B.	
Encryption	
	Chooses plain text $M \in \mathbb{E}(\mathbb{F}_p)$. Chooses an ephemeral key k. Uses Bob's public key Q_B to compute $C_1 = kP \in \mathbb{E}(\mathbb{F}_p)$. and $C2 = M + kQ_B \in \mathbb{E}(\mathbb{F}_p)$. Sends cipher text $(C1, C2)$ to Alice.
Decryption	
Computes $C2 - n_B C_1 \in \mathbb{E}(\mathbb{F}_p)$. As $n_B C_1 = n_B(kP) = K(n_B P) = kQ_B$, this quantity is equal to M.	

2.2.6 DLP-$E(F_q)$ based cryptosystems (Elliptic curve cryptography - ECC)

Elliptic Curve Cryptography can be used in some cryptosystems such as Elliptic ElGamal and Elliptic Curves Diffie-Hellman key exchange [28] which are explained in this section

2.2.7 Elliptic curves ElGamal public key cryptosystem

ElGamal is a cryptosystem based on the discrete logarithm problem. Table 2.5 shows ElGamal version for Elliptic Curves.

2.2.8 ECDH key exchange

As in RSA digital signature scheme, examples presented here are in the context of embedded devices, so FPGAs are used as entities instead of using Bob and Alice as in

the previous example.

An elliptic curve $\mathbb{E}(\mathbb{F}_{2^m})$ and a particular point P must be agreed between FPGA sensor and FPGA server before to realize the Elliptic Curve Diffie-Hellman (ECDH) key exchange. Then FPGA sensor and FPGA server select a secret integer each says n_A and n_B respectively. So they can compute the scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$

$$\underbrace{Q_A = n_A P}_{\text{FPGASensor}} \qquad \text{and} \qquad \underbrace{Q_B = n_B P}_{\text{FPGAServer}} \tag{2.28}$$

and exchange Q_A and Q_B values. FPGA sensor then compute $n_A Q_B$ value with its secret integer n_A while FPGA server compute $n_B Q_A$ value with its secret integer n_B. Now they have a shared secret value

$$n_A Q_B = (n_A n_B)P = n_B Q_A. \tag{2.29}$$

A graphical view of ECDH scheme in a sensor network is presented in Figure 2.9. The shared secret key can be used in symmetric encryption schemes such as AES or 3DES.

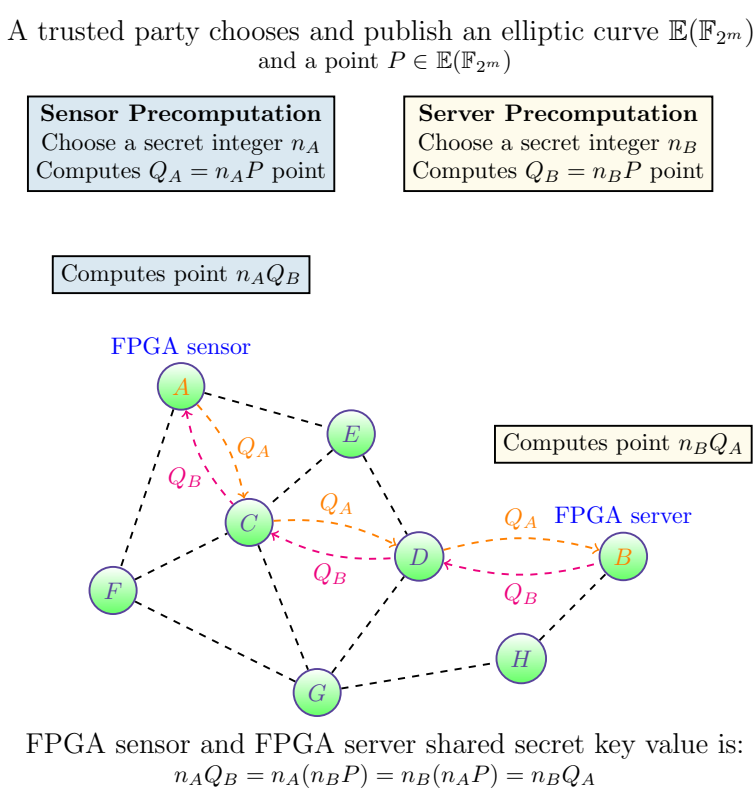

Figure 2.9: Elliptic Curve Diffie-Hellman key exchange protocol.

Table 2.6: Required finite field operations in point addition and point double in elliptic curves cryptography.

Finite field operation	Point addition	Point double
Multiplication	2	4
Addition	6	1
Subtraction	6	4
Inversion	1	1
Square	1	2

Table 2.7: Finite field operations required in a point multiplication in elliptic curve cryptography over $\mathbb{F}_p$ with 256-bits in size.

Finite field operation	Required operations
Multiplication	1536
Addition	256
Subtraction	2560
Inversion	512
Square	768

In both algorithms, ECDH and ECC ElGamal, the main and most costly operation is scalar multiplication. This operation is performed using a series of underlying group and finite field operations (e.g., sums, inversions, squaring, multiplications, and exponentiation over $\mathbb{F}_p$ or $\mathbb{F}_{2^m}$).

2.2.9 Finite field operators role in DLP-$E(F_q)$ based cryptosystems

In elliptic curve cryptosystems, the most costly operation is the scalar multiplication. Scalar multiplication is achieved by performing addition, multiplication, squaring and inversion in the finite field $\mathbb{F}_q$.

According to formulas presented in section 2.2.3 for elliptic curves over prime field $\mathbb{F}_p$, required operations for point addition and point double in elliptic curves are shown in Table 2.6.

Using conventional exponentiation algorithms such as binary exponentiation or Montgomery powering ladder [43], a point multiplications in an elliptic curve with 256 bit in size may require up 256 point additions and 256 point doubling. However, required operations for computing a point multiplications in an elliptic curve with 256 bits in size are shown in Table 2.7.

As it can be seen in Tables 2.6 and 2.7, the required finite field operations in ECC are high, and are the most time-consuming operations in those ECC public key cryptosystems. As a consequence, in the literature, several works have proposed different approaches to construct finite field hardware accelerators but mainly motivated to achieve faster computations at the cost of higher hardware resources. Even at present, motivation remains to find the most appropriate architecture for this operation depending on the context of application and the underlying computing platforms.

In the context of pervasive computing, hardware implementations are restricted to use few hardware resources. So, it is necessary to keep the use of hardware as low as possible and achieve a better performance than software implementations.

The main aim of known hardware solutions is to provide high levels of security, without considering requirements of resource-constrained devices. A new field called lightweight cryptography (LWC) is an emerging research field, and focuses on designing schemes for devices with constrained capabilities [23].

2.3 Finite field operators as key elements in PKC realizations for constrained environments

Finite field operators are of high relevance for public key cryptography implementations since they are the main components in which algorithms efficiency is sustained. Since PKC algorithms operate over large numbers, underlying finite field operations are very costly. Consequently, finite field operators have been widely studied to improve performance of PKC implementations. Software implementations try to improve clock cycles required in finite field arithmetic or reduce the necessary memory space in which operands, partial results, and lookup tables are stored. Another aim of software implementations is to reduce required memory access to operands and partial results. However, for some applications, software finite field operators are not fast enough to meet time requirements, especially when resource-constrained devices are used, for example, 8-bit processors. In those cases, hardware finite field operators are required.

Traditionally, design and implementation of hardware modules for embedded systems have been carried by Application Specific Integrated Circuits (ASICs), since it provides high performance and/or low power budget that many systems require (at the expenses of long and difficult design cycles) [44].

However, for hardware realization of finite field operator, Field-Programable Gate Arrays FPGAs could be preferred because of flexibility, low cost, fast time to market, and long-term maintenance [45]. Particularly for cryptographic applications, FPGAs

have the advantage of reconfiguration or reprogramming whenever a new security requirement is necessary or when an algorithm must be adapted to support higher security levels [46]. Today FPGAs are not only used as rapid prototyping devices but as final products [47].

Next section presents a summary of the finite field operators required for cryptography applications.

2.3.1 Multiplication, inversion and squaring operators in F_q

Basic operations for public key cryptography are multiplications, inversion and squaring. For cryptosystems based in the discrete logarithm problem over F_p* the basic operation is multiplication over F_p. Besides, elliptic curve cryptography can be accomplished in prime field F_p or in binary field F_{2^m}. For ECC cryptography multiplication, inversion and squaring operations are required, either in prime field $\mathbb{F}_p$ or binary field $\mathbb{F}_{2^m}$ depending on the field on which curves are defined.

Multiplication over $\mathbb{F}_p$

Multiplication in F_q with $q = p$ is of great relevance for cryptography algorithms based in the discrete logarithm problem over F_p such as RSA, ElGamal, Digital Signature Algorithm (DSA), among others. F_p multiplication is computed as $a \times b \bmod p$. Multiplication in F_p is commonly accomplished in one of the following approaches [30]:

- First multiply, then divide.
- Multiplication steps and reduction are interleaved.
- Brickell's method.
- Montgomery method.

Depending on needs implementation and the target device it is possible to select one or another approach. Montgomery method replaces division operation by additions, subtraction, and siftings. However, Montgomery method requires to transform operands to a Montgomery representation; when only a multiplication is required the Montgomery method seems to be more inefficient than other approaches. However, when several multiplications are needed in a continuous way Montgomery method only requires operands transformations at start and end of all multiplications. For hardware realizations, division seems to be a very costly operation. Thus, Montgomery

method is a very attractive approach for multiplication over F_p that serves as an engine for $\mathbb{F}_p^*$ exponentiation.

Addition over $\mathbb{F}_{2^m}$

Addition of $A, B \in \mathbb{F}_{2^m}$ elements is a straightforward operation. Thus, its computational complexity is usually neglected [30]. Addition in $\mathbb{F}_{2^m}$ is commonly accomplished by the xor operation of coefficients of polynomials A and B.

Squaring over $\mathbb{F}_{2^m}$

Squaring an element A in $\mathbb{F}_{2^m}$ is a linear operation since it is only required to insert $0's$ between consecutive elements of A as shown in figure 2.10.

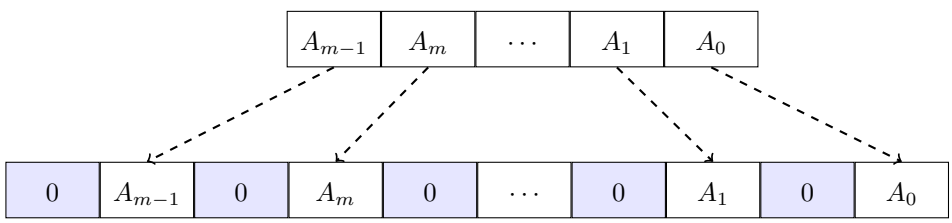

Figure 2.10: Square of the element $A \in \mathbb{F}_{2^m}$.

So, square elements in $\mathbb{F}_{2^m}$ is an easy task. However, it is required to reduce the square polynomial. NIST has proposed finite fields generated by trinomials and pentanomials, and corresponding fast reduction algorithms which are faster than generic reduction algorithms and do not have storage overheads [40].

Multiplication over $\mathbb{F}_{2^m}$

Multiplication over $\mathbb{F}_{2^m}$ is a fundamental operation in elliptic curve cryptography. This operation is widely studied and analyzed since is the core of arithmetic required in ECC.

It is well known that a $\mathbb{F}_{2^m}$ finite field could be defined using an irreducible polynomial: $F(x) = x^m + f_{m-1}x^{m-1} + \cdots + f_2x^2 + f_1x + 1$ where $f_i \in \mathbb{F}_{2^m}$ for $0 \leqslant i \leqslant m$ [48]. If α is a root of $F(x)$ then an element $A \in \mathbb{F}_{2^m}$ could be represented in polynomial basis as $A(\alpha) = \sum_{i=0}^{m-1} a_i\alpha^i$, $a_i \in \mathbb{F}_2$. Thus, multiplication of $A, B \in \mathbb{F}_{2^m}$ elements is accomplished by a polynomial multiplication of A and B elements with coefficients in $\mathbb{F}_2$ following a polynomial reduction by irreducible polynomial $F(x)$.

Traditionally, the hardware implementations of multiplication over $\mathbb{F}_{2^m}$ have been accomplished in one of the following [49]:

- Full-parallel, A and B operands' coefficients are processed in parallel with achieving a very high throughput but at the price of high circuit area consumption.
- Bit-serial, a single bit of the multiplier A is processed at a time, while the coefficients of the multiplicand B are processed in parallel. If the operand A has m bits, multiplication takes m steps to be computed.
- Digit-serial, the main idea is to process d coefficients of A multiplier at each time, while multiplicand B is processed in parallel. If $d = 1$ a Bit-serial approach is achieved. However, the parameter d allows a trade-off between area and performances.

Hardware finite field operators for multiplication over $\mathbb{F}_{2^m}$ have used a variety of techniques to improve throughput or reduce area resources.

Inversion

Among finite field arithmetic operations used in public key cryptography: addition, subtraction, multiplication, and inversion of non zero elements, inversion is the most time-consuming operation [30]. The inverse of a non-zero element $a \in \mathbb{F}_{2^m}$ is defined as the unique element $a^{-1} \in \mathbb{F}_{2^m}$ such that $aa^{-1} = 1$ [30]. In the literature several algorithms have been proposed to compute the multiplicative inverse of an element $a \in \mathbb{F}_{2^m}$, most of them are based on the extended Euclidean algorithm or in the Fermat's Little Theorem (FLT).

Inversion operation is mainly used in ECC operations. However, some algorithms use different point coordinates to avoid inversion operations. In this case, only it is required to compute the inversion of one element at the coordinates transformation.

As conclusions, finite field multiplication is the most used critical operation in both of them, prime field $\mathbb{F}_p$ and binary field $\mathbb{F}_{2^m}$. Therefore, finite fields multiplication algorithms and implementations are widely studied in the research community since DLP public key cryptosystems performance depends directly on this operation.

2.3.2 Exponentiation operator in F_q^*

Exponentiation operation in $\mathbb{F}_q^*$ is the main operation in some public key cryptosystems such as RSA, DH, and ElGamal. As $\mathbb{F}_p^*$ is a subset of the finite field $\mathbb{F}_p$, it is common to refer to exponentiation in $\mathbb{F}_p^*$ or $\mathbb{F}_p$ indistinctly.

Exponentiation in $\mathbb{F}_p$ is defined as the operation $g^e \bmod p$. g is an integer, with $0 \leqslant g < p$, and e is an arbitrary positive integer. The basic method for exponentiation

by multiplying g by itself $e-1$ times is inefficient. So, faster algorithms have been proposed to compute $g^e \bmod p$ [50], for example:

- Binary method: This method scans exponent bits either from left to right or from right to left. At each step, a square operation is computed, and depending on the scanned bit value a subsequent multiplication can be performed.
- m-ary method: The generalized form of the binary method can scan bits of the exponent: two at a time (quaternary method), three at a time (the octal method), etc., more generally, $\log_m$ at a time (the m-ary method). This method is based on m-ary expansion of the exponent. Digits of e are scanned, then squaring and subsequent multiplications are computed accordingly [51].
- Sliding window techniques: In the m ary method the exponent is decomposed in d-bit words. Probability of a d-word being zero is equal to 2^{-d}, assuming that 0 and 1 bits are produced with equal probability. The sliding window algorithms provide a compromise by allowing zero and nonzero words of variable length; this strategy aims to increase the average number of zero words while using relatively large values of d.
- Montgomery Powering Ladder (MPL): This method scan exponent bits, and in each step performs a square and a multiplication. The main advantages of MPL are that it does not have conditional jumps nor extra operations, as in other approaches, which makes it resistant to certain kind of side-channel attacks, such as the Simple Power Analysis (SPA) attack [52].

Modular exponentiation is the critical operation in PKC. However, modular exponentiation is computed by a large number of multiplications. So, modular exponentiation and modular multiplications are the underlying operations for discrete logarithm problem (in multiplicative group $\mathbb{F}_p^*$) based cryptography.

2.3.3 Exponentiation (scalar multiplication) operator in $E(F_q)$

Point multiplication or *scalar multiplication* is the operation to compute kP, where k is an integer and P is a point on an elliptic curve E defined over a field $\mathbb{F}_q$ [40]. This operation dominates the execution time of curved based public key cryptography. There is a wide range of approaches to compute this operation. Furthermore, the study of algorithms and implementations of this operation is a very active research area. Some of the algorithms used to compute scalar multiplications are [40]:

- Binary method: This is the additive version of the binary method for exponentiation. Scalar bits are scanned either from left to right or from right to left, and at each step, a point double is computed and depending on the scanned bit value, a subsequent point addition can be performed. Assuming that 0 and 1 bits of the exponent are produced with equal probability, it is expected that the binary algorithm executes approximately $m/2$ point additions and m point doubles.

- Non-adjacent form (NAF): This method is expected to compute approximately $m/3$ point additions and m point double. To achieve this, a signed digit representation of the exponent k is used, $k = \sum_{i=0}^{l-1} k_i 2^i$ where $k_i \in \{0, \pm 1\}$. NAF is a particularly useful signed digit representation. A NAF of a positive integer k is an expression $k = \sum_{i=0}^{l-1} k_i 2^i$ where $k_i \in \{0, \pm 1\}$, $k_{l-1} \neq 0$ and no two consecutive digits k_i are nonzero. NAF length is l [40].

- Montgomery method: Julio López and Ricardo Dahab [53] proposed an algorithm to compute scalar multiplication based in Montgomery idea [54]. In this method, scalar bits are scanned, and the algorithm performs a point addition and a point double at each step. Compared to the binary method that requires $m/2$ point additions and m point double, Montgomery method required m point additions and m point double at each step. However, operations uniformity makes this algorithm resistant to certain power analysis attacks, since there is no conditions or extra operations.

Since scalar multiplication is the main operation in ECC several approaches have been proposed to speed up this operation. This section presented some of the most representative algorithms that compute the general operation of scalar multiplication. However, an approach for computing scalar multiplication when a fixed point is required have been proposed too, for example, Fixed-base windowing methods and Fixed-base comb methods [40]. However, this work is focussed on scalar multiplication in a general way, and not for a fixed point.

2.4 Summary

A brief mathematics background for public key cryptography has been presented in this chapter. Firstly, central concepts such as groups, finite fields, and the discrete logarithm problem were presented as a basis for the public key cryptography. Secondly, public key cryptosystems such as RSA, ElGamal, and ECDH were described to analyze underlying operations required in each of them. Finally, importance of finite field

operators was presented, as well as different approaches proposed by the research community to improve each of these operations.

The next chapter gives a detailed state of the art study of the most relevant finite field operators studied in this section.

Chapter 3

State of the art

As it was stated in Chapter 1, operations in groups and finite fields are critical for public key cryptography. In Chapter 2, it also was pointed out that critical finite field and group operations for public key cryptography are multiplication in $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$, and exponentiation in multiplicative group $\mathbb{F}_p^*$ and in additive elliptic curve group $\mathbb{E}(\mathbb{F}_{2^m})$. Therefore, this chapter surveys previous works and main techniques used to construct finite field operators for public key cryptography.

Firstly, an introduction to hardware finite field operators is presented. Secondly, hardware finite field multipliers over $\mathbb{F}_p$ are presented, followed by exponentiation over $\mathbb{F}_p^*$. Next, multipliers over $\mathbb{F}_{2^m}$ are surveyed. Finally, related works for hardware implementation in FPGAs for exponentiation over elliptic curves $\mathbb{E}(\mathbb{F}_{2^m})$ are presented.

3.1 Finite field operators in hardware

Finite field operators have been implemented both in hardware and software. Traditionally, pure software implementations of public key cryptography are attractive when flexibility is required, for example, when changes in parameters, key size or other parameters need to change frequently. However, execution time for software implementations is higher than hardware implementations. Some resource-constrained devices bring small processors (4-bit, 8 bit or 16-bit), and public key software implementation in these platforms do not meet time requirements for critical applications. Hardware architectures are required to speed up critical operations (finite field arithmetic) for public key cryptography. Application Specific Integrated Circuits (ASICs) have been the leading platform for hardware implementations of public key cryptography. The main drawback with ASICs is that changes to the hardware are not possible. Field Programmable Gate Array's (FPGAs) offers hardware reconfigurable which can be used to speed up the critical tasks, but with the flexibility that if requirements change,

the hardware can be reconfigured. Since cryptography requirements change depending on the application, FPGAs are commonly used as target technology.

This chapter is focused on hardware techniques implementations of finite field operators in FPGAs. The main design goal for hardware designers is high throughput or reduced area. On the one hand, high throughput designs are necessary for some operations that work with great amount of data, and commonly require a considerable amount of hardware resources. On the other hand, hardware designs optimized for a small amount of hardware are necessary for small resources constrained devices. Although there are different technologies for finite field operators implementations, such as ASICs, this research project is concerned mainly in hardware finite field operators in reconfigurable computing platforms: Field Programmable Gate Array's (FPGAs). therefore, state of the art review is related on this technology. The metric used for size comparison in hardware architectures is slices.

3.2 Hardware architectures for the multiplication operator over $\mathbb{F}_p$

Exponentiation over $\mathbb{F}_p$ with large numbers is commonly accomplished by repeated multiplications over $\mathbb{F}_p$, which is an operation that consumes a considerable amount of time in general purpose processors. Consequently, hardware finite field operators have been studied to speed up multiplication over $\mathbb{F}_p$.

One of the most used algorithm to accomplish multiplication over $\mathbb{F}_p$ is the Montgomery algorithm since it replaces trial division by a series of additions and divisions by a power of two that can be easily implemented in hardware devices [55].

Several techniques have been developed to reduce hardware resources and/or speed up the Montgomery computation. One of them is the use of Carry Save Adders (CSA) in partial operations to reduce the critical path in carry propagation additions [56, 57, 58]. The primary challenge in the Montgomery multiplier with CSA is the number format conversion from CSA to binary representation. Another common technique for implement the Montgomery algorithm is the use of systolic arrays such as in [59, 60, 61]. Systolic arrays help to improve computation time since many Process Elements (PE) works in parallel. However, systolic arrays commonly require a significant amount of hardware resources depending on the number of PE. High radix design (digit-serial) commonly improves the bit-serial implementations since a group of bits are processed at a time at the cost of extra hardware resources. Because of this, small hardware architectures commonly uses bit-serial implementation

or high radix with small digit size (i.e. 2 and 4 bits digit sizes). Another technique used to reduce hardware resources and speed-up partial multiplications is the Karatsuba algorithm [62, 63].

In [64] several algorithms are presented to implement the Montgomery multiplication, mainly in software. Algorithms are proposed according to whether multiplication and reduction steps are separated or integrated, and how operands are processed, in operand scanning one loop moves through it one word at a time, while in the product scanning the loop moves through partial product results. In Separated Operand Scanning (SOS) product and reduction steps are separated, so in multiplication steps, size of partial result is double of the multiplicands. In Coarsely Integrated Operand Scanning (CIOS), multiplication and reduction steps are interleaved, so in each step, partial results are reduced and extra storage resources are not required. Other algorithms were proposed in [64] i.e. Finely Integrated Operand Scanning (FIOS), Finely Integrated Product Scanning (FIPS) and Coarsely Integrated Hybrid Scanning (CIHS). However, CIOS version of Montgomery algorithm operates faster than these other algorithms.

CIOS version of the Montgomery algorithm [64] was implemented for a low-cost FPGA Spartan 3 of Xilinx in [59] in a systolic array version. Operands are processed in words of radix R which determines the bit length of partial multipliers. The number of PE can be adjusted to meet area vs performance trade-off.

When bit-serial or digit-serial approaches of the Montgomery algorithm are implemented long operands are used in the intermediate results, addition delay of these operands is a crucial aspect since carry propagation requires a considerable amount of time when operands get bigger. Carry-save representation of operands and CSA have been used to reduce the critical path in the Montgomery algorithm [58, 57].

In [57] it is proposed a Montgomery multiplier that uses CSA in order to avoid carry propagation, and carry-skip addition is used to reduce time conversion of the carry save representation to the binary representation. Walter proposes to avoid final subtraction in the Montgomery algorithm at the cost of one extra iteration in the main loop [4], see Algorithm 3. However, since addition is very efficient with CSA and an efficient carry-skip circuit is used to convert partial operands, in [57] it is used the original Montgomery algorithm. This $\mathbb{F}_p$ multiplier is developed in a digit-serial and bit-serial approach, presenting results for digit size of 1 (bit-serial), 2, 4, and 8 bits, and operand sizes of 512, 1024, and 2048 bits. Furthermore, the digit-serial Montgomery multiplier from [57] is used to construct an exponentiation $\mathbb{F}_p$ module (discussed in the next section).

In [58], CSA architecture for implementing the Montgomery algorithm is used. The carry save adders are used in the main loop of Algorithm 3 to avoid large carry propagation of partial results during the stage of the computation. The main characteristic of the Montgomery multiplier presented in [58] is that the same datapath that computes the partial result of the main iteration of the Montgomery algorithm is used to convert the final results from CSA representation to binary representation. Different to other works that require additional logic to realize the conversion [65], the Montgomery multiplier architecture presented in [58] requires a small number of hardware resources.

In [58, 57], CSA is used to improve the critical path in the Montgomery multiplication algorithm. It is well suited for bit-serial architectures since the critical path is improve at the cost of a small amount of logic for CSA. High-radix (digit-serial) implementations can be used to improve latency at the cost of area results as in [57]. In [66], it is proposed the use of a canonical operands representation, where an integer is expressed in canonical form as $X_{CR} = (x_{n-1}, x_{n-2}, \cdots, x_0)$ where $x_i \in \{-1, 0, 1\}$. The main advantage of the canonical form is that the Hamming weight of an n bit integer is $n/3$. To take advantage of the canonical representation, in [66] it is proposed an expansion of an n-bit integer as $X_{SD} = (z_{n-1,z_{n-2},\cdots,z_0})$. Each z_i includes a number of consecutive zero bits that can be followed by a nonzero digit (1 or -1). With this expansion, each partial multiplication can compute a digit (z_i) that can be all zeros or one followed by zeros, than can be easily computed with only one CSA and shift operations. The main drawback with the proposed expansion in [66] is the clock cycles and hardware resources required to realize operands conversions, first to canonical representation, and next to the proposed expansion. Furthermore, digits size is variable, so for different operands the Montgomery algorithm has different execution time.

In [63] the Karatsuba algorithm for partial multiplications in the Montgomery algorithm is used. A 256 bits Montgomery multiplier is presented. The Karatsuba algorithm is used in a recursive way down to partition operands to 32 bits where four embedded multipliers (DSP) units are used to accomplish partial results. Although a high throughput is achieved with a small number of slices, the Montgomery multiplier presented in [63] require a significant amount of Digital Signal Processing (DSP), 108 (not at disposal for some low-cost FPGAs).

Another used technique for large integer multiplication is the use of the Karatsuba-Offman algorithm. This algorithm reduces the overall running time of the multiplication of two N-digits integer to $O(N^{log_2 3})$, as compared to $O(N^2)$ in the traditional multiplication algorithm. In [62] Karatsuba-Offman algorithm is used to speed up

Table 3.1: Approaches taken in hardware finite field operator $\mathbb{F}_p$ multiplier.

Ref.	size	Target	Algorithm	Approach	Slices	Time
[59]	1020	Spartan 3	Montgomery (CIOS)	Systolic Array	1553	7.62 µs
[57] (d=2)	1024	Virtex 5	Montgomery	Digit-serial (CSA)	12323 (LUTs)	1339 ns
[58]	1024	Virtex 2	Montgomery (CSA)	New CSA Conversion	4512	9021 ns
[66]	1024	Virtex 5	Montgomery	High Radix with CSA	6105	0.883 µs
[63]	256	Virtex 6	Montgomery	Karatsuba	19362 (LUTs) (108 DSPs)	328 ns
[62]	1024	Virtez 7	Montgomery	Karatsuba	235	34.45 µs
[67]	256	Virtex 6	Interleave multiplication	bit-serial	3900 (LUTs)	1.3 µs

the Montgomery algorithm. The advantage of the Montgomery multiplier presented in [62] is operands processing in a digit-digit fashion, but since the Karatsuba-Offman algorithm is recursive, in [62] a digit size of 16 bits is used to take advantage of DSP units in the FPGA, and a significant amount of DSP units are required to implement the Karatsuba-Offman multipliers.

Other recent works for hardware finite field $\mathbb{F}_p$ multipliers on FPGAs have proposed the use of the classical interleaved multiplication technique in bit-serial or digit serial approach. [67] presents the bit-serial and digit-serial $\mathbb{F}_p$ multiplier, $a \times b \bmod p$, using the interleaved multiplication algorithm where each bit in b is tested at a time, and the main loop only requires multiplication by two and additions modulo p. The digit-serial $\mathbb{F}_p$ multiplier presented in [67] occupies 26% more FPGA LookUp Table (LUT)s than the bit-serial implementation. Since many if conditions are used in the main loop of the algorithm, the execution time is not constant for all operands.

Table 3.1 summarizes state of the art works for hardware finite field multiplication over $\mathbb{F}_p$. The most compact design in state of the art is the work presented in [62] which uses Montgomery and Karatsuba algorithms. Furthermore, that design processes operands digit by digit in a pipeline fashion. Montgomery implementations with Carry Save Addition is commonly used to speed up operation frequency since it is not required a carry propagation for additions. However, CSA require extra hardware logic for CSA modules. Moreover, High-Radix CSA help to reduce latency since many bits are processed at a time, but they require a considerable amount of hardware resources, as in [57] and [66].

3.3 Hardware architectures for exponentiation operator over $\mathbb{F}_p^*$

Exponentiation in $\mathbb{F}_p^*$ is commonly accomplished by repeating multiplications over $\mathbb{F}_p$ which is a considerable time-consuming operation. Exponentiation over $\mathbb{F}_p^*$ directly depends on the efficiency of multiplication over $\mathbb{F}_p$ and how many multiplications are required. For a detailed description of exponentiation over $\mathbb{F}_p^*$ algorithms, the reader could refer to [68]. The most used algorithms for exponentiation over $\mathbb{F}_p^*$ are the binary method (Least Significant Bit (LSB) and Most Significant Bit (MSB) also known as Right to Left (R2L) and Left to Right (L2R)) and the Montgomery Powering Ladder (MPL). The MPL algorithm is very attractive for hardware implementations for $\mathbb{F}_p^*$ exponentiation since it is a constant time algorithm resistant to certain side-channel attacks such as Simple Power Analysis (SPA). Furthermore, in each iteration of the main loop of Algorithm 6, there are not data dependency which allows two multipliers to execute in parallel. Table 3.2 resumes most relevant works of $\mathbb{F}_p$ hardware exponentiator architectures.

In [59] it is presented a hardware finite field operator for exponentiation over $\mathbb{F}_p$ that use the MPL algorithm. There are no details of its implementation but it uses the Montgomery CIOS algorithm to perform partial multiplications implemented in a systolic array where the number of PEs can be configured according to a trade-off area/speed. Block RAMs are used to store input values, partial multiplication results, and final exponentiation result.

In [66], $\mathbb{F}_p$ exponentiation algorithms R2L and L2R are used. Since [66] uses an integer expansion of operands, it was required to modify algorithms R2L and L2R, mainly for conversion of the number format representations with pre-computation and post-computation before and after the main loop. The R2L algorithm is implemented with two partial multipliers while the L2R only requires one, saving hardware resources at the cost of lower latency.

Sutter et al. [57] present the implementation of R2L and L2R exponentiation algorithms. The partial multiplier is a Montgomery CSA version in bit serial and digit-serial versions. In [57] at each iteration of R2L and L2R algorithms, operands are converted from CSA representation to binary representation with a carry-skip addition. R2L implementation require two partial Montgomery CSA multipliers which can compute products in parallel.

In [66] a hardware finite field exponentiator over $\mathbb{F}_p$ is deployed which makes use of L2R and R2L exponentiator algorithms. These algorithms were modified to

Table 3.2: Hardware approaches for finite field exponentiation over $\mathbb{F}_p$.

Ref.	size	Target	Algorithm	Approach	Slices	Time
[59]	1020	Spartan 3	MPL	Systolic Array	3899	7.95 ms
[57] (d=1)	1024	Virtex 5	R2L	Bit-serial (CSA)	8242 FF/ 11330 LUTs	2.98 ms
[57] (d=2)	1024	Virtex 5	R2L	Digit-serial (CSA)	8243 FF/ 15427 LUTs	2.03 ms
[57] (d=2)	1024	Virtex 5	L2R	Digit-serial (CSA)	13387 FF/ 27750 LUTs	2.03 ms
[66]	1024	Virtex 5	R2L	High Radix with CSA	6776	1.37 ms
[66]	1024	Virtex 5	L2R	High Radix with CSA	12716	0.92 ms
[62]	1024	Virtez 7	R2L	Karatsuba	3046	0.54 ms
[69]	1024	Virtez 5	MPL	Common-multiplicand	3218	3.18 ms

make use of the Montgomery multiplier with signed-digit representation and integer expansion proposed in [66]. The main advantage of this architecture is using integer expansion with variable digit sizes in which zeros follow a one, or digits with many zero values are processed.

A hardware finite field operator for exponentiation over $\mathbb{F}_p$ is presented in [62] which makes use of R2L binary exponentiation algorithm to exploit parallelism between partial multiplications, since there is no data dependency unlike with L2R binary exponentiation algorithm. The main characteristic of $\mathbb{F}_p$ exponentiator presented in [62] is the use of the Montgomery algorithm for partial multiplications which in turn makes use of the Karatsuba algorithm for partial multiplications to reduce hardware resources.

In [69], it is presented a hardware finite field exponentiator over $\mathbb{F}_p$ that computes partial multiplications by common-multiplicand Montgomery algorithm. The common-multiplicand Montgomery algorithm was introduced in [70] for software implementations. This algorithm is based in the observation that in the binary exponentiation algorithm (same for MPL) there are required a square and a multiplication that share one operand (for example $A \times B$ and $A \times A$). The basic idea is that the common part of the partial Montgomery multiplications ($A \times B$ and $A \times A$) can be computed once rather than twice. In [69] the algorithm was modified to take advantage of hardware implementations for FPGAs.

In this section the most relevant state of the art works for hardware finite field exponentiation over $\mathbb{F}_p$ were presented. The binary exponentiation algorithm (L2R and R2L) and the Montgomery Powering Ladder (MPL) are the milestone algorithms for finite field exponentiation. Hardware designs in the state of the art only modify these algorithms (L2R, R2L, MPL) to adopt respective partial multipliers and to make

some pre and post computations like the input operands representation in CSA form or integer expansion [66].

3.4 Hardware architectures for the multiplication operator over $\mathbb{F}_{2^m}$

A finite field of characteristic two is denoted as $\mathbb{F}_{2^m}$. Elements of $\mathbb{F}_{2^m}$ can be represented using a polynomial basis [71] (i.e. $\sum_{i=0}^{m-1} a_i x^i, a_i \in \{0, 1\}$). In this representation a $\mathbb{F}_{2^m}$ element can be expressed by m binary bits which is advantageous for hardware implementations. Multiplication over $\mathbb{F}_{2^m}$ is one of the main operations used for public key cryptography.

There are three main families of algorithms to compute $A(x)B(x) \bmod F(x)$ (being $F(x)$ a degree-m monic irreducible polynomial over $\mathbb{F}_p$, where p is a prime): full parallel, bit-serial, and digit-serial [72]. The Full parallel approach requires a considerable amount of hardware resources. Bit-serial approach has been the most compact implementation. Digit-serial approach allows a trade-off between computation time and circuit area. The most widely used algorithms to perform multiplication over $\mathbb{F}_{2^m}$ are the Most Significant Element first (MSE) and the Least Significant Element first (LSE). Both of them have been used in bit-serial or digit-serial approach.

In multiplication over $\mathbb{F}_{2^m}$ partial results need to be reduced. Bertoni et al. [73] presented an easy way to perform modulo reduction when partial results have coefficients with powers greeter than $m-1$ (e.g. a^m). They presented algorithms to compute $\mathbb{F}(q)$ multiplications. Furthermore, practical implementations for $\mathbb{F}(3^m)$ and $\mathbb{F}(2^m)$ are presented.

Beuchat et al. [72] surveyed some of the most representative $\mathbb{F}_{2^m}$ implementations using MSE and LSE algorithms (including implementations presented in [73]). For MSE algorithm bit-serial and digit-serial algorithms are presented. Digit-serial implementations require $\lceil m/D \rceil$ iterations using $m-1$-degree partial results [74]. However, in [75] it is proposed $m+D-1$-degree partial results to improve computation performance at the cost of one extra iteration, requiring $m+1$ iterations to compute multiplication over $\mathbb{F}_{2^m}$. For LSE algorithm, bit-serial and digit-serial algorithms are presented. Digit-serial algorithm proposed in [73] require $m+1$ iterations and keeps $m+D-1$-degree partial results to improve computation performance. Beuchat [72] conclude that MSE first approach requires less hardware and offers higher throughput than LSE first. Furthermore, MSE algorithms are almost always more efficient than LSE first algorithms.

Table 3.3: Hardware approaches for multiplication over $\mathbb{F}_{2^m}$.

Ref.	Field	Target	Algorithm	Approach	Slices	Time (ns)
[72]	$\mathbb{F}(2^{233})$	Spartan 3	MSE	Digit-serial	3458	58.0
[72]	$\mathbb{F}(2^{233})$	Spartan 3	LSE	Digit-serial	3504	62.0
[72]	$\mathbb{F}(2^{409})$	Spartan 3	MSE	Digit-serial	5406	153.0
[76]	$\mathbb{F}(2^{233})$	Virtex 6	Schoolbook method	Digit-serial	1643 (LUTs)	802.4
[71](d=1)	$\mathbb{F}(2^{233})$	Virtex 5	LSE	Digit-serial	714 (LUTs)	415.0
[71](d=16)	$\mathbb{F}(2^{233})$	Virtex 5	LSE	Digit-serial	2351 (LUTs)	35.0
[79]	$\mathbb{F}(2^{233})$	Spartan 3	MSE	Digit-digit	406	219.0
[80]	$\mathbb{F}(2^{163})$	Virtex II	M-I algorithm	Systolic Array	1399	

In [76] is presented a multiplier over $\mathbb{F}_{2^m}$. As main characteristic multiplication and reduction steps are performed separately. It is stated that for a finite field generated by irreducible polynomials $F(x)$ defined in standards (NIST [22]), reduction can be performed by a set of *xor* operations [40, 77]. In [76] is considered only the multiplication step, implemented in a digit-serial approach. A digit-size of 16 is proposed since in most cases 16-bit words give better results. However, the approach of multiplication and next reduction gives a partial result of doubling the operands size. For example, the operands used in [76] are of 233-bits, and the partial multiplication gives a result of 264-bits, which requires standard logic to be stored.

In [71] a elliptic curve point multiplication architecture over $\mathbb{F}_{2^m}$ is presented. Since multiplication over $\mathbb{F}_{2^m}$ is the underlying operation of elliptic curves, in [71] is used a LSE digit-serial multiplier [78] to improve the performance of multiplication over $\mathbb{F}_{2^m}$. However, a digit size of one bit (bit-serial) is the most compact version of the multiplier presented in [71].

In [79] is presented a digit-digit multiplier over $\mathbb{F}_{2^m}$ based in MSE algorithm. Operands, modulus and partial results are partitioned in digits and processed one digit at a time. The main advantage compared to digit-serial or bit-serial implementations is that operands and partial results can be stored in Block RAM (BRAM)s instead of in shift registers which saves standard logic (slices).

Another approach to compute the multiplication over $\mathbb{F}_{2^m}$ is to perform multiplication and inversion using the architectural structure of an inversion architecture. Multiplication can be computed without an increase in hardware resources. In [80] it is proposed a systolic hardware architecture to compute multiplication/inversion in the same hardware. Furthermore, an arithmetic unit is constructed that can perform all $\mathbb{F}_{2^m}$ arithmetic operations required in elliptic curve cryptography.

Table 3.3 summarize the most relevant works for multiplication over $\mathbb{F}_{2^m}$, the main used algorithms, and area/time results.

3.5 Hardware architectures for the scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$

The most time consuming operation in Elliptic Curve Cryptography is the exponentiation over $\mathbb{E}(\mathbb{F}_{2^m})$ also known as scalar multiplication or point multiplication. Efficiency of exponentiation over $\mathbb{E}(\mathbb{F}_{2^m})$ directly depends on the selected parameters: finite field, curve, coordinate system, multiplication algorithm over the field, inversion algorithm over the field, and exponentiation algorithm. This section presents the state of the art works most relevant for ECC arithmetic.

In [1] is proposed a lightweight hardware architecture for elliptic curve point multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$. The architecture is composed by three main modules: control unit, arithmetic unit, and memory bank unit. In turn the arithmetic unit consists of components: multiplication block, squared block, and addition block. Addition and square over $\mathbb{F}_{2^m}$ operations are computed full parallel, totally combinational. However, multiplication over $\mathbb{F}_{2^m}$ is computed via the bit serial approach using MSE algorithm. The bit-serial approach is commonly used for low area requirements. The memory bank consists of six registers, three of them are shift registers required in the multiplication block, since one bit of one operand is required at a time whilst the other is required full parallel. The control block is in charge of starting one of the arithmetic modules, and enable/disable shift registers and write registers. The architecture presented in [1] uses affine coordinates, and the Itoh-Tsuji algorithm to compute field inversion over $\mathbb{F}_{2^m}$. The lightweight hardware architecture presented in [1] reports results in ASICs platforms. This architecture was reimplemented for FPGAs in this thesis research for comparisons. Figure 3.1 shows the lightweight hardware architecture for scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$.

In [81] is presented an area-efficient hardware architecture for scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$ in FPGAs. These architecture consists of blocks: addition, multiplication and square block for field $\mathbb{F}_{2^m}$ arithmetic, memory bank, and control module. The memory bank consists of eight shift registers which can be required in full parallel or one bit at a time. The multiplication block is implemented with the digit-serial approach using the MSE algorithm, and requires several pipeline stages and so temporary registers incrementing hardware resources. Projective coordinates are used to avoid the inversion operation in the main loop of the Montgomery algorithm for scalar multiplication. The required operations in the Montgomery algorithm were carefully schedule to take advantage of the arithmetic modules. The inversion operation over $\mathbb{F}_{2^m}$ is computed using the Itoh-Tsuji algorithm.

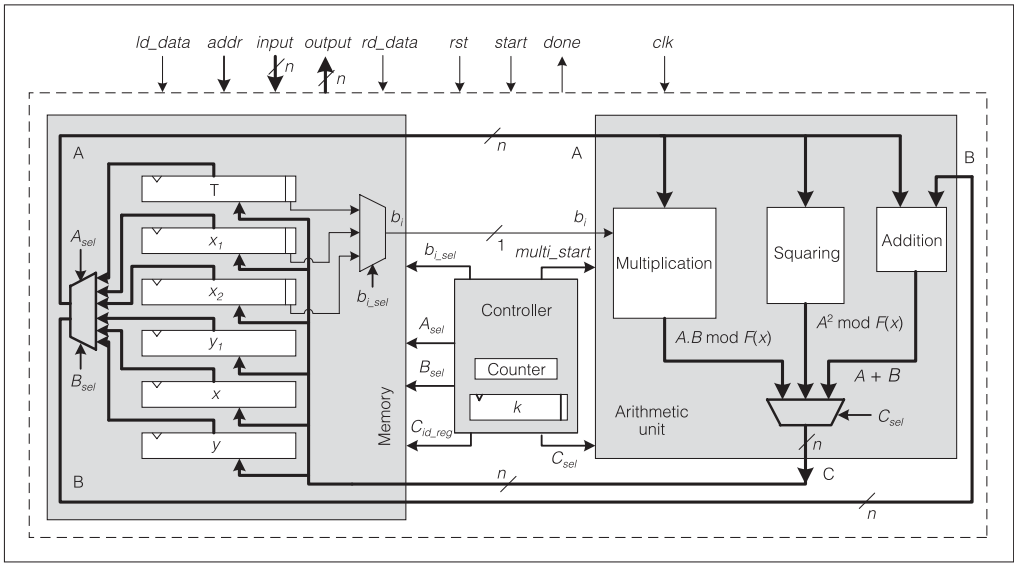

Figure 3.1: Lightweight hardware architecture for scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$ presented in [1].

In [82] is proposed a hardware architecture for elliptic curve point multiplication over $\mathbb{F}_{2^m}$ for resource constrained devices. Same as the previous state of the art works, the main blocks are: adder, squarer, and inversion modules for arithmetic over $\mathbb{F}_{2^m}$, control unit and register unit. However, the control unit is subdivided in: projective to affine controller, doubling controller, addition controller and a finite state machine.

Most of the state of the art hardware architectures implement blocks for: addition, square, multiplication, control unit, and memory bank. However, in [2] is proposed to use a hardware-software co-design where only arithmetic blocks are implemented in hardware modules: addition, square and multiplication. The control unit that is in charge to start arithmetic modules and control the scalar multiplication algorithm is implemented in a PicoBlaze processor which is in charge of the storage of elements too. Figure 3.2 shows the hardware-software co-design proposed in [2].

In [71] is presented a hardware architecture for scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$. Projective coordinates are used to avoid trial division field. The Montgomery ladder point multiplication algorithm is used. However, partial field operations (addition, square, multiplication, inversion) were schedule to take advantage of three partial multipliers and one inversion module. Multiplication over $\mathbb{F}_{2^m}$ is achieved by the digit-serial MSE algorithm. Division (inversions) is computed in digit-serial approach too. Two inversions are required in the Montgomery ladder algorithm, at the end of the main loop, however, the inversion module compute one inversion whiles the main

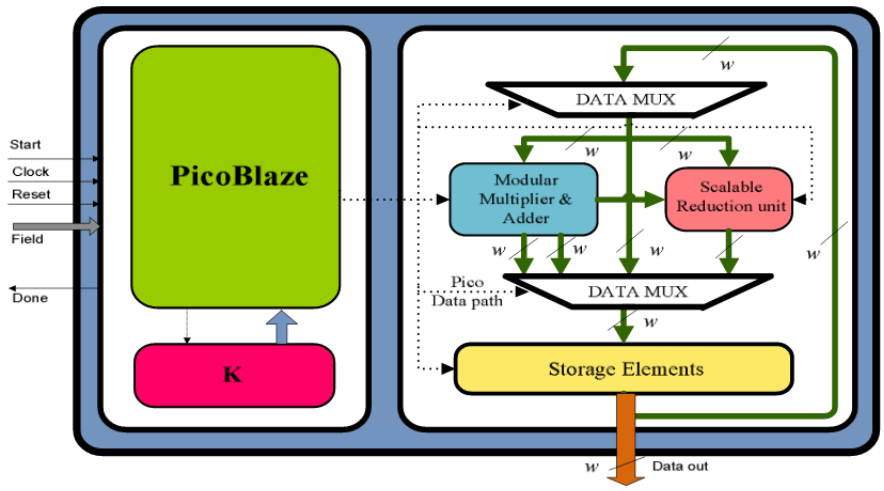

Figure 3.2: Hardware-software co-design for scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$ proposed in [2].

loop is executed, and at the end of the loop only one inversion is required.

In [83] is presented a hardware architecture to compute scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$. These hardware architecture uses affine coordinates together with the left-to-right binary method for scalar multiplication. In each iteration it is required a point double and point addition, the architecture implements two modules, one for point double and other for point addition. Each module is composed of the arithmetic sub modules: addition, inversion, multiplication and squarer. So, each point operation can be computed in parallel. The field multiplication over $\mathbb{F}_{2^m}$ is achieved by the MSE algorithm. Addition is accomplished by xor operation. The field inversion over $\mathbb{F}_{2^m}$ is computed by a modified version of the Euclidean algorithm [84]. A control unit is in charge of orchestrating the field operations required for the binary method for scalar multiplication.

In [85] is presented a hardware architecture for scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$. The main characteristic in this hardware architecture is the use of a bit-parallel Karatsuba multiplier for the field multiplication over $\mathbb{F}_{2^m}$. The Itoh-Tsuji algorithm is used for field inversion. For scalar multiplication is used the left-to-right binary algorithm with a signed representation of the scalar. Square and addition were implemented combinationally. Since bit-parallel Karatsuba multipliers require a considerable amount of time to compute the operation, a three and four pipeline stage were analyzed to reduce the critical path. Four stage pipeline strategy was the best according to area/speed.

Most of the state of the art works presented in this section use the bit-serial or digit-serial approach to implement hardware finite field operators. However, hardware resources required in these approaches depend directly on the operands size (around

Table 3.4: Hardware approaches for exponentiation over $\mathbb{E}(\mathbb{F}_{2^m})$.

Ref.	Field	Target	Mult. Algorithm	Approach	Slices	Time
[1]	$\mathbb{F}(2^{193})$	ASIC	MSE	Digit-serial	17723 GE	41.70 ms
[71]	$\mathbb{F}(2^{233})$	Virtex 5	MSE	Digit-serial	6487	19.89 µs
[81]	$\mathbb{F}(2^{233})$	Virtex 7	MSE	Digit-serial	2647	16.01 µs
[82]	$\mathbb{F}(2^{163})$	Spartan 3	LSE	Bit-serial	3383	2.23 ms
[2]	$\mathbb{F}(2^{193})$	Spartan 3	Comba wxw	Digit-serial	473	125.00 ms
[83]	$\mathbb{F}(2^{233})$	Kintex 7	MSE	Bit-serial	3016	2.66 ms
[85]	$\mathbb{F}(2^{163})$	Virtex 5	Karatsuba	Bit-parallel	3789	10.00 µs

233-2048 bits). The bit-serial approach requires small amount of hardware resources compared to the digit-serial or full-parallel approach, but for large operands even bit-serial state of the art hardware implementations require a considerable amount of hardware resources (slices). Some recent works are proposing to use a digit-digit approach, for example [62] and [3]. The main drawback with the Montgomery multiplier presented in [3] is the use of shift registers to store partial results, and for [62] is to fit the digit sizes to FPGAs embedded DSP multipliers. Another technique to reduce hardware resources in digital design is the use of microprogramming where control signals are stored in memory (commonly computed in software) avoiding hardware resources for a complex control unit. So, this research follows the digit-digit computation approach and makes use of multipliers and memory blocks embedded in most of the FPGAs. Since memory blocks are bigger than operands, it is proposed to used part of memory blocks to store control signal avoiding logic to compute them. The Karatsuba method is a promising algorithm to compute partial multiplications, similar to [62], but in this research Karatsuba algorithm is proposed for compute the $\mathbb{F}_{2^m}$ partial multiplications.

3.6 Summary

This chapter presented the most relevant state of the art approaches to compute finite field operations for public key cryptography: multiplication over $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$, and exponentation over $\mathbb{F}_p$ and elliptic curves over $\mathbb{F}_{2^m}$. The main approach for multiplication over finite fields $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$ are the bit-serial and digit-serial. The bit-serial approach is commonly used when low area is required, and most of the state of the art work with this requirement use it. The next chapter presents the proposed finite field operators with a small area foot print, that it is expected to be used in resource constrained devices.

Chapter 4

Proposed lightweight finite field operators

This chapter presents the novel finite field operators proposed in this thesis. According to the literature review, the critical arithmetic operations are multiplication in the finite fields $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$, and exponentiation in the multiplicative group $\mathbb{F}_p^*$ and in the additive elliptic curve group $\mathbb{E}(\mathbb{F}_{2^m})$. All the finite field operators and exponentiation designs described in this chapter are designed having low area as main design goal. All designs and its validations are done using FPGA as underlying technology.

It is proposed to use a digit-digit computing approach commonly used in software implementation [86], but taking advantages of hardware platforms. With the novel finite field operators (multiplication over $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$, and exponentiation over $\mathbb{F}_p^*$ and $\mathbb{E}(\mathbb{F}_{2^m})$) it is expected to enable some PKC schemes (i.e, RSA and ECC) in resource-constrained devices.

To begin with, the digit-digit approach for operand scanning and processing is presented. After that, it is presented the proposed finite field operators for multiplication over $\mathbb{F}_p$ followed by exponentiator over $\mathbb{F}_p^*$. Then, it is presented the finite field operators for multiplication over $\mathbb{F}_{2^m}$. Finally, details are given about the finite field operator for scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$.

4.1 Digit-digit computation approach

Given $A, B \in \mathbb{F}_q$, each one of n bits in length, it is possible to represent them as:

$$A = a_{n-1}, a_{n-2}, a_{n-3}, \cdots, a_0. \qquad a_i \in \{0, 1\} \tag{4.1}$$

$$B = b_{n-1}, b_{n-2}, b_{n-3}, \cdots, b_0. \qquad b_i \in \{0, 1\} \tag{4.2}$$

However, it is possible to represents elements of a finite field as digits (set of bits)

of size k. For example, A can be represented by d digits of k bits, $d = \lceil n/k \rceil$:

$$A = A_{d-1}, A_{d-2}, A_{d-3}, \cdots, A_0. \tag{4.3}$$

Either in the prime field or the binary field design and implementation of hardware architectures for multiplication in finite fields has commonly followed one of the next approaches:

- Full parallel: Both operands are processed fully in one step, or if pipelined in a few steps. This kind of approach is the fastest of the designs. However, since the operands are numbers with 163-4096 bits in size this kind of architecture requires a considerable amount of hardware resources. Possibly, this hardware architecture will not be able to fit in low-cost FPGAs. Figure 4.1 shows a graphical view of the computation of a finite field multiplication in full parallel hardware architecture.

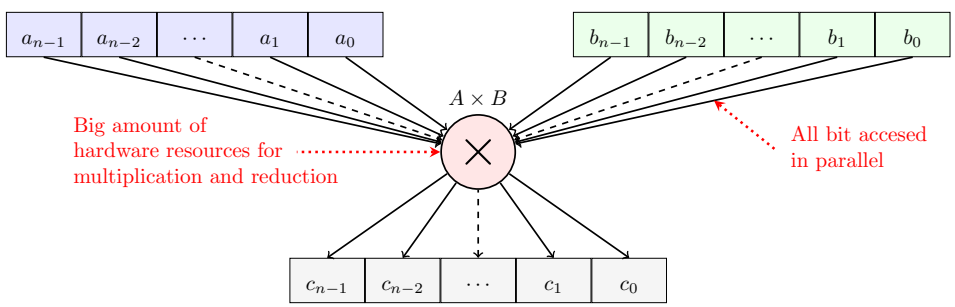

Figure 4.1: Graphical view of a full parallel multiplier.

- Digit-serial: Finite field multiplier can be implemented in different ways according to the available resources. Digit-serial multipliers [87] allow implementations with speed, area, and power consumption trade-offs. In this approach, several coefficients of B are processed at the same time, while all the coefficients of the operand A are processed in parallel. The digit size k defines the number of coefficients of B that are processed in parallel. The use of small digits results in hardware architectures with a small footprint in the area, but big digit size result in faster hardware architectures at the cost of most hardware resources. Figure 4.2 shows a graphical view of the computation of a finite field multiplication in a digit-serial approach.

- Bit-serial: Bit serial is a particular configuration of the digit-serial where the digit size is one bit.

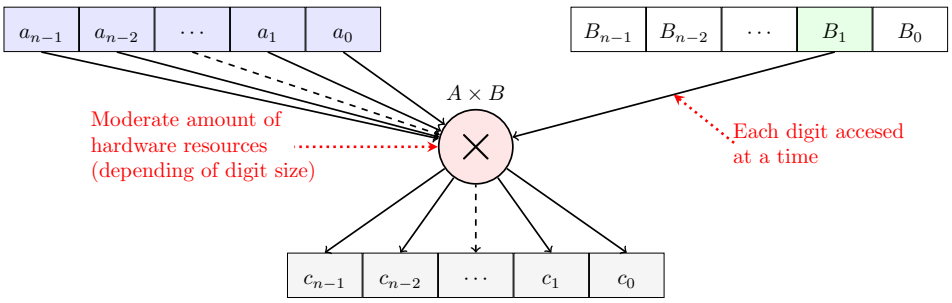

Figure 4.2: Graphical view of a digit-serial multiplier. One digit of B is processed at a time. In this step the digit B_1 is processed.

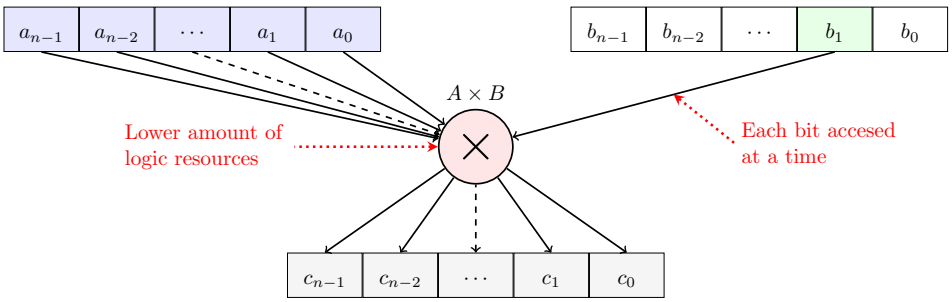

Figure 4.3: Graphical view of a bit-serial multiplier. One bit of B is processed at a time. In this step the bit b_1 is processed.

In this work, digit-digit computation approach is studied. Similar to digit-serial, digit-digit approach allows exploring designs with area/performance trade offs.

In the digit-digit approach, all operands (multiplier, multiplicand, and modulus) and partial results are partitioned into digits and processed iteratively. Furthermore, the final result is delivered in a digit fashion too, which allows to use it as input in a new operation. By varying digit size, it is possible to find configurations that meet requirements of resources, throughput or efficiency. The digit-digit approach is commonly used to implement algorithms for general processors (software) since general processors usually work in words of 8, 16, 32 or 64 bits. However, digit-digit hardware implementations can take advantage of some hardware design techniques such as pipeline or processing operands in a parallel way.

In the next section new compact finite field operators over $\mathbb{F}_p$, $\mathbb{F}_{2^m}$, $\mathbb{F}_p^*$ and $\mathbb{E}(\mathbb{F}_{2^m})$ are presented. The primary goal for the compact finite field operators is to reduce the required hardware resources instead of achieving high throughput computations. However, it is expected that in spite of the low area resources in the hardware design the throughput will be higher than achieved with software implementations in general purpose processors.

As main distinctions, compact hardware architectures proposed in this research:

- Have low area as main design goal instead of high performance,
- Implement a datapath based on the digit size instead of the operands size,
- The same datapath could be used to compute modular exponentiation for different operand sizes (scalability) since datapath is based on digit size, not in operand size,
- Efficiency (area-time) is not lost even using fewer area resources.

4.2 Compact digit-digit Montgomery multiplication in $\mathbb{F}_p$

This section presents design and implementation of hardware architectures for multiplication over $\mathbb{F}_p$. These architectures aim at reducing hardware resources compared to designs presented in state of the art works.

In this work, the Montgomery multiplication algorithm is used to compute multiplication over $\mathbb{F}_p$ since it avoids division operations by computing some extra additions, subtractions and shifts operations. Division operation is a costly operation for hardware implementations that require a large amount of hardware resources and time

of computation. So, the Montgomery method is an alternative that is friendly for hardware implementations of multiplication over $\mathbb{F}_p$.

Montgomery multiplication

The Montgomery Multiplication Algorithm (MMA) [55] listed in Algorithm 1 has been used as a foundation for several implementations of modular multiplication. Given two numbers $A, B \in \mathbb{F}_p$, they are first transformed to the Montgomery domain by doing $A' = A \times R \mod p$ and $B' = B \times R \mod p$. A' and B' are called Montgomery numbers. MMA uses A', B' together with a number R such that $\gcd(p, R) = 1$. Here, p is an N-bit integer number with $2^{N-1} \leqslant p < 2^N$. It is common to use $R = 2^N$. Based on this fact, it is possible to compute numbers R^{-1} and p' using the identity $R \times R^{-1} + p \times p' = 1$, with $0 < R^{-1} < p$ and $0 < p' < R$, using methods such as the extended Euclidean Algorithm. The Montgomery product is defined as $A' \times B' \times R^{-1} \mod p$.

Algorithm 1 Montgomery multiplication algorithm (**MMA**)

Require: Integers A', B', $R = 2^N$, and p a N-bit prime number.
Ensure: $A' \times B' \times R^{-1} \mod p$

```
1: t ← A' × B'
2: q ← (t mod R) × p' mod R
3: u ← (t + qp)/R
4: if u ⩾ p then
5:     u ← u − p
6: end if
7: return u;
```

The transformation of A to A' and viceversa can be done using the MMA algorithm, since $A' = \text{MMA}(A, R^2)$, and $A = \text{MMA}(A', 1)$. Thus, one modular multiplication $A \times B \mod p$ in $\mathbb{F}_p$ requires to compute the next four MMA multiplications:

$$\begin{aligned} A' &= \text{MMA}(A, R^2) \\ B' &= \text{MMA}(B, R^2) \\ Z' &= \text{MMA}(A', B') \\ C &= \text{MMA}(Z', 1). \end{aligned}$$

Additional operations for number conversion, together with the additional computation of p', make the Montgomery method inefficient for computing a single

multiplication in $\mathbb{F}_p$ if compared with traditional multiplication algorithms.

However, the Montgomery algorithm is significantly faster when many consecutive multiplications are required, such as in a $\mathbb{F}_p$ exponentiation. In this case, domain conversion is needed only at the beginning and at the end of the cumulative multiplications. Exponentiation over $\mathbb{F}_p$ is the primary operation for cryptographic algorithms based on the discrete logarithm problem over $\mathbb{F}_p^*$, so, the Montgomery multiplication algorithm seems to be adequate for this type of applications.

Proposed digit-digit $\mathbb{F}_p$ multiplication algorithm

To a better understanding of the algorithms and architectures presented in this section, notation used from here on is shown in Table 4.1.

Table 4.1: Notation.

Symbol	Description
N	Operand size in bits = $\log_2 P$
n	Total k-bit digits of operands = $\lceil n/k \rceil$
p	The modulus defining $\mathbb{F}_p$
X, Y, A	Elements in $\mathbb{F}_p$
β	Radix $\beta = 2^k$
p'	Precomputed value, $p' = -p^{-1} \mod \beta$
Z_i	The ith digit of element $Z \in \mathbb{F}_p$
$X^{<i>}$	Value of X at iteration i

Let X, Y be numbers in $\mathbb{F}_p$. Using radix $\beta = 2^k$, digit-based representation of X, Y is defined as in Eq. 4.4.

$$X = \sum_{i=0}^{n-1} X_i \beta^i, \qquad Y = \sum_{i=0}^{n-1} Y_i \beta^i \tag{4.4}$$

$$X_i, Y_i \in \{0, 1, \ldots, \beta - 2, \beta - 1\} \tag{4.5}$$

Lets define $MMD(X, Y, p)$ as the function that computes the Montgomery product of X, Y, processing them internally in a digit-by-digit fashion.

Some works in the literature have studied and proposed a hardware module for the Montgomery algorithm using a digit-digit approach. One of the most recent is reported in [3], and could well serve as the *MMD* function. Although the multiplier

presented in [3] was developed to be used in cryptography operations such as in RSA cryptosystems, the multiplier as it is could not be useful for constructing a hardware architecture for some exponentiation algorithm such as the MPL, or binary exponentiation. The main reasons are:

- The multiplier in [3] does not take into account that the result of the multiplication is used again as one of the input operands, as it is required in the exponentiation algorithms. The internal and external dataflow in the multiplier should be redesigned to avoid additional and redundant storage.
- In [3], the partial results at each iteration i are stored in a shift register, not in a memory. Thus, additional latency would be required to move the content of the shift register to memory if the partial result is required in future operations, such as in exponentiation in $\mathbb{F}_p$.

The first step in the proposed design methodology was to redesign the Montgomery hardware architecture in [3] in order to have a useful *MMD* module based on the Montgomery multiplier able to allow compact hardware architecture for $\mathbb{F}_p$ exponentiation.

Proposed hardware architecture for digit-digit $\mathbb{F}_p$ multiplication

Algorithm 2 was presented in [3] for iterative computation of a Montgomery product. In that algorithm, the product is obtained one digit at a time per clock cycle, stored and obtained from a shift register A that shifts k-bits (one digit) to the right at a time. This shift register represents variable A in Algorithm 2 that stores partial multiplications at each iteration i.

On one hand, the Montgomery multiplier in Algorithm 2 delivers the result in a shift register. On the other hand, input operands for the multiplier reside in memory blocks. This is the main inconvenient when using Algorithm 2 as a building block for $\mathbb{F}_p$ exponentiation architecture, because the multiplication result at iteration i (stored in a shift register) must be the input data to the multiplier at iteration $i+1$ (and must reside in a memory block). A shift register–memory block interface would be needed to solve this problem, of course with the associated cost of additional resources and increased latency.

In this research work, Algorithm 2 and its corresponding datapath were redesigned in such a way that the product and partial results in A reside in a memory block. The main changes in the dataflow include the control for the read/write operations

Algorithm 2 Iterative digit-digit MMA algorithm presented in [3]

Require: $X=\sum_{i=0}^{n-1} X_i\beta^i$, $Y=\sum_{i=0}^{n-1} Y_i\beta^i$, $p=\sum_{i=0}^{n-1} p_i\beta^i$,
1: $0 < X, Y < 2 \times p$, $R = \beta^n$, with $p' = -p^{-1} \bmod \beta$
Ensure: $A=\sum_{i=0}^{n-1} a_i\beta^i = X \times Y \times R^{-1} \mod p$
2: $A \leftarrow 0$;
3: **for** $i \leftarrow 0$ to $n-1$ **do**
4: $\quad c^{<0>} \leftarrow 0$
5: $\quad$ **for** $j \leftarrow 0$ to $n-1$ **do**
6: $\quad\quad s^{<j>} \leftarrow [A_0 + X_j \times Y_i]$
7: $\quad\quad$ **if** $j = 0$ **then**
8: $\quad\quad\quad q^{<i>} \leftarrow (s^{<j>} \times p') \mod \beta$
9: $\quad\quad$ **end if**
10: $\quad\quad r^{<j>} \leftarrow q^{<i>} \times p_j$
11: $\quad\quad \{c^{<j+1>}, t^{<j>}\} \leftarrow s^{<j>} + r^{<j>} + c^{<j>}$
12: $\quad\quad A \leftarrow SHR(A)$
13: $\quad\quad A_{n-1} \leftarrow t^{<j>}$
14: $\quad$ **end for**
15: $\quad A \leftarrow SHR(A)$
16: $\quad A_{n-1} \leftarrow c^{<n>}$
17: **end for**
18: **return** A;

over A in lines 5, 11, 12, 14 and 15 in Algorithm 2. With these changes, the partial Montgomery multiplication at the end of iteration i can be now treated as an input operand at iteration $i+1$ by multiplexing data ports in the corresponding memory blocks, thus avoiding the introduction of more logic and time overhead.

Algorithm 2 is based on the Montgomery algorithm proposed by C. Walter [4], Algorithm 3. From a sequential computing approach, lines 3 and 4 of Algorithm 3 could be performed by the set of operations described in Eq. 4.6. Once $q^{<i>}$ has been computed, partial multiplications $t_1 = X \times Y_i$ and $t_4 = q^{<i>} \times p$, and addition $t_5 = A^{<i>} + t_1$ could be performed in a digit by digit fashion. That is, for each iteration i in Algorithm 3, $A^{<i+1>}$ is computed by processing iteratively digits X_j, A_j, and p_j from X, $A^{<i>}$, and p respectively, thus computing a digit j of $A^{<i+1>}$ at a time (see Fig. 4.4).

$$\begin{aligned} t_1 &= X \times Y_i & t_4 &= q^{<i>} \times p \\ t_2 &= A_0 + X_0 \times Y_i & t_5 &= A^{<i>} + t_1 + t_4 \\ q^{<i>} &= t_2 \times p' \bmod \beta & A^{<i+1>} &= t_5/\beta \end{aligned} \tag{4.6}$$

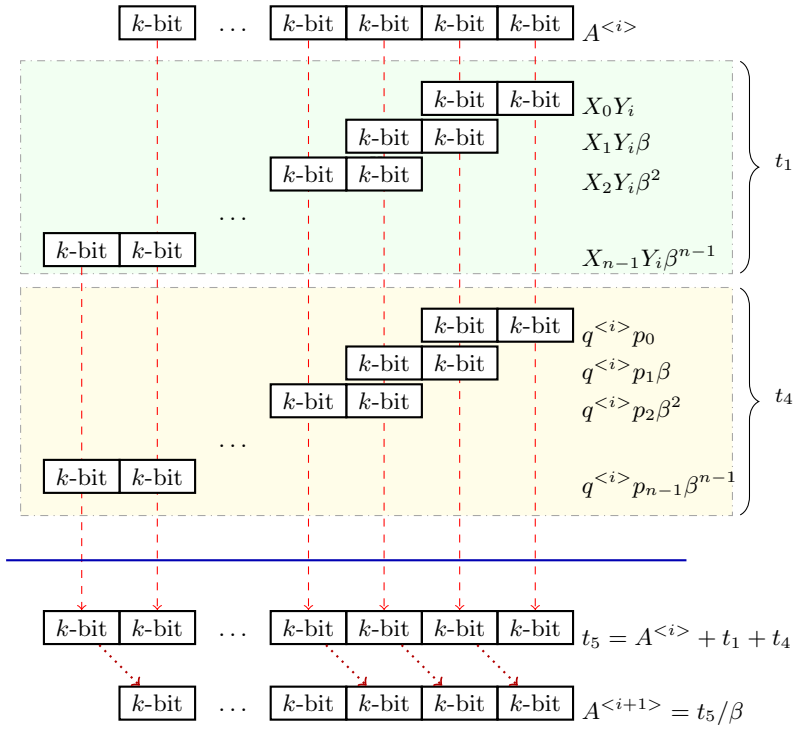

Figure 4.4: $A^{<i+1>}$ computation in a digit by digit approach.

Algorithm 3 Iterative Montgomery Multiplication [4]

Require: Integer X and Y, with $0 \leqslant X, Y < 2 \times p$, $R = \beta^{n+1}$ with $\gcd(p, \beta) = 1$, and $p' = -p^{-1} \bmod \beta$

Ensure: $A = X \times Y \times R^{-1} \bmod p = \sum_{i=0}^{n} A_i \beta^i$

1: $A \leftarrow 0$;
2: **for** $i \leftarrow 0$ to n **do**
3: $\quad q^{<i>} \leftarrow (A_0 + X_0 \times Y_i) \times p' \bmod \beta$
4: $\quad A^{<i+1>} \leftarrow ([A^{<i>} + X \times Y_i] + q^{<i>} \times p)/\beta$
5: **end for**
6: **return** A_n;

Fig. 4.4 shows digit by digit operations for computing $A^{<i+1>}$ iteratively. At the beginning of iteration i, $q^{<i>}$ is computed. Then, each digit of $A^{<i+1>}$ is obtained at each next clock cycle j. Note that the first digit (always equal to zero) will be discarded at the end of iteration i when the operation t_5/β executes. So, digits of $A^{<i+1>}$ must be stored in the corresponding output memory starting from iteration $j = 1$.

Algorithm 4 reflects modifications to Algorithm 2 needed for computing a digit-digit Montgomery multiplication, well suited to be used in the proposed $\mathbb{F}_p$ exponentiator.

Algorithm 4 New iterative Montgomery Multiplication algorithm

Require: $X=\sum_{i=0}^{n-1} X_i\beta^i$, $Y=\sum_{i=0}^{n-1} Y_i\beta^i$, $p=\sum_{i=0}^{n-1} p_i\beta^i$,
1: $0 < X, Y < 2 \times p, R = \beta^n$ with $p' = -p^{-1} \mod \beta$
Ensure: $A = \sum_{i=0}^{n-1} a_i\beta^i = X \times Y \times R^{-1} \mod p$
2: $A \leftarrow 0$;
3: **for** $i \leftarrow 0$ to $n-1$ **do**
4: $\quad c^{<0>} \leftarrow 0$
5: $\quad$ **for** $j \leftarrow 0$ to $n-1$ **do**
6: $\quad\quad s^{<j>} \leftarrow [A_j + X_j \times Y_i]$
7: $\quad\quad$ **if** $j = 0$ **then**
8: $\quad\quad\quad q^{<i>} \leftarrow (s^{<j>} \times p') \mod \beta$
9: $\quad\quad$ **end if**
10: $\quad\quad r^{<j>} \leftarrow q^{<i>} \times p_j$
11: $\quad\quad \{c^{<j+1>}, t^{<j>}\} \leftarrow s^{<j>} + r^{<j>} + c^{<j>}$
12: $\quad\quad$ **if** $j > 0$ **then**
13: $\quad\quad\quad A_{j-1} \leftarrow t^{<j>}$
14: $\quad\quad$ **end if**
15: $\quad\quad$ **end for**
16: $\quad\quad A_{n-1} \leftarrow c^{<n>}$
17: $\quad$ **end for**
18: $\quad$ **return** A;

The inner loop of Algorithm 4 requires n clock cycles. One clock cycle is needed at the beginning to compute $q^{<i>}$. One clock cycle at the end of the inner loop is necessary to store the last carry, $c^{<n>}$, in memory A, as explained previously. So, the computing of $q^{<i>}$ $(i > 0)$ and the writing of $c^{<n>}$ can occur during the same clock cycle. Additionally, the output of each memory bank can be pipelined to reduce the critical path at the cost of an extra cycle in the latency. If this is done, the total latency of the hardware module for *MMD* implementing Algorithm 4 requires $n(n+1)+4$ clock cycles.

Figure 4.5 shows the dataflow for read and write operations over A memory. Two counters are used to address each digit in A memory, $addWrA$ for writing and $addRdA$ for reading. j value for any of these counters indicates reading or writing of A_j digit.

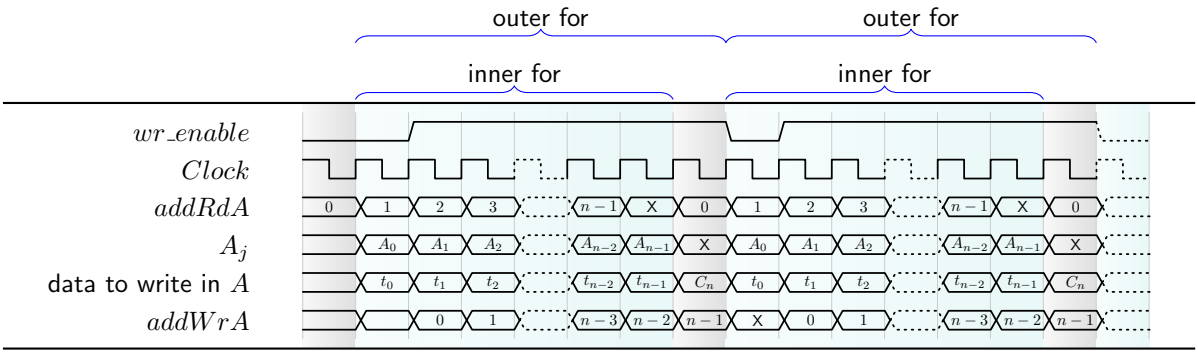

Figure 4.5: Dataflow in variable A acording to algorithm 4

The proposed hardware architecture for digit-by-digit Montgomery multiplication is shown in Fig. 4.6. In that figure, module for implementing *MMD* function has three $k \times k$ multipliers, four internal registers, and two $2k$-bit adders. $q^{<i>}$, which is computed only at the first j-iteration, depends of p' and t_2. Once $q^{<i>}$ is computed, t_1 and t_4 could be computed in parallel. Finally, partial results are added to obtain $A^{<i+1>}$. A control module orchestrates dataflow from and to memory blocks.

Control Module

In order to analyze the impact of control logic module in entire hardware architecture for $\mathbb{F}_p$ multiplier, two implementation options were followed: the first one is based on Finite State Machine (FSM) and the second one was a microcode approach.

Control module for the Montgomery multiplier implemented with an FSM only depends on the actual state. FSM is mainly constructed with counters, comparators, and registers. An FSM implementation requires a considerable amount of standard logic to implement all these components together with the logic to enable or disable control signals in those components depending on current state. Contrary, a microprogramming approach stores necessary signals in a memory, and only logic to read these instructions are needed as well as a memory block to store the microprogram.

Suppose that operand has $size = 1024$ and digit size is $k = 16$, so there are $1024/16 = 64$ digits per operands, all of them sequentially accessed. Since latency in Montgomery multiplier is $64 * 65 + 4 = 4162$, a Montgomery multiplication, for this

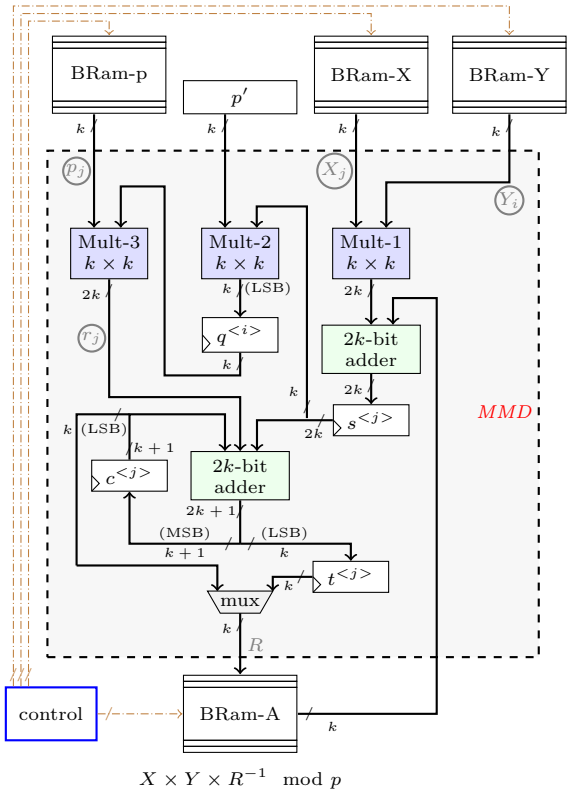

Figure 4.6: New digit-digit Montgomery multiplier architecture (Algorithm 4), memory and result reside in memory blocks.

Table 4.2: Microcode for the control module.

addr-x	addr-p	addrA	wrA	rst-cj	load-qi	mux-cj	to store
0	3f	3b	1	0	0	0	0x00ffb8
0	0	3c	1	0	0	0	0x0003c8
1	0	3d	1	0	0	0	0x0103d8
2	1	3e	1	1	1	0	0x0207ee
3	2	3f	1	1	0	1	0x030bfd
4	3	3f	0	0	0	0	0x040ff0
5	4	0	1	0	0	0	0x051008
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
3e	3d	39	1	0	0	0	0x3ef798
3f	3e	3a	1	0	0	0	0x3ffba8

example, takes 4162 cycles. Note that at each i iteration of Algorithm 4, control signals are identical. Consequently, it is possible to store only signals for one iteration and read them many times. In previous example, it is required to store only 65 control signals and read them 64 times. Microcode for architecture in Figure 4.6 is presented in Table 4.2. Address of Bram-y memory is used as index to count how many times the microcode is read. We propose to use the block memory that stores operand P to store the microcode, and use a dual port memory; one port to read operand P and other to read the microcode.

4.2.1 A variant of the digit-digit Montgomery implementation

Montgomery algorithm takes advantage of the radix $\beta = 2^k$ to make reduction step easy for hardware implementations since division and modulus operation for a power of two are cheaper in hardware. However, when digit size grows, for example from 32 to 64 bits, hardware resources required for the hardware architecture grown very fast. So, in this section, it is proposed a variant digit size for the Montgomery multiplication. In this way, the datapath can process operands in different digit size which made hardware implementation resources grow slower than if the same digit size is used for the two operands.

The proposed approach to accomplish a Montgomery multiplication with a general digit-digit approach is to use two digit sizes, one for X and another for Y. In this way the numbers X and Y could be represented in a different base $\beta = 2^k$:

$$X = \sum_{i=0}^{n_x-1} x_i \beta_x^i \qquad\qquad Y = \sum_{i=0}^{n_y-1} y_i \beta_y^i \tag{4.7}$$

Whit this representation, it is assumed that X is n_x digits in size, and Y is n_y digits in size. Furthermore, the digits size for X is k_x-bits, and k_y-bits for Y. Consequently, the partial multiplication $x \times y_i$ could be implemented in digit-digit fashion with different digit sizes for X and Y. It is is assumed that the digit size $k_x > k_y$.

Partial results to calculate A could be implemented in this new digit-digit size with two different digit sizes. Operand X, P, and A are partitioned in n_x digits of k_x-bits in size. The Y value is partitioned in n_y digits of k_y-bits in size. With this new configuration, A result can be computed in this new digit-digit approach, see Figure 4.7.

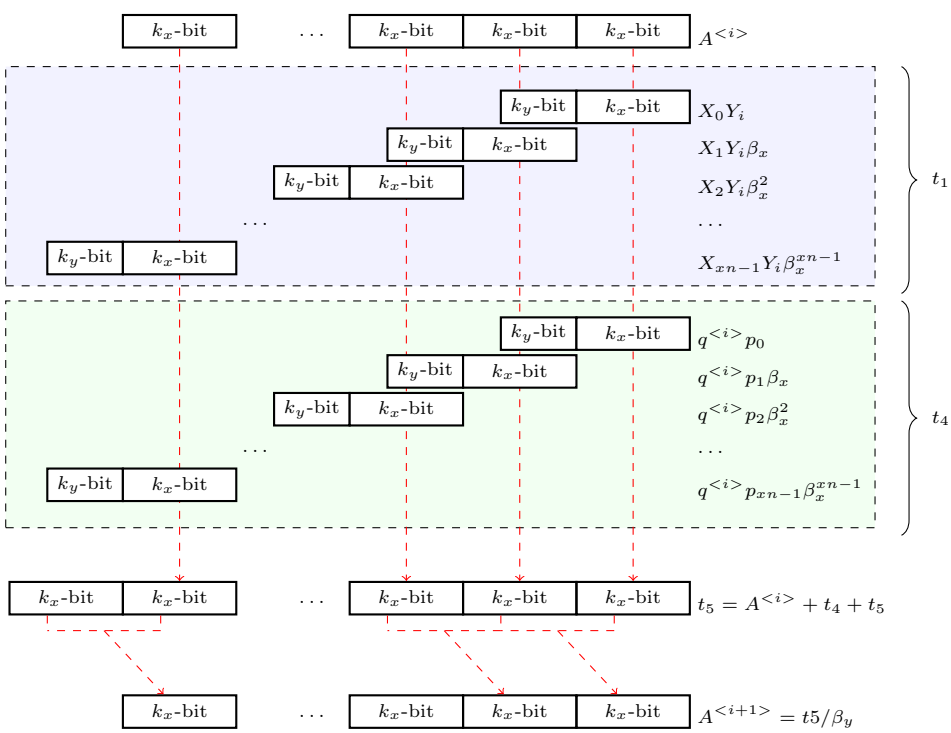

Figure 4.7: $A^{<i+1>}$ computation using different digit sizes for X and Y.

New variant digit-digit approach makes possible to calculate new partial result A in an iterative way using digits of k_x-bits in size, and having a carry of k_y-bits in the partial multiplications. In each clock cycle, a digit result of A is computed. At the end of each iteration, one digit of k_y bits is discarded. This can be achieved do not writing

the first k_y bits of the first computed digit A_0, see Figure 4.7. So, at each iteration, there are required to compute two partial values t to write one digit of the new value of $A^{<i+1>}$.

Algorithm 5 Montgomery digit-digit multiplication with different digit size.

Require: $X = \sum_{i=0}^{xn-1} X_i \beta_x^i$, $Y = \sum_{i=0}^{yn-1} Y_i \beta_y^i$, $p = \sum_{i=0}^{xn-1} p_i \beta_x^i$
Require: $0 < X, Y < 2 * M, R = \beta_x^{xn}$, $\gcd(M, R) = 1$, and $M' = -M^{-1} \bmod \beta_y$
Ensure: $A = \sum_{i=0}^{xn-1} A_i \beta_x^i = X \times Y \times R^{-1} \bmod M$

```
A ← 0;
for i ← 0 to yn − 1 do
    c^<0> ← 0
    for j ← 0 to xn − 1 do
        s^<j> ← [A_j + X_j * Y_i]
        if j = 0 then
            q^<i> ← (s^<j> * p') mod β_y
        end if
        r^<j> ← q^<i> * p_j
        {c^<j+1>, t^<j>} ← s^<j> + r^<j> + c^<j>
        if j > 0 then
            A_{j−1} ← t^<j>(yk − 1 downto 0)||t^<j−1>(xk − 1 downto yk)
        end if
    end for
    A_{xn−1} ← c^<xn>||t^<xn−1>(xk − 1 downto yk)
    return A
end for
```

The proposed algorithm for the alternative digit-digit approach is shown in Algorithm 5, and the proposed hardware architecture that implements this algorithm is presented in Figure 4.8. Since in the finite field exponentiation implementations it is desirable that operands and partial results will be stored in memory blocks with the same word-size, in this approach it is used digit size xk as memory word. However, the reading of memory BRAM-Y requires an extra component to reduce the operand to the digit size yk as can be seen in Figure 4.8.

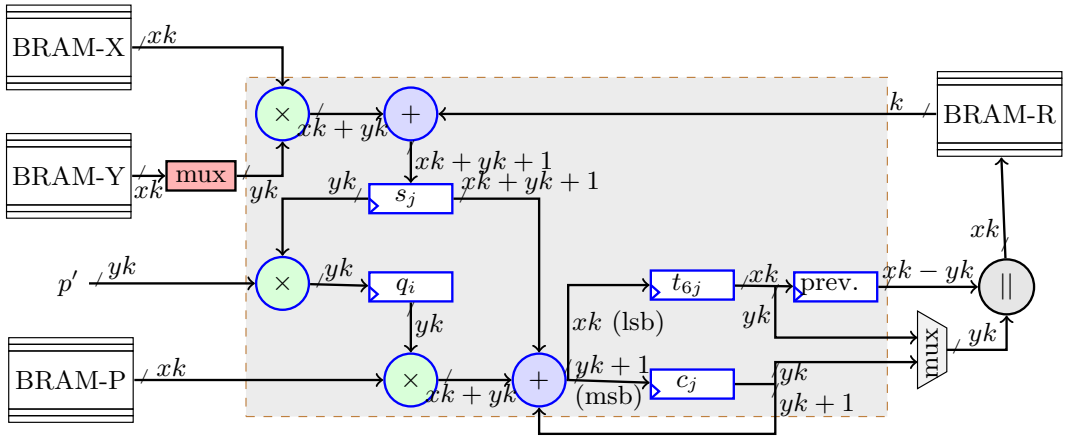

Figure 4.8: Montgomery multiplier digit-digit different digit size approach.

4.3 Compact exponentiation in $\mathbb{F}_p^*$

In this section, it is described the proposed finite field operator for the $\mathbb{F}_p^*$ exponentiator. The main goal of this hardware architecture is to reduce the required hardware resources compared to state of the art $\mathbb{F}_p^*$ exponentiators. To reduce the hardware required it is proposed to implement the exponentiator in a digit-digit approach making use of the Montgomery multiplier proposed in the previous section.

The finite field $\mathbb{F}_p$ with p a prime number is defined as the set of integers $\{0, 1, \ldots, p-1\}$ together with operations of addition and multiplication modulo p [21]. Exponentiation in $\mathbb{F}_p$ is defined as $g^e \mod p$ with $g \in \mathbb{F}_p$ and $e \in \mathbb{N}$. The basic method for exponentiation by multiplying g by itself $e-1$ times is totally inefficient. So, faster algorithms have been proposed to compute g^e, one of the most used nowadays is the Montgomery Powering Ladder method [88].

The MPL algorithm was originally proposed as a way to speed up the scalar multiplication in the elliptic curve domain [88]. Later, Joe and Yen [43] extended its scope to execute exponentiation in an abelian group. The main advantages of MPL are that it does not have conditional jumps nor extra operations, as in other approaches, which makes it resistant to certain kind of side-channel attacks, such as the Simple Power Analysis (SPA) attack [52]. The MPL method for $\mathbb{GF}(p)$ exponentiation is listed in Algorithm 6. It is assumed that the exponent e is L bits in size, and e_i is the ith bit of e.

The crucial operation in the MPL algorithm is $\mathbb{F}_p$ multiplication. One of the most used algorithms for efficient multiplication in $\mathbb{F}_p$ is the Montgomery method [55]. This algorithm employs only simple addition, subtraction and shifts operations to avoid

Algorithm 6 MPL method for exponentiation in $\mathbb{GF}(p)$

Require: $g \in \mathbb{GF}(p)$, $e = (e_{L-1}, \cdots, e_0)_2 \in \mathbb{N}$ and p a prime number defining $\mathbb{GF}(p)$
Ensure: $g^e \mod p$

```
R0 ← 1; R1 ← g;
for i = L − 1 downto 0 do
    if e_i = 1 then
        R0 ← R0 × R1 mod p;
        R1 ← R1 × R1 mod p;
    else
        R0 ← R0 × R0 mod p;
        R1 ← R1 × R0 mod p;
    end if
end for
return R0;
```

trial division by modulus p, which is very expensive in hardware implementations.

Lets define $MMD(X, Y, p)$ as the function that computes the Montgomery product of X, Y, processing them internally in a digit-by-digit fashion. With previous notation, Algorithm 6 can be described as Algorithm 7, where exponentiation operation g^e mod p is computed using digit-by-digit processing. In that algorithm, it is assumed that both g and g^e are in the Montgomery domain. The exponent e is expressed in the same way than in Algorithm 6, but '1' must be treated as a Montgomery number, that is, it must be transformed to $1 \times 2^N \mod p$.

Algorithm 7 Digit-digit MPL algorithm

Require: $e = (e_{L-1}, \cdots, e_0)_2$, $g = \sum_{i=0}^{n-1} g_i \beta^i$, p
Ensure: $C = \sum_{i=0}^{n-1} C_i \beta^i = (g^e) \times R \mod p$

```
X ← 1 × 2^N mod p;
Y ← g;
for i = L − 1 downto 0 do
    if e_i == 1 then
        X ← MMD_0(X, Y, p);
        Y ← MMD_1(Y, Y, p);
    else
        X ← MMD_0(X, X, p);
        Y ← MMD_1(Y, X, p);
    end if
end for
return X;
```

A direct hardware implementation of Algorithm 7 requires two modules for the *MMD* function, say MMD_0 and MMD_1, which can work in parallel at each iteration. The main advantage of Algorithm 7 is that the k-bit digits of operands X and Y can be stored in $n \times k$ memory blocks, so the hardware realization of the *MMD* function does not require internal logic to store its operands.

However, note that in Algorithm 7 partial result at iteration i become the input data at iteration $i+1$. That is, operands' digits are used and overwritten during the same iteration. So, the main challenge to implement Algorithm 7 without using additional and redundant storage for operands is to design a control logic that correctly parses, accesses and reuses operands' digits directly from block memories.

The critical component in Algorithm 7 is the embedded Montgomery multiplier. The Montgomery digit-digit multiplier developed in the previous section could be ideally used as the MMD_0 and MMD_1 modules required in Algorithm 7.

Hardware architecture for MPL

The hardware module for *MMD* in Fig. 4.6 is used to construct the hardware architecture for the MPL algorithm. Inputs of *MMD* module are the digits X_j, Y_i, p_j, p', and A_j digits of the partial multiplication $A^{<i>}$.

Consider $e_i = 1$ in execution of Algorithm 7. Y operand is two inputs to the *MMD* multiplier at line 6. Thus, digits from Y are read at the outer (Y_i) and at the inner loop (Y_j) of Algorithm 4. The same applies to X when $e_i = 0$. Therefore, we considered dual port memories when designing the MPL architecture to store and access digits from X and Y to execute Algorithm 7.

The *MMD* hardware module in Fig. 4.6 now delivers multiplication result to a memory, and that memory becomes in one Montgomery Multiplier operand at the next iteration. Instead of moving all the content of the memory assigned to $A^{<i+1>}$ to one of the input memories assigned to X or Y, our approach is to define a strategy to switch the role of memories: at one time behaving as an input operand (with read operations) and at another time behaving as the multiplication result (with write operations).

In this context, a total of four memories are required: BRam-XX, BRam-YY, BRam-X, and BRam-Y. At the beginning of Algorithm 7, g and '1' are loaded into BRam-X and BRam-Y respectively, and BRam-XX and BRam-YY play the role of write memories. In the next iteration, memories change their role, so BRam-XX and BRam-YY are input operands and BRam-X and BRam-Y are now write memories to store the multiplication result in the next iteration. This process continues until all bits of the exponent are processed.

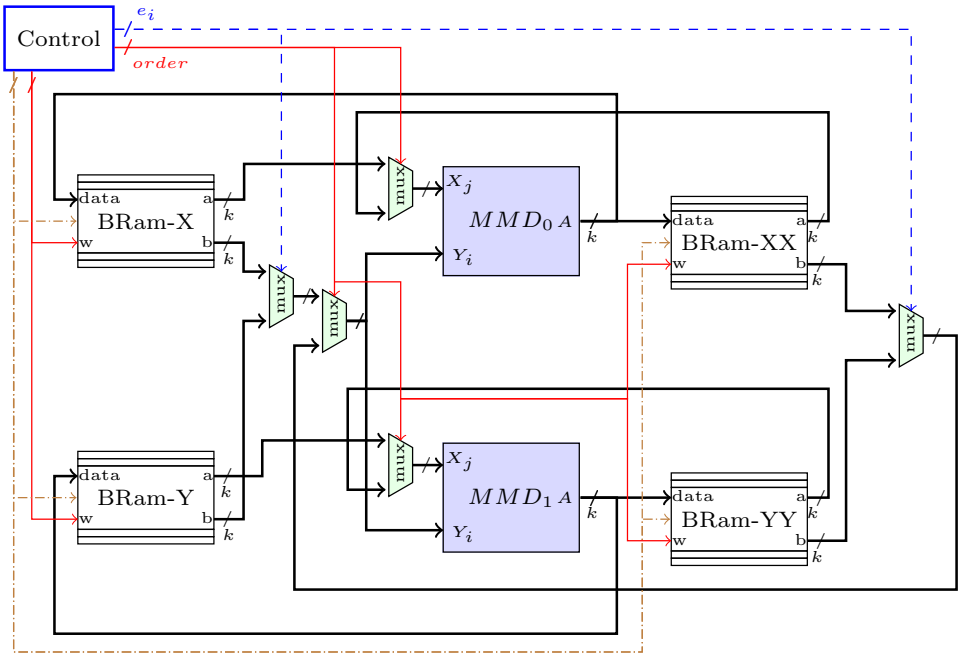

Figure 4.9: Digit-digit Montgomery Powering Ladder architecture (Algorithm 7).

The hardware architecture for the MPL algorithm is shown in Fig. 4.9. The main blocks, denoted by MMD_0 and MMD_1, are digit-by-digit Montgomery multipliers executing Algorithm 4. Input ports for these modules are current input operands at iteration i and output port corresponds to the resulting multiplication delivered digit-by-digit. Other signals such as p', p and A_j for *MMD* shown in Fig. 4.6 have been omitted for clarity.

A control unit manages the entire dataflow and stimulates memory blocks for reading and writing. As it was commented before, dual-port memories are used to access two digits at a time from an operand, respectively addressed by outer and inner loops in Algorithm 4. These two ports are indicated in block memories of Fig. 4.9 as 'a' and 'b'.

The proposed hardware architecture presented in Fig. 4.9 takes advantage of available embedded BRAMs in commercial FPGAs. Exponent e, modulus p, and four temporary variables BRam-X, BRam-Y, BRam-XX and BRam-YY were mapped to FPGA BRAMs. e exponent and p modulus were mapped to single port BRams since only one word per cycle is required. However, the other operands were mapped to dual-port BRAMs to read from and write to the memory during the same clock cycle. Since all the operands are stored in independent BRAMs, they can be accessed in parallel

without memory bottlenecks. Nevertheless, in digit-digit multiplication approach, only one digit (word) per clock cycle is computed at a time, thus increasing latency, see Algorithm 4.

Although reusing of memories saves FPGA resources, the control unit to appropriately stimulate these memories (to read and write digits of operands and partial results) gets more complex. Each memory port requires signals for data input/output, read/write addresses, enable/disable signals, among others. The control unit is in charge of all these signals for orchestrating the algorithm execution and the data flow.

FPGA families have a different number of embedded BRAMs, with a maximum word size. When the word size is bigger than the one allowed, multiple BRAMs are combined to create a single larger Random-access memory (RAM). That can increase memory traffic, area and access time due to interconnections between BRAMs. Because of that, in this work, only word sizes (digit size) of 4, 8, 16, 32, and 64 were implemented.

A relevant aspect of hardware architectures for cryptography applications is their resistance to side-channel attacks. In order to reveal certain secret information when a hardware module performs an encryption/decryption operation, an attacker can perform an analysis of the power dissipation, the electromagnetic radiation, or the operating time of internal operations while the hardware module executes. The Simple Power Analysis (SPA) and Differential Power Analysis (DPA) proposed by Kocher [89] are two of the best-known attacks. However, constant time algorithms are resistant to certain side-channel attacks. A broad study about side-channel attacks is presented in [90]. Our proposed Algorithm 4 is a constant time algorithm as the MPL algorithm is. So the proposed algorithms favor the creation of hardware architecture resistant to some side-channel attacks such as SPA.

4.4 Operators over $\mathbb{F}_{2^m}$

In this section, it is presented a compact hardware architecture for multiplication over $\mathbb{F}_{2^m}$. The target of the proposed design is resource-constrained devices, so small area is preferable to high-speed approaches. The proposed architecture can be used as a building block for elliptic curve cryptography implementations in resource-constrained devices.

Multiplication over $\mathbb{F}_{2^m}$.

Multiplication over $\mathbb{F}_{2^m}$ is one of the main operations in Elliptic Curve Cryptography (ECC) over $\mathbb{F}_{2^m}$. The Most Significant Element first (MSE) and Least Significant Element first (LSE) are commonly used algorithms to compute multiplications over $\mathbb{F}_{2^m}$. The MSE and LSE are digit-serial algorithms. Hardware implementations of LSE multiplication algorithm require more hardware resources than hardware implementations of MSE first algorithms [72]. However, in this thesis project, a novel digit-digit $\mathbb{F}_{2^m}$ multiplier algorithm based on MSE algorithm has been proposed. The proposed algorithm uses a digit-digit approach that compared with the traditional approach allows smaller architectures. The main advantage of the proposed algorithm is that all partial operations are computed with digits, which leads to more compact implementations than algorithms that work with full operands.

It is well known that a $\mathbb{F}_{2^m}$ finite field could be defined using an irreducible polynomial: $F(x) = x^m + f_{m-1}x^{m-1} + \cdots + f_2x^2 + f_1x + 1$ where $f_i \in \mathbb{F}_{2^m}$ for $0 \leqslant i \leqslant m$ [48]. If α is a root of $F(x)$ then an element $A \in \mathbb{F}_{2^m}$ could be represented in polynomial basis as $A(\alpha) = \sum_{i=0}^{m-1} a_i\alpha^i$, $a_i \in \mathbb{F}_2$. Furthermore, since α is a root of $F(x)$:

$$F(\alpha) = \alpha^m + \sum_{i=0}^{m-1} f_i\alpha^i = 0 \tag{4.8}$$

$$\alpha^m = -\sum_{i=0}^{m-1} f_i\alpha^i = \sum_{i=0}^{m-1} (-f_i)\alpha^i = \sum_{i=0}^{m-1} f_i\alpha^i \tag{4.9}$$

According to [73], all elements $\alpha^{m+i} \in A$, $i \geqslant 0$ can be reduced using Equation 4.9.

This work follows the convention that $A, B, C \in \mathbb{F}_{2^m}$, being A the multiplicand, B the multiplier and $C = A \times B \bmod F(\alpha)$. Elements A and B are represented by polynomials:

$$A = \sum_{i=0}^{m-1} a_i\alpha^i, \qquad B = \sum_{i=0}^{m-1} b_i\alpha^i, \qquad a_i, b_i \in \mathbb{F}_{2^m}, \text{for } 0 \leqslant i < m \tag{4.10}$$

C element is given by:

$$C = A \times B \bmod F(\alpha) = \left(\sum_{i=0}^{m-1}\sum_{j=0}^{m-1} a_ib_i\right) \bmod F(\alpha) = \sum_{i=0}^{m-1} c_i\alpha^i \tag{4.11}$$

To perform reduction in the partial product $A \times B$, a polynomial of degree $2m-2$, is not practical due to a $2m-2$ bits register to store the partial result require hardware resources that can be saved. In this work an iterative reduction it is presented.

An element $B \in \mathbb{F}_{2^m}$ can also be represented as a sum of $w = \lceil m/d \rceil$ polynomials (digits) each of d coefficients in $\mathbb{F}_2$ [79].

$$B = \sum_{i=0}^{m-1} b_i \alpha^i = \sum_{i=0}^{w-1} B_i \alpha^{id}, \qquad \text{where} \quad B_i = \sum_{j=0}^{d-1} b_{id+j} \alpha^j \tag{4.12}$$

So, multiplication $C = A \times B \bmod F(x)$ can be expressed as:

$$\begin{aligned} C &= A \times B \bmod F(x) \\ &= \left(A \times \sum_{i=0}^{w-1} B_i \alpha^{di} \right) \bmod F(\alpha) \end{aligned} \tag{4.13}$$

Modulus operation is distributive [48]. So, equation 4.13 can be rewritten as:

$$\begin{aligned} C &= \sum_{i=0}^{w-1} (AB_i \alpha^{di} \bmod F(\alpha)) \\ &= AB_0 \bmod F(\alpha) + \\ &\quad AB_1 \alpha^d \bmod F(\alpha) + \\ &\quad AB_2 \alpha^{2d} \bmod F(\alpha) + \\ &\quad \vdots \\ &\quad AB_{w-1} \alpha^{(w-1)d} \bmod F(\alpha) \end{aligned} \tag{4.14}$$

At each iteration partial product $P^{<i>}(\alpha) = AB_i$, a polynomial of degree $d + m - 2$, must be reduced $\bmod F(\alpha)$. Parsing elements of B from left to right (MSE), C computation at iteration i, $0 \leqslant i \leqslant w - 1$, is determined by recurrence:

$$C^{<0>} = 0 \tag{4.15}$$

$$C^{<i+1>} = \alpha^d (C^{<i>} \bmod F(\alpha)) + P^{<w-1-i>}(\alpha) \tag{4.16}$$

Where polynomial $\alpha^d (C^{<i>} \bmod F(\alpha))$ is at most of degree $d + k - 1$, while $P^{<i>}$ is of degree $(d + k - 2)$. After w iterations $C^{<w-1>}$, a polynomial of degree $d + k - 1$, needs reduction. So, it is required an extra iteration with $B_{-1} = 0$. In this case $P^{<-1>} = 0$, and $C^{<w>} = \alpha^d (C^{<w-1>} \bmod F(x))$ is the result. The final result is $C^{<w>}/\alpha^d$, that can be computed only discarded the digit $C_0^{<w>}$.

Degree of $C^{<i>}$ from Equation 4.16 is at most $(d + m - 1)$. So, it is required to reduce elements α^j of $C^{<i>}$, $m - 1 < j \leqslant d + m - 1$. Equation 4.9 can be

applied to achieve reduction of $C^{<i>}$. When $F(x)$ is a trinomial or pentanomial, $\alpha^m = \sum_{i=0}^{m-1} f_i \alpha^i = g(\alpha)$ a polynomial of degree g with $g < m$. Elements α^{m+t} with $t \leqslant m-1-g$ can be reduced using equivalence $\alpha^{m+t} \bmod F(\alpha) = g(\alpha)\alpha^t$. With this considerations, reduction in equation 4.16, $C^{<i>} \bmod F(\alpha)$, can be defined as:

$$\begin{aligned} C^{<i>} \bmod F(\alpha) &= \sum_{i=0}^{m-1} c_i \alpha^i + \left(\sum_{i=m}^{m-1+d} c_i \alpha^i \right) \bmod F(\alpha) \\ &= \sum_{i=0}^{m-1} c_i \alpha^i + \left(\sum_{i=0}^{d-1} c_{m+i} \alpha^i g(\alpha) \right) \bmod F(\alpha) \\ &= C_m^{<i>}(x) + C_d^{<i>}(x) \times g(x) \end{aligned} \tag{4.17}$$

$C^{<i>}$ is partitioned in two polynomials $C_m^{<i>}(x)$ and $C_d^{<i>}(x)$ of degree $m-1$ and d respectively. The partial multiplication $C_d^{<i>} \times g(x)$ will not require modular reduction if $d + g < m$. So, using 4.17 in 4.16 it is obtained:

$$C^{<i+1>} = \alpha^d (C_m^{<i>} + C_d^{<i>} \times g(x)) + P^{<w-1-i>} \tag{4.18}$$

4.5 Compact digit-digit $\mathbb{F}_{2^m}$ multiplication

In this section it is presented an iterative way to compute equation 4.18 in an iterative digit by digit way, and the corresponding hardware compact architecture. The polynomial $C_m^{<i>}$, $g(x)$ and A can be represented in $w = \lceil m/d \rceil$ digits. Since the B_i degree is $d-1$, $P^{<i>}$ computation can be achieved iteratively, taken digit B_i and iterating through A digits:

$$\begin{aligned} P^{<i>}(x) &= A(x) \times B_i(x) \\ &= \sum_{j=0}^{u-1} A_j \alpha^{jd} \times B_i(x) \\ &= \sum_{j=0}^{u-1} (A_j \times B_i) \alpha^{jd} \\ &= \sum_{j=0}^{u-1} P_j^{<i>} \alpha^{jd} \end{aligned} \tag{4.19}$$

And, $\alpha^D (C_m^{<i>}(x) + C_d^{<i>}(x) \times g(x))$ in equation 4.18 can be rewritten as:

$$\begin{aligned}
\alpha^{D}(C_m^{<i>}(x) + C_d^{<i>}(x) \times g(x)) &= \sum_{j-0}^{w-1} C_j^{<i>}\alpha^{jd+d} + C_d^{<i>}(x) \times \sum_{j=0}^{w-1} G_j\alpha^{jd+d} \\
&= \sum_{j=0}^{w-1} (C_j^{<i>} + C_d^{<i>}(x) \times G_j)\alpha^{jd+d} \\
&= \sum_{j=0}^{w-1} R_j^{<i>}\alpha^{jd+d} \qquad (4.20)
\end{aligned}$$

Once $P^{<i>}$ and $R^{<i>}$ are processed in an iterative way one digit at a time, equation 4.18 can be rewritten as:

$$C^{<i+1>} = \sum_{j-0}^{w-1} (R_j^{<i>}\alpha^{jd+d} + P_j^{<i>}\alpha^{jd}) \qquad (4.21)$$

At each iteration, values $P_j^{<i>}$ and $R_j^{<i>}$ can be computed in a parallel way. Furthermore, to be clearer in the computations in operations, $P_j^{<i>}$ and $R_j^{<i>}\alpha^d$ can be added to form a variable $S_j^{<i>}$. New variable $S_j^{<i>}$ is $d + d + d$ bits in size as shown in Figure 4.10.

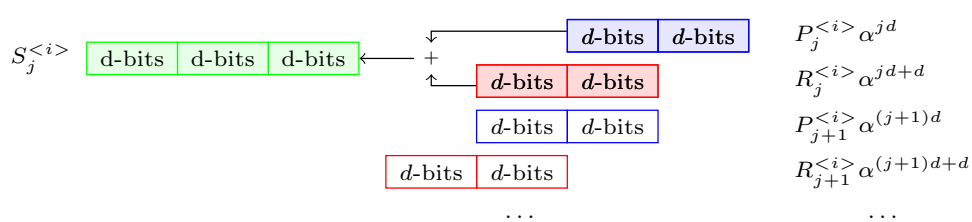

Figure 4.10: Computation of P_j and R_j.

Now at each iteration it is possible to compute a new value $S_j^{<i>}$, and a new digit of $C^{<i+1>}$ as shown in Figure 4.11.

At the end of each iteration i, it is required to store the new value of $C_d^{<i>}$.

With all these considerations the proposed algorithm for computing multiplication over $\mathbb{F}_{2^m}$ is presented in Algorithm 8.

Three different hardware architecture will be presented to compute multiplication over $\mathbb{F}_{2^m}$ according to Equation 4.16, Figure 4.11, and Algorithm 8.

4.5.1 Hardware architecture 1 (Compact implementation)

According to Figures 4.10 and 4.11 Equation 4.9 can be computed in an iterative way. Hardware architecture 1 is a straightforward implementation of Algorithm 8.

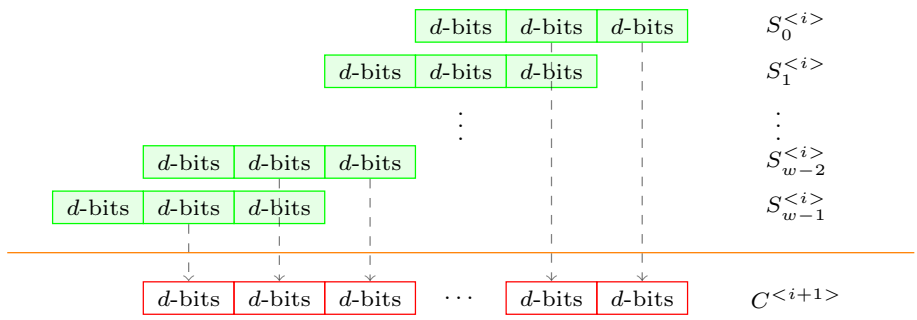

Figure 4.11: Partial computation of $S_j^{<i>}$.

Algorithm 8 Digit-digit $\mathbb{F}_{2^m}$ multiplier algorithm.

Require: $A, B, F \in \mathbb{F}_{2^m}$ ▷ $A = \sum_{i=0}^{w-1} a_i \alpha^{iD}$ ▷ $B = \sum_{i=0}^{w-1} b_i \alpha^{iD}$

```
cD ← 0
for i ← 0 to w + 1 do
    carry ← 0
    s ← 0
    for j ← 0 to w do
        P_i ← b_{digits−i} × a_{j+1}
        R_j ← c_j + cD × f_{j+1}
        s ← Pi + (R_j << d) + carry
        c_j ← s[d − 1 downto 0]
        carry ← s >> d
    end for
    cD ← s >> bitsLastDigit
end for
c_w ← carry
return c
```

▷ $c = A \times B \mod F$ ▷ $c = \sum_{i=0}^{w-1} c_i \alpha^{iD}$

Figure 4.12 shows the proposed hardware architecture. In this architecture there are required two $d \times d$ partial multipliers, one $2d + d$ adder, one $2d + 3d$ adder, one d-register to store $C_d^{<i>}$, and one 2d register to store the partial carry.

The main components are two partial multipliers which required a considerable amount of hardware resources.

4.5.2 Hardware architecture 2 (Karatsuba version)

The second hardware architecture was implemented similar to architecture 1, but implementing partial $d \times d$ multipliers with the Karatsuba algorithm. In this approach the d-bits operands are partitioned in two $d/2$ operands and three $d/2 \times d/2$ partial

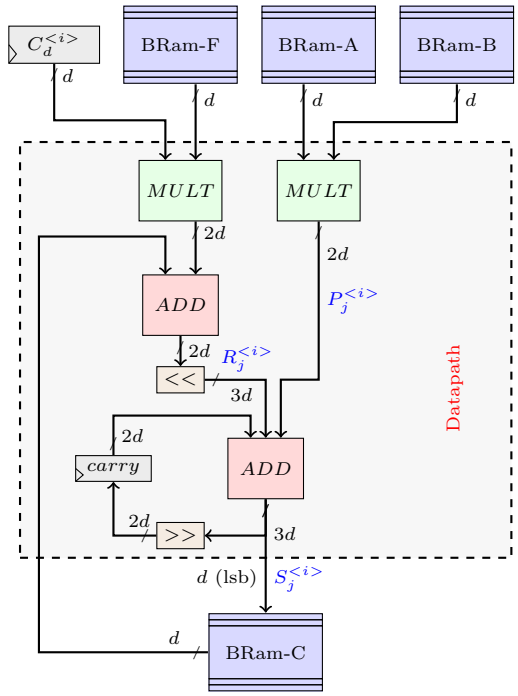

Figure 4.12: Hardware architecture 1.

multiplications are required. This approach can be used to compute partial $d/2 \times d/2$ multiplications in a recursive way. In this work, we use a case base of 4×4 multipliers. When a $d \times d$ multiplication is required, operands are partitioned and the Karatsuba method is used in a recursive way, but when a $d \times d$, with $d = 4$ is required the multiplier is implemented as architecture 1.

4.5.3 Hardware architecture 3 (One partial multiplier version)

The main components in architectures 1 and 2 are partial multipliers which require more than 50% of hardware resources for the $\mathbb{F}_{2^m}$ multiplier. This section presents a hardware architecture that only requires one partial $d \times d$ multiplier when a trinomial is used.

Table 4.3 shows polynomials recommended by National Institute of Standards and Technology (NIST). There are two trinomials and three pentanomials. If the trinomial $m = 233$ recommended by NIST is used polynomial $g(x) = x^{74} + 1$ is required for the reduction step. So, if $d = 74$ (digit size) is used, when a digit j of $g(x)$ (G_j) is read, only the two first digits will have a value of 1, when $j > 1$ digit G_j will be always 0. In this case, partial multiplier that compute $C_d^{<i>}(x) \times G_j$ always compute a multiplication of the form $(C_d^{<i>}(x) \times 1)$ or $(C_d^{<i>}(x) \times 0)$ which can be implemented only with a 'and'

Table 4.3: NIST recommended irreducible polynomials for binary fields.

m	Polynomial
571	$z^{571} + z^{10} + z^5 + z2 + 1$
409	$z^{409} + z^{87} + 1$
283	$z^{283} + z^{12} + z^7 + z^5 + 1$
233	$z^{233} + z^{74} + 1$
163	$z^{163} + z^7 + z^6 + z^3 + 1$

gate. In conclusion, when a trinomial of form $x^m + x^k + 1$ is used, it is possible define the digit size $d = k$. In this case, partial multiplier that compute $C_d^{<i>}(x) \times G_j$ can be implemented using only a multiplexer as is shown in Figure 4.13

When a digit size $d < k$ is used, first partial multiplication $C_d^{<i>}(x) \times G_j$ will be $C_d^{<i>}(x) \times 1 = C_d^{<i>}(x)$. It is required to calculate the digit in which bit k is stored, and in which position in the digit. So, when the digit that store the bit k is processed, it could be computed only shifting the value of $C_d^{<i>}(x)$. All others digits G_j are zero. Thus, it is not required to compute the partial multiplications nor to store the value of the full reduction polynomial.

4.6 Compact exponentiation in $\mathbb{E}(\mathbb{F}_{2^m})$

Elliptic Curve Cryptography (ECC) has been called to be the next generation cryptography ideal for resource-constrained devices [39, 91]. The main operation in ECC is the scalar multiplication $Q = kP$ where k is a private key, Q is a public key, and P is a base point on an elliptic curve $\mathbb{E}$. The private key k is very difficult to compute from knowledge of P and Q.

For cryptography applications, it is very important to select the correct parameters for ECC implementations. The main parameters to consider are the finite field, elliptic curves, coordinates system, and scalar multiplication algorithm. In this thesis research the next parameters were chosen:

Finite field: In this work the binary field $\mathbb{F}_{2^m}$ is selected due to it is well suited for hardware implementations since it is carry free [91]. Furthermore, the binary field allows simplified squared arithmetic.

Elliptic Curves: Elliptic curves 233 and 409 recommended by NIST were selected since a compact multiplier over $\mathbb{F}_{2^m}$ for finite fields generated by trinomials has been

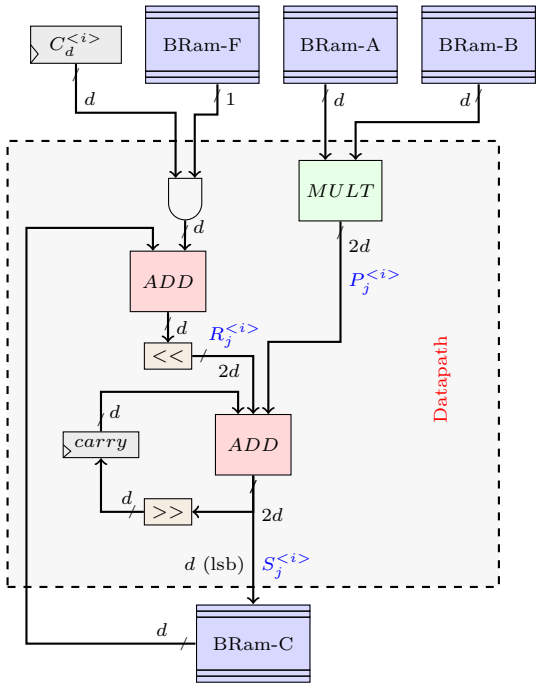

Figure 4.13: Hardware architecture 3.

proposed. For that, it is proposed to take advantage of this multiplier to be used as a basic building block for the ECC implementation.

Coordinates: Projective coordinates were selected since they reduce the number of inversions over $\mathbb{F}_{2^m}$ compared to affine coordinates when computing point addition in the elliptic curve.

kP **algorithm:** The Montgomery algorithm for elliptic curves, proposed by López-Dahab [53], (Algorithm 9) was used due to:

- It allows saving resources since point addition/double do not consider y coordinate.
- The field inversion operation is only required for coordinate conversion at the end of the main loop since projective coordinates are used.
- It is a constant time algorithm resistant to some side channels attacks such as Simple Power Analysis (SPA).

The point addition (Madd), point double (Mdouble), and conversion coordinates (Mxy) functions used in the Montgomery algorithm 9 are shown in Algorithms 10, 11 and 12.

Algorithm 9 Montgomery Scalar Multiplication

Require: $k \geqslant 0$
Require: $P = (x, y) \in \mathbb{E}(\mathbb{F}_{2^m})$

```
if k = 0 or x = 0 then
    return (0, 0).
end if
P1(X1, Z1) ← (x, 1)
P2(X2, Z2) ← (x^4 + b, x^2)
for i from l − 2 downto 0 do
    if k_i = 1 then
        P1 = Madd(P1, P2)
        P2 = Mdouble(P2)
    else
        P2 = Madd(P2, P1)
        P1 = Mdouble(P1)
    end if
end for
return Q = Mxy(P1, P2, P)                    ▷ Q = kP
```

Algorithm 10 Add Algorithm

```
procedure MADD(P1(X1, Z1), P2(X2, Z2), P(x, y))
    Z3 ← (X1Z2 + X2Z1)^2
    X3 ← xZ3 + X1X2Z1Z2
    return P3(X3, Z3)                        ▷ P3 = P1 + P2
end procedure
```

Algorithm 11 Doubling Algorithm

```
procedure MDOUBLE(P1(X1, Z1))
    Z2 ← X1^2 Z1^2
    X2 ← X1^4 + bZ1^4
    return P2(X2, Z2)                        ▷ P2 = 2P1
end procedure
```

Algorithm 12 Coordinates conversion projective-affine.

1: **procedure** GXY($P_1(X_1, Z_1), P_2(X_2, Z_2), P(x, y)$)
2: $\quad x_q \leftarrow X_1 Z_1^{-1}$
3: $\quad Y_{int} \leftarrow (X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)Z_1 Z_2$
4: $\quad y_q \leftarrow (x + x_q)Y_{int}(xZ_1 Z_2)^{-1} + y$
5: $\quad$ **return** $Q(x_q, y_q)$
6: **end procedure**

It is proposed to compute operations according to the schedule shown in Figure 4.15: Variables TA_i and TD_i are temporary. This schedule has been carefully selected not to overwrite temporary values that will be needed later. It is proposed to use two memory blocks to store one elliptic curve point $P(X, Y)$. So, at each iteration of algorithm 9, there are required two elliptic curve points P_1 and P_2. Furthermore, it is required to store partial results in temporary memories, since it is not possible overwriting operands memories because they are required in next operations. So, it is proposed to use two temporary points (4 memory blocks), shown in Figure 4.14.

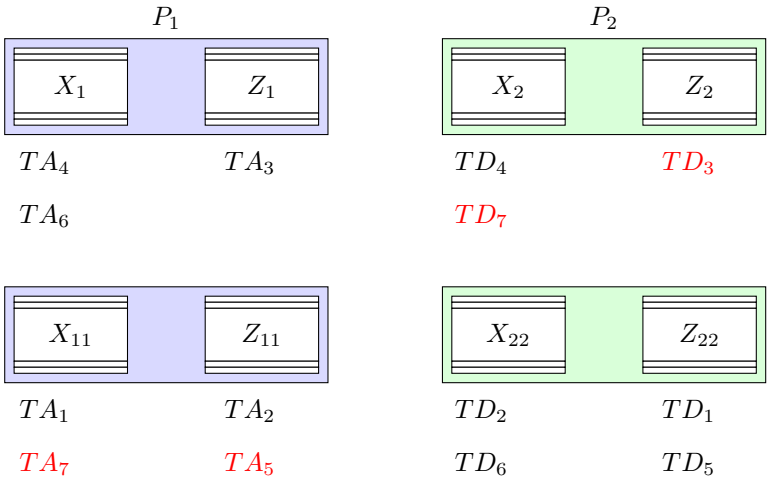

Figure 4.14: Memory blocks for partial result operations.

As it can be seen in Figure 4.15, there are required to compute some square operations in $\mathbb{F}_{2^m}$. It has been presented in Subsection 2.3.1 that the square operation of an element $A \in \mathbb{F}_{2^m}$ can be accomplished by inserting 0's between consecutive elements of A. So, only a reduction step is required to accomplish the square. For example, Algorithm 13 is the proposed algorithm for fast reduction proposed by NIST for the irreducible polynomial $f(z) = z^{233} + z^{74} + 1$ [40], and commonly used for reduction after the square operation.

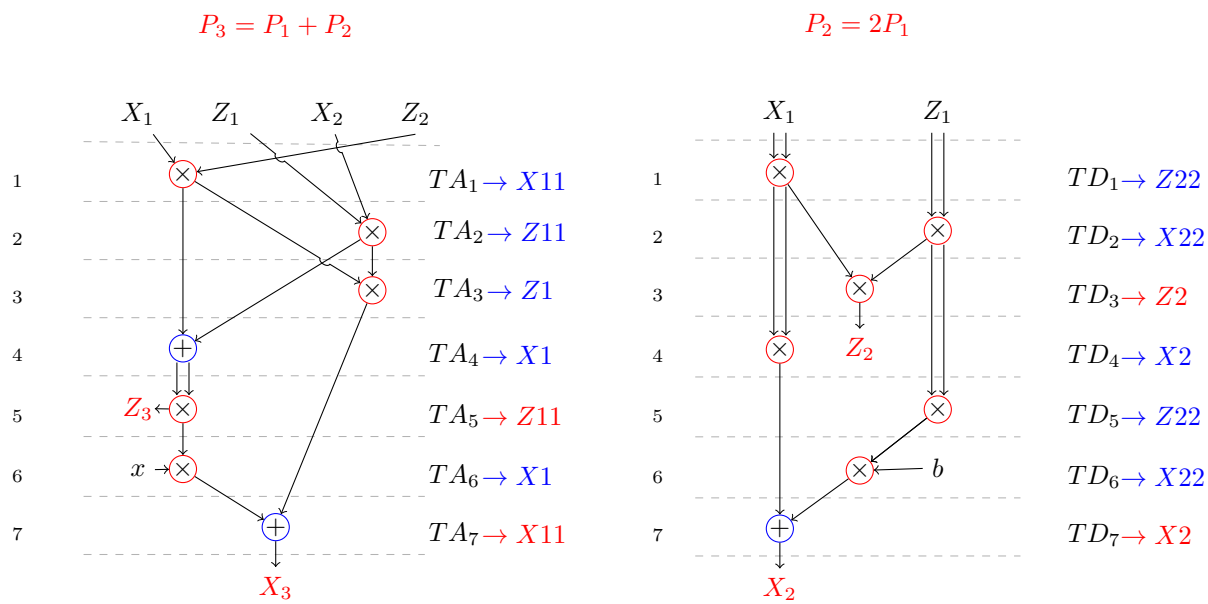

Figure 4.15: Schedule for the point addition and point double operations in the context of elliptic curves.

Algorithm 13 Fast reduction modulo $f(z) = z^{233} + z^{74} + 1$ (With $W = 32$)

Require: A binary polynomial $c(z)$ of degree at most 464.

Ensure: $c(z)$ mod $f(z)$.

```
for i ← 15 downto 8 do                          ▷ Reduce C[i]z^{32i} modulo f(z)
    T ← C[i].
    C[i − 8] ← C[i − 8] ⊕ (T << 23).
    C[i − 7] ← C[i − 7] ⊕ (T << 9).
    C[i − 5] ← C[i − 5] ⊕ (T << 1).
    C[i − 4] ← C[i − 4] ⊕ (T << 31).
end for
T ← C[7] >> 9.                                  ▷ Extract bits 9-31 of C[7]
C[0] ← C[0] ⊕ T.
C[2] ← C[2] ⊕ (T << 10).
C[3] ← C[3] ⊕ (T >> 22).
C[7] ← C[7]&0x1FF.                              ▷ Clear the reduced bits of C[7]
return (C[7], C[6], C[5], C[4], C[3], C[2], C[1], C[0]).
```

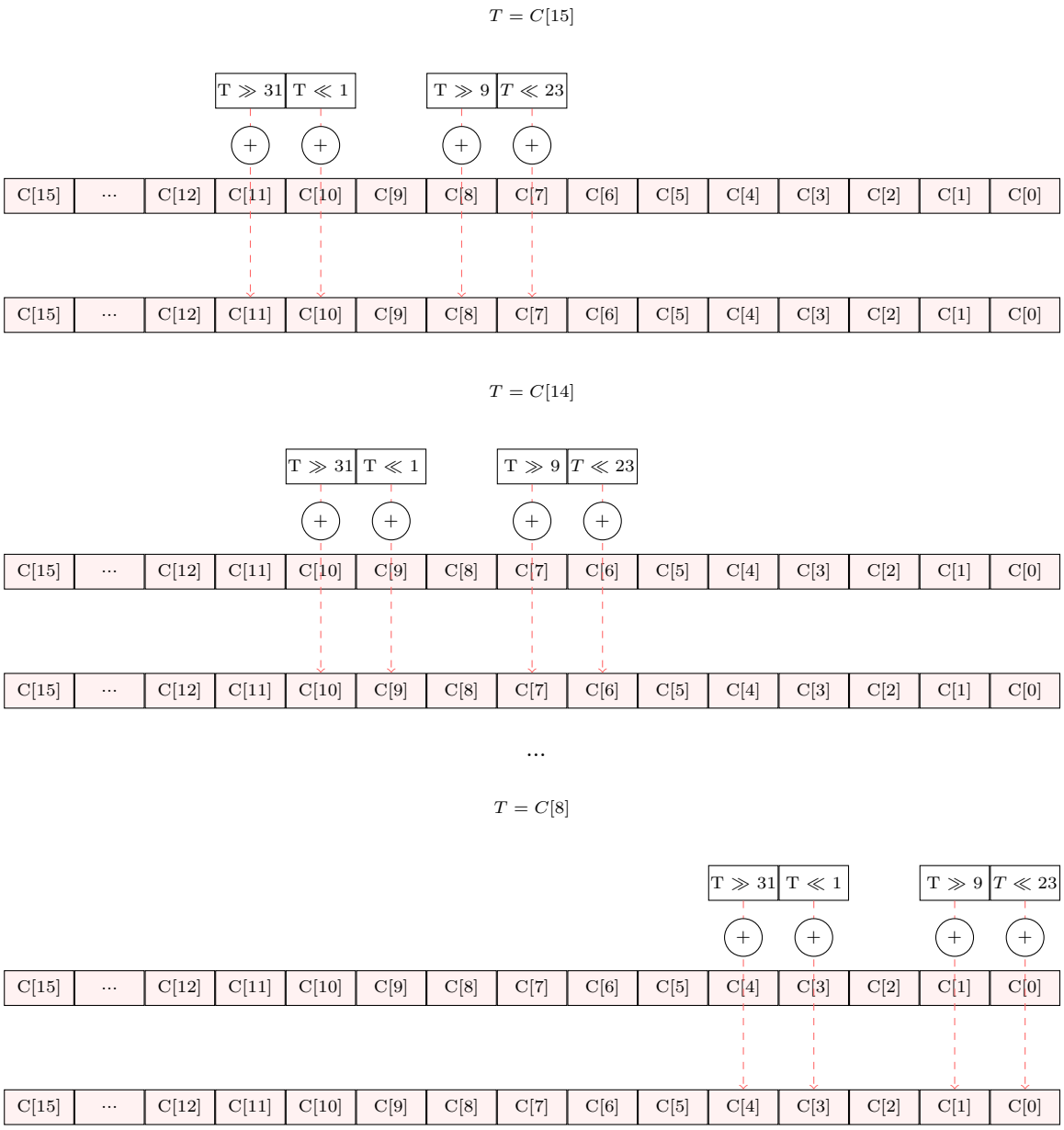

Figure 4.16: Graphical view of the fast reduction Algorithm 13.

In Figure 4.16, it is shown a graphical view of Algorithm 13. The fast reduction algorithm allows executing four digits computations in parallel, for example when $T = C[15]$ it is possible to compute values of $C[11]$, $C[10]$, $C[8]$ and $C[9]$ which is very attractive if all the required digits are available. However, in the digit-digit approach that is proposed in this thesis, the fast reduction algorithm requires a significant amount of memory readings and writings. For example, one iteration of for loop (line 1) in Algorithm 13 requires one clock cycle for reading each of the values $T, C[i-8], C[i-7], C[i-5]$ and $C[i-4]$. Furthermore, one extra clock cycle for writing each of the computed values $C[i-8], C[i-7], C[i-5]$ and $C[i-4]$ is required. A total of 9 clock cycles at each iteration of the for loop are required. If an operand size of 233 bits with digit size of 32 bits are used, then the for loop in Algorithm 13 requires $9 * 8 = 72$ clock cycles in a digit-digit computation. However, the proposed hardware architecture for multiplication in $\mathbb{F}_{2^m}$ with the same operand and digit sizes (233, 32) requires $8 * 9 = 72$ clock cycles too. So, for the aims of this research, it is possible to use the same multiplier as the square operator, since the implementation of the fast reduction algorithm requires additional logic that can be saved reusing the same multiplier.

At each iteration of Algorithm 9, there are required to compute a point addition (Madd) and a point double (Mdouble) operations. So, according to Figure 4.15 it is possible to compute Madd and Mdouble operations in a parallel way since there is no data dependency. However, two $\mathbb{F}_{2^m}$ multipliers are required. Furthermore, it is possible to compute Madd and Mdouble operations with only one adder. Madd operation requires 5 field multiplications while Mdouble requires 6. Mdouble only requires the adder after 6 field multiplications, at this time, Madd operation will finish leaving the adder free to be used for Mdouble operation. The general view of the ECC scalar multiplier is shown in Figure 4.17.

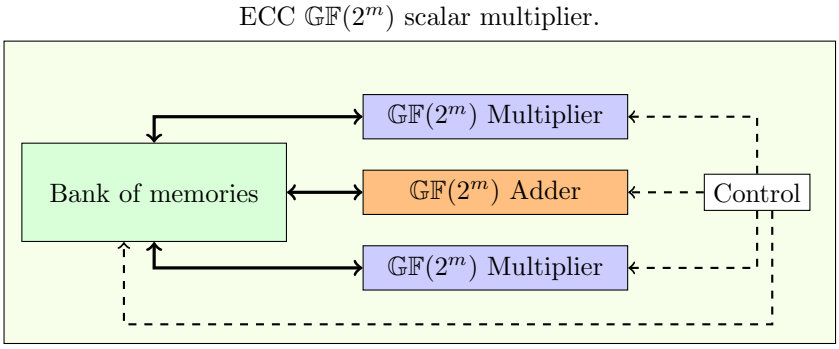

Figure 4.17: General view of the ECC scalar multiplier architecture (Algorithm 9).

4.7 Summary

In this section, the design and implementation of the proposed finite field operators were presented. The main aim of the operators was to reduce the hardware resources for hardware implementations. All proposed operators use the digit-digit approach to reduce hardware resources. A Montgomery multiplier was developed with the digit-digit approach, and then a Montgomery Powering Ladder $\mathbb{F}_p$ exponentiator was implemented with two Montgomery multipliers that work in parallel. A $\mathbb{F}_{2^m}$ multiplier based in the MSE algorithm was designed using the digit-digit approach. Three versions of $\mathbb{F}_{2^m}$ multipliers were developed: a direct implementation of the proposed digit-digit algorithm, a Karatsuba version, and the last one require only one partial multiplier in finite fields generated by trinomials. This multiplier was used to implement a scalar multiplier over $\mathbb{E}(\mathbb{F}_{2^m})$ in the context of elliptic curves, in which only two digit-digit multipliers and one digit-digit adder are required. In the next section, there are presented the implementation results in FPGAs of the proposed finite field operators presented in this section.

Chapter 5

Implementation and Results

This chapter presents results obtained in this research. First, it presents tools used for experimentation and evaluation metrics. Next, it describes digit and operands size used in the proposed operators. Following, results of proposed arithmetic operators are presented: operators in fields $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$; and operators in groups $\mathbb{F}_p^*$ and $\mathbb{E}(\mathbb{F}_{2^m})$. After, comparison with state of the art works is presented. Finally, the chapter presents the design of a hardware/software co-design that was developed for in-circuit verification of all designs proposed in this dissertation. Hw/sw co-design implements the Elliptic Curve Diffie-Hellman (ECDH) key exchange and RSA digital signature protocols that make use of the proposed field and group operators, which validates and give evidence of the practical use of the operators developed in this work.

5.1 Tools and evaluation metrics

In this research work, all proposed architectures for field and group operators were described in VHDL. However, software implementations were done for test vectors generation used in the verification phase. Test vectors were used in both simulation and in-circuit verification.

The design flow used in this work and shown in Figure 1.7 was automated, allowing to produce different hardware architectures under different configurations (operand size, digit size, optimization criteria, etc).

FPGAs were used as computing platforms to prototype all hardware architectures developed in this doctoral research. These devices permit to explore area/performance trade-offs as well as to evaluate designs and compare them against related works in the literature. FPGAs also allow exploring trade-offs between area and processing speed. The software tools used to assist all design, implementation and validation phases are:

- Simulation: Model Technology ModelSim ALTERA STARTER EDITION vsim

10.3d Simulator 2014.10 Oct 7 2014.

- Synthesis and implementation: Vivado v2016.1 (64-bit) and ISE 14.7 from Xilinx.
- FPGA platform:
  - Virtex 7 and Virtex 5 from Xilinx.
  - MicroZed low-cost development board based on the Xilinx Zynq®-7000 All Programmable SoC.
- Test vectors:
  - $\mathbb{F}_p$: test vectors created with the BigInteger library from Java (JDK 8u181).
  - $\mathbb{F}_{2^m}$ and $\mathbb{E}(\mathbb{F}_{2^m})$: test vectors created with the Number Theory Library (NTL) for C++ (NTL 9.7.0).
- Software implementations of the RSA Digital Signature and ECDH: implemented in software in the MicroZed Board under Linux with the MIRALC library for comparisons.

The primary metric to evaluate the compactness of the architectures is the number of slices. This allows comparing the size of two architectures. Another metric of interest is the throughput, that measures the number of bits that a hardware architecture process for time unit (see its definition in Equation 5.2). This metric allows comparing speed/performance of architectures. The last metric used in this work is efficiency, that shows the number of bits processed by area unit (slice). The efficiency is calculated with equation 5.1.

$$\mathrm{efficiency} = \frac{\text{throughput}}{\text{slices}}(\mathrm{bps/slice}) \tag{5.1}$$

$$\mathrm{throughput} = \frac{\text{Frecuency} \times \text{num. bits}}{\text{num. clock cycles}}(\mathrm{bits/s}) \tag{5.2}$$

The proposed hardware architectures were implemented in different FPGAs. Virtex 5 and Virtex 7 were used to compare the proposed hardware architectures with most of the state of the art works. The MicroZed board (a low-cost development board based on the Xilinx Zynq®-7000 All Programmable SoC) was used to implement arithmetic operators and hw/sw co-design. The MicroZed board is a state of the art FPGA commonly used in industrial IoT applications. The Zynq-7000 system integrates the software programmability of an ARM®-based processor (Cortex9) with

the hardware programmability of an FPGA Artix 7, which enable the flexibility of software application with the advantage of the high speed custom hardware design in the FPGA Artix 7.

5.2 Area performance trade-off approach

In this work, it is followed one of the approaches in the literature when implementing hardware architectures in FPGAs, the use of embedded IP cores such as DSP and BRAM modules. This is generally done to reduce the amount of standard logic of the FPGA, leaving more resources to implement other parts of the security protocol or the application. Also, this implementation approach allows incrementing the operational frequency and thus improving execution time and throughput.

Design and implementation of cryptography hardware architectures in FPGAs depend on the efficient use of architectural features provided in the targeted FPGA. The Xilinx FPGAs used in this work have embedded cores DSPs and BRAMs which have been employed to reduce the standard logic usage of the proposed designs. Similar building blocks can also be found in other Xilinx FPGA families such as in the Virtex, Spartan, Kintex, Artix, etc, as well as in the Stratix II and Cyclone II devices of Intel's FPGAs. So, the proposed implementation technique can be adapted to other FPGAs with similar features.

5.3 Compact FPGA operators in $\mathbb{F}_p$

The implementation results for the Montgomery multiplier (Figure 4.6) in the Virtex-7 FPGA are shown in Figure 5.1. The scalability of the proposed multiplier is confirmed with the area results shown in Figure 5.1a, where it is observed that the size of the operands do not greatly affect the number of slices as the digits do. The best configurations in terms of area usage are for $\{k = 8, s = 256\}$, $\{k = 4, s = 512\}$ and $\{k = 8, s = 1024\}$. When $k > 16$, the area increases considerably, possibly due to the interconnections between the Configurable Logic Block (CLB)s.

The operands size also does not affect the multiplier frequency, but the digit size does, as it is shown in Figure 5.1b. This mainly happens because of multipliers complexity and other components in the datapath increase as the digits get bigger, which also increase the critical path in the circuit. A larger operand size will require more digits to process, increasing the latency but not affecting the word size in the datapath or the complexity of internal hardware modules (adders and multipliers). However, if a greater digit size is used, latency is reduced. This reduction comes with

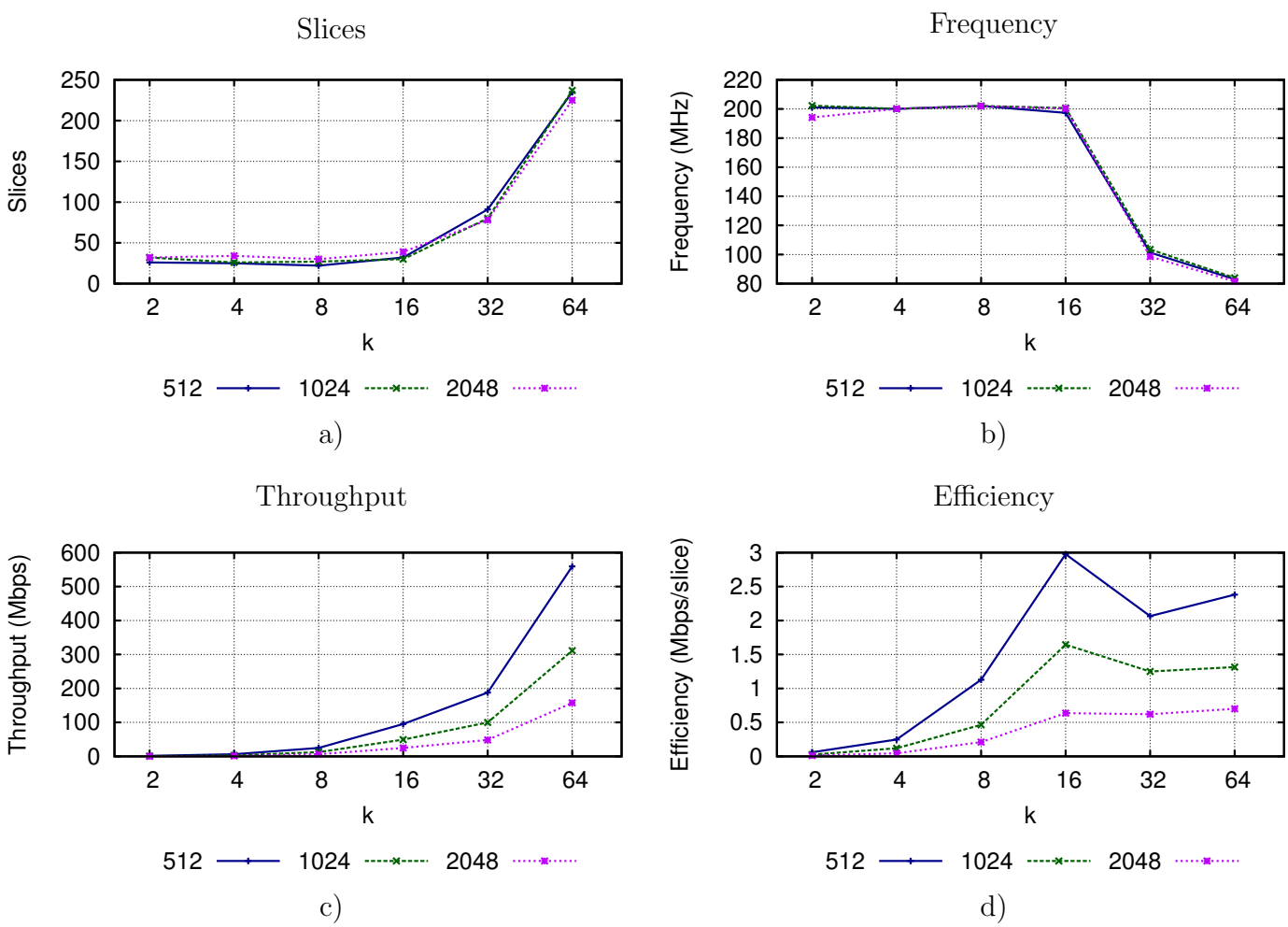

Figure 5.1: Implementation results of the Montgomery multiplier (Figure 4.6) in the Virtex-7 FPGA.

an increasing in the throughput, as Figure 5.1c reveals. The best result is obtained for $k = 64$, with a throughput of 311.48 Mbps for an operand size of 1024. Figure 5.1d reveals that the best efficient $\mathbb{F}_p$ multiplier operator obtained is with a digit size $k = 16$ for an operand size of 512 or 1024 bits. When the operand size is 2048 bits, the most efficient multiplier is the one using $k = 64$.

Microprogramming approach

The compact $\mathbb{F}_p$ multiplier was implemented with the proposed microprogramming approach proposed in Section 4.2. In this approach, datapath and control logic (FSM and microprogramming) modules of the $\mathbb{F}_p$ multiplier were independently implemented obtaining the results shown in Tables 5.1 and 5.2, respectively.

Table 5.1: Implementation results for the datapath block (1024 bits operand size).

Digit size	Slices	BRams	DSP	Freq. (MHz)
16	16	0	4	232.61
32	66	0	11	135.64
64	230	0	33	97.66

Table 5.2: Implementation results for the control block, using the microprogramming and FSM approaches (1024 bits operand size).

Digit size	Slices	BRams	DSP	Freq. (MHz)
16	8	0	0	447.42
32	9	0	0	490.67
64	9	0	0	397.93
16 (FSM)	34	0	0	227.58
32 (FSM)	20	0	0	339.21
64 (FSM)	15	0	0	398.40

Most of the logic in the datapath are multipliers and adders which are implemented by embedded DSP modules. Contrary to the datapath, the hardware resources for the control unit decrease with bigger digits in the FSM approach, but in the microprogramming approach, the resources do not vary. Table 5.2 shows the advantage of using a microprogramming approach. Since all the control signals are stored in a memory block, it is not required to generate the control signals; they only need to be read from the memory block which significantly increases the maximum frequency of operation.

Variable digit-digit $\mathbb{F}_p$ multiplier.

In the digit-digit $\mathbb{F}_p$ multiplier with the same digit size for all the operands and partial results, the required hardware resources in the FPGAs grow very fast when bigger digit sizes are used. For example, a $\mathbb{F}_p$ multiplier with 1024 bits operand size with a digit size of 32 bits require 80 slices, and with the same operand size but a digit size of 64 need 237 slices. This experiment tries to fit the gap between two digit sizes using a different digit size for the operands. In this section it is presented the obtained results for the $\mathbb{F}_p$ multiplier with a variable digit-digit approach presented in Section 4.2.1 Figure 5.2 shows implementation results of the $\mathbb{F}_p$ multiplication architecture with different digit sizes for the operands X, Y and P. Operand size of 1024 bits was used in this experiments. Digit sizes expressed in the form x-y indicates that x is the digit size

of the operand X and y the one of the operand Y:

- 4-4, 4-8, 4-16, 4-32, 4-64.
- 8-8, 8-16, 8-32, 8-64.
- 16-16, 16-32, 16-64.
- 32-32, 32-64.

For digit sizes: $\{4, 8, 16, 32\}$ the best results were obtained when using a multiplier with the same digit size. However, for 32-64 configuration it is possible fitting the gap between 32-32 and 64-64 results with an efficiency similar to 64-64. The $\mathbb{F}_p$ multiplier requires 80 and 237 slices with a digit size of 32 and 64 bits respectively. 32-64 configuration requires 137 slices which is not possible to obtain using the same digit sizes.

These results give evidence that using variable digit architectures can be attractive when the hardware resources disposal is not enough for a digit-digit size $k1$ but they are excessive for a digit size $k2$. For example, a hardware designer that have at his disposal 150 slices could not implement a 1024 $\mathbb{F}_p$ multiplier with a 64-64 digit configuration since that would requires 237 slices. However, if it is implemented with a 32-32 digit size configuration it is just only requires 80 slices, and almost 70 slices will be wasted in nothing. If the designer had at his disposal the variable digit-digit version of the multiplier, he could use the 32-64 configuration that requires 137 slices only with a throughput of 176 Mbps instead of the 99 Mbps achieved with the 32-32 configuration.

5.4 Compact FPGA exponentiation in $\mathbb{F}_p^*$

MPL exponentiator results

The implementation results for the Montgomery Powering Ladder (MPL) architecture are shown in Figure 5.3. It can be observed that the complexity of the MPL architecture strongly depends on the underlying $\mathbb{F}_p$ multiplier. For digit sizes from 2 to 16, the area resources remain less than 110 slices. However, the amount of area resources increases considerably when $k = 32$ and $k = 64$. In the same way, the clock frequency remains over 180 MHz when $k \leqslant 16$ but degrades considerably when $k = 32$ and $k = 64$, in consequence of greater delays due to using larger area. Throughput is considerably reduced, in the order of Kbps, achieving its best for greater digit sizes. In

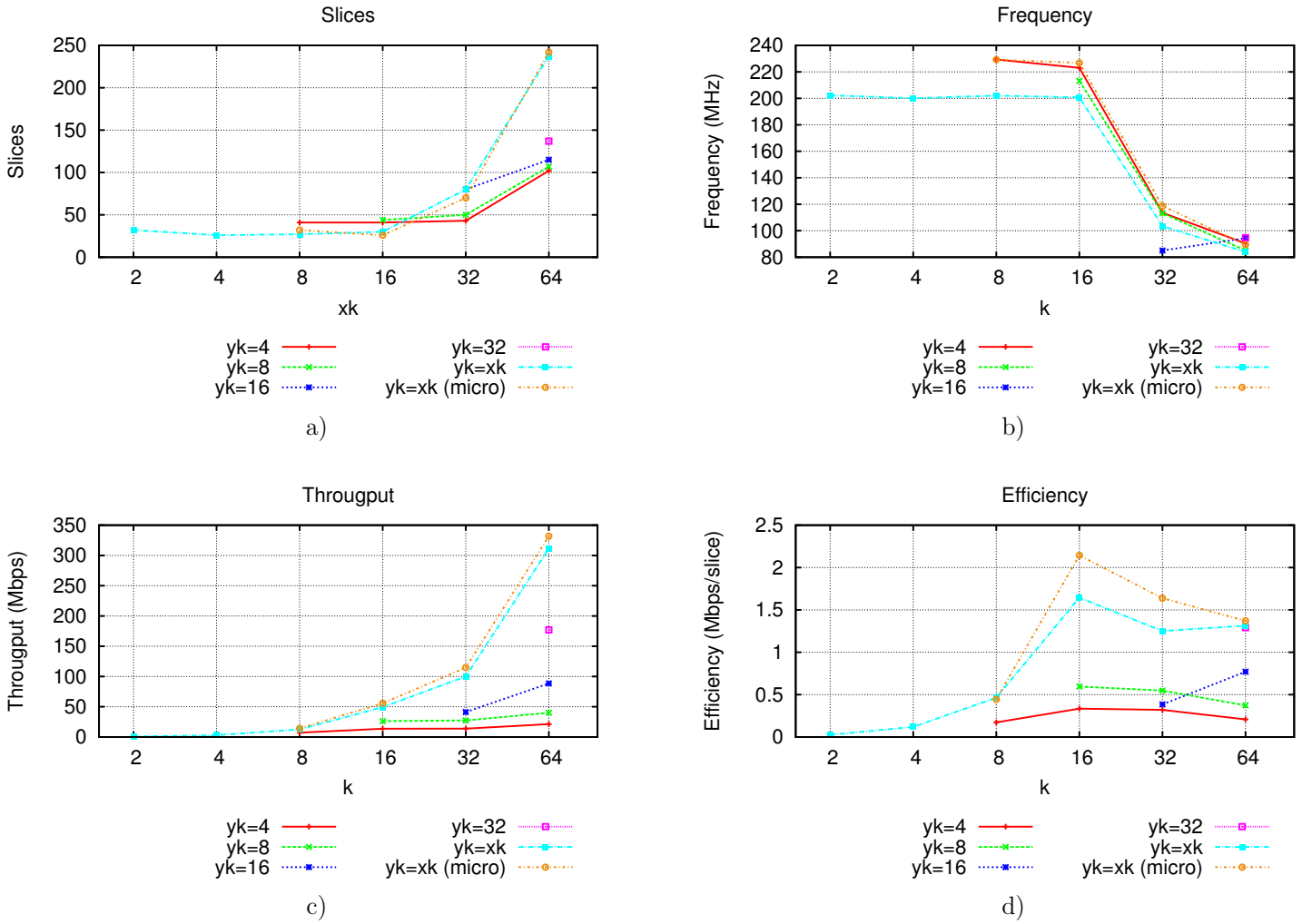

Figure 5.2: Implementation results of the $\mathbb{F}_p$ multiplier with different digit sizes (figure 4.8) in the Virtex-7 FPGA for 1024 bits operand size.

terms of efficiency, considerably better implementations are obtained for greater digit sizes: the best results are for $k \geqslant 16$. When $k \leqslant 16$ the partial multiplications fit in a single DSP module, but when $k > 16$ partial multiplications in the datapath require several interconnected DSP modules, which increases the number of slices required for interconnection and decreases the frequency.

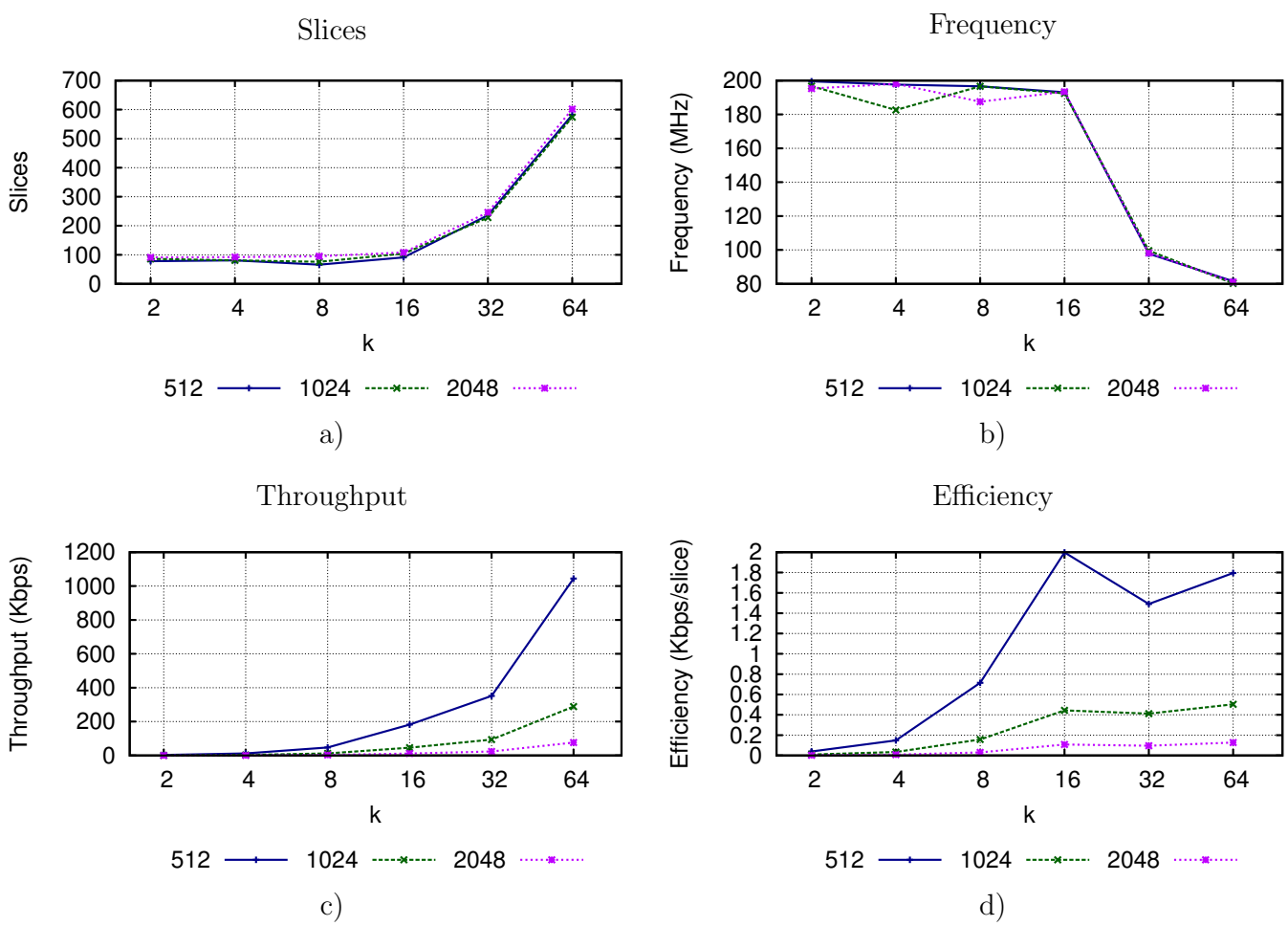

Figure 5.3: Implementation results for the $\mathbb{F}_p^*$ exponentiator (MPL) architecture for a Virtex-7 Xilinx FPGA.

5.5 Compact FPGA operators in $\mathbb{F}_{2^m}$

Figure 5.4 shows the place and route results for the three $\mathbb{F}_{2^m}$ multiplier architectures for the Virtex 7 Xilinx FPGA. These results consider the datapath and control module. Memory blocks are not part of the architectures and thus not considered in the results.

The hardware resources used by architecture v1 and architecture v2 are very similar for smaller digit size (2, 8, 16), but for digit sizes of 32 and 64, architecture v2 required few hardware resources. For example:

- Operand size: 233
  - Digit size: 32
    - v1 : 408 slices.
    - v2 : 363 slices.
  - Digit size: 64
    - v1 : 1520 slices.
    - v2 : 1343 slices.
- Operand size: 409
  - Digit size: 32
    - v1: 413 slices.
    - v2: 371 slices.
  - Digit size: 64
    - v1: 1417 slices.
    - v2: 1061 slices.

Are reduction is due to the implementation of the Karatsuba partial multipliers. For small digit sizes (2, 8, 16) the saved standard logic due to the Karatsuba approach is not noticeable because interconnections between partial multipliers have a cost similar to the one saved. However, for digits size of 32 and 64 the saved logic due to the Karatsuba approach is higher than the one used for interconnecting the partial multiplier, so it is reflected in the final area resources shown in Figure 5.4.

In the hardware architecture v3 it is eliminated one partial multiplier (see Figure 4.13), which lead to an area reduction compared to architectures v1 and v2. Furthermore, the frequency is considerably better in architecture v3, mainly for digit size of 32 and 64. This is due to the elimination of one partial multiplier too. As a consequence, architecture v3 achieve a higher Throughput and Efficiency than architectures v1 and v2.

The obtained results are shown in Figure 5.4 for operand size of 233 and 409. Area results for the implementation of the bit serial multipliers are very low, and very similar to the results obtained for the proposed digit-digit multiplier for digit sizes of 8

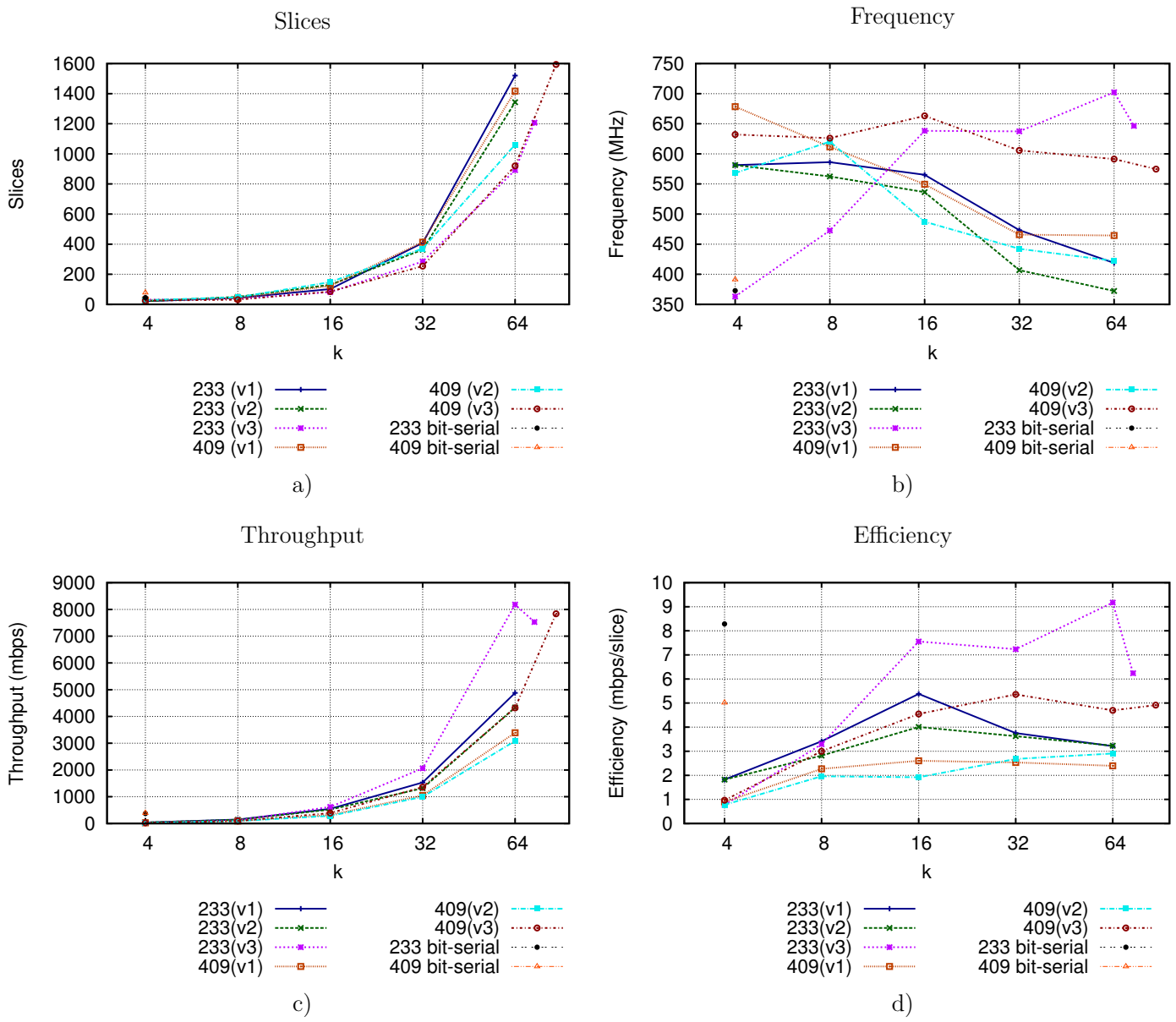

Figure 5.4: Implementation results for the $\mathbb{F}_{2^m}$ multiplier architecture in the Virtex-7 Xilinx FPGA.

and 16. However, storage resource for operands and final results are not considered in the hardware designs. For the bit serial approach, there will be required some registers to store operands and partial results while for the proposed hardware architecture there will be necessary BRAMs which not consume standard logic. So, the saved slices will be reflected in higher layers of the operands such as the ECC point multiplication. The efficiency for the proposed hardware architecture is very similar than the obtained in the bit-serial approach.

5.6 Compact FPGA $\mathbb{E}(\mathbb{F}_{2^m})$ exponentiator

This section presents the obtained results for the $\mathbb{E}(\mathbb{F}_{2^m})$ exponentiator. The proposed $\mathbb{E}(\mathbb{F}_{2^m})$ exponentiator uses the $\mathbb{F}_{2^m}$ multiplier v3 presented in section 4.5.3 which takes advantage of the binary finite fields generated by irreducible trinomials such as the NIST recommended polynomials 233 and 409, see Table 4.3.

For a fair comparison, one of the smaller bit serial $\mathbb{F}_{2^m}$ exponentiator [91] was implemented. The bit-serial $\mathbb{F}_{2^m}$ exponentiator [91] takes advantage of the bit serial Most Significant Bit approach for the field multiplication $\mathbb{F}_{2^m}$, the sum operation is achieved by the xor operand, the square operation in $\mathbb{F}_{2^m}$ is performed in two steps, first an expansion with interleaved 0's, then reducing the double-sized result with the reduction polynomial, and the inversion operation is performed with the Itoh-Tsujii algorithm [92].

Figure 5.5 shows the implementation results for the proposed $\mathbb{E}(\mathbb{F}_{2^m})$ exponentiator (scalar multiplier) for the elliptic curves k-233 and k-409 recommended by NIST. Furthermore, it shows the implementation results for the $\mathbb{E}(\mathbb{F}_{2^m})$ bit serial reimplementation proposed in [91] for the Elliptic Curve k-233 recommended by NIST. Figure 5.5a shows the are results for the proposed hardware architecture and the bit serial implementation. It is observed that the proposed approach requires a smaller amount of standard logic (slices) in the FPGAs for digit size 4, 8, and 16. For digit size 32 and 64 the required slices are higher than the compact $\mathbb{E}(\mathbb{F}_{2^m})$ bit-serial implementation. According to Figure 5.5b the $\mathbb{E}(\mathbb{F}_{2^m})$ bit-serial approach achieves an operation frequency of almost 200 MHz, and the maximum frequency in the $\mathbb{E}(\mathbb{F}_{2^m})$ digit-digit approach depends on the digit size. For example, for a digit size of 4 bits is higher than 200 MHz, but as the size of the digit increases the frequency of operation will decrease, for a digit size of 32 the maximum frequency achieved is 115 MHz for operand size of 233 and 112 MHz for an operand size of 409 bits. The $\mathbb{F}_{2^m}$ partial multipliers mainly determine the maximum frequency. For the $\mathbb{E}(\mathbb{F}_{2^m})$ bit-serial approach only and gates

are required, so higher frequency is achieved. In the digit-digit approach, the partial multipliers of digits determine the maximum frequency, and as the digit size grows the partial multipliers require a more significant period to generate the partial result.

The throughput achieved with the $\mathbb{E}(\mathbb{F}_{2^m})$ bit-serial approach is of 56 kbps. For the proposed $\mathbb{E}(\mathbb{F}_{2^m})$ digit-digit approach the digit size 4 and 8 achieve a throughput of 7 and 28 kbps respectively. However, the proposed $\mathbb{E}(\mathbb{F}_{2^m})$ digit-digit with digit size 16 achieves a throughput of 85 kbps, higher than the $\mathbb{E}(\mathbb{F}_{2^m})$ bit-serial approach. Figure 5.5 shows the efficiency results of the proposed hardware architectures. The $\mathbb{E}(\mathbb{F}_{2^m})$ bit-serial implementation achieves an efficiency of 0.069 kbps/slice very similar than the proposed $\mathbb{E}(\mathbb{F}_{2^m})$ digit-digit implementation with a digit size of 8 bits (0.064 kbps), but requires fewer hardware resources (slices). However, the proposed $\mathbb{E}(\mathbb{F}_{2^m})$ digit-digit implementation with a digit size of 16 achieves higher efficiency than the $\mathbb{E}(\mathbb{F}_{2^m})$ bit serial, and requires fewer hardware resources. The $\mathbb{E}(\mathbb{F}_{2^m})$ bit-serial architecture requires 862 slices and achieve an efficiency of 0.069 kbps, while the digit-digit hardware architecture with a digit size of 16 needs 626 slices and achieves an efficiency of 0.132 kbps, almost the double than the bit-serial approach with fewer hardware resources.

In this section, there have been presented implementation results for all the proposed finite field operators. In the next section, a comparison with state of the art is presented.

5.7 Comparisons

This section presents a comparison of the proposed finite field operators with the state of the art works for FPGAs platforms. Firstly, in some cases it is very complex to provide a fair comparison since there are works that report their results in different platforms, or with distinct parameters than the one used in this research. However, the provided comparison is as fairly as possible according to the platform device and parameters used.

The obtained results shown in previous sections for finite field multipliers over $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$, and for group operations over $\mathbb{F}_p^*$ and elliptic curves $\mathbb{E}(\mathbb{F}_{2^m})$ are compared with state of the art works in the literature. In the case of elliptic curves scalar multiplication, a bit serial approach has been implemented and evaluated to fairly compare the proposed digit-digit finite field operator with a well-known bit-serial implementation in the same device and under the same conditions.

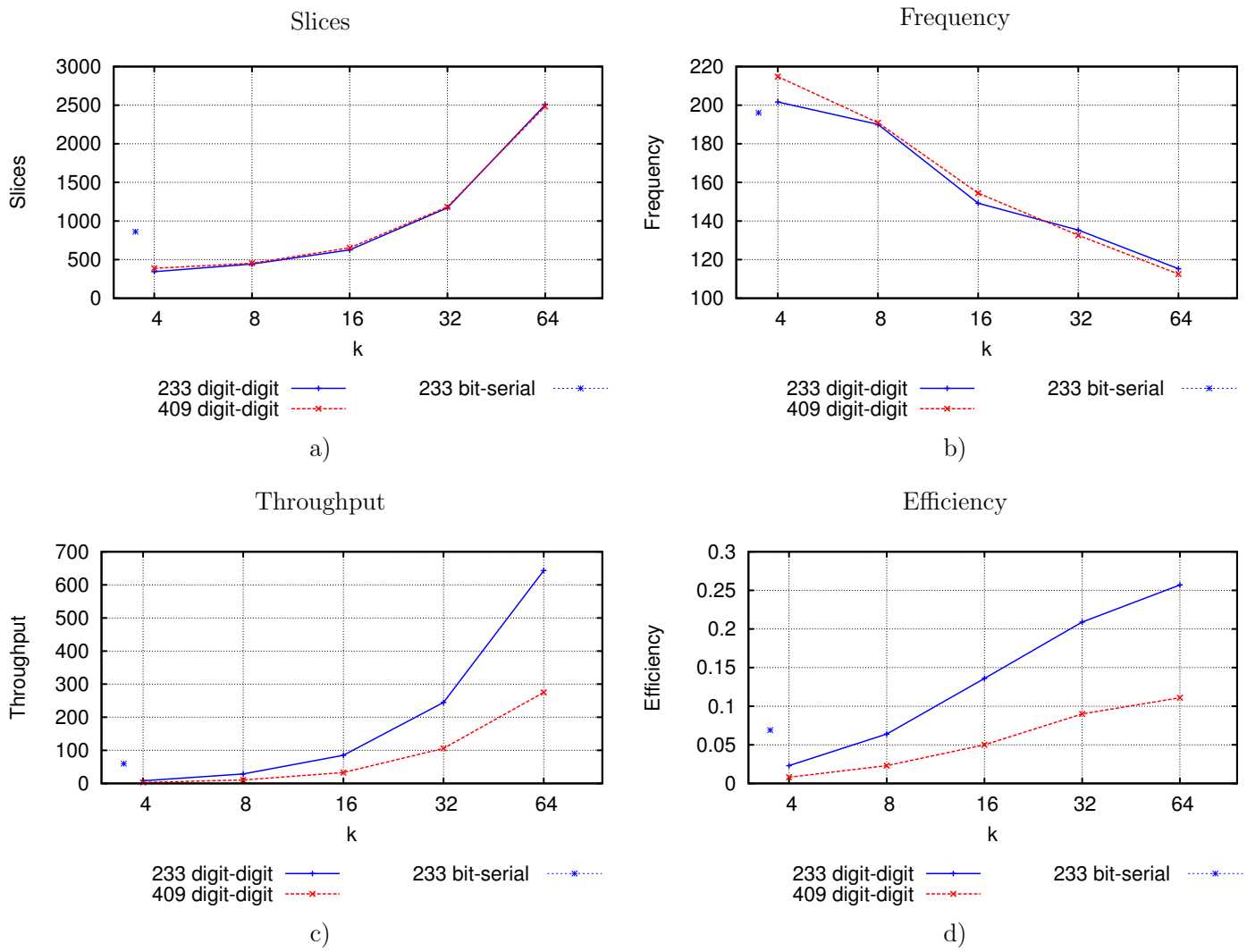

Figure 5.5: Implementation results for the $\mathbb{E}(\mathbb{F}_{2^m})$ scalar multiplier in the Zynq-7000 Xilinx FPGA.

5.7.1 $\mathbb{F}_p$ multiplication

This section presents a comparison for the proposed $\mathbb{F}_p$ multiplier. Table 5.3 shows some of the most representative works for multiplication over $\mathbb{F}_p$. Most of the works present their results in different technologies, for example, Xilinx FPGAs: Virtex 2, Virtex 5, or the Spartan Family. However, this thesis research project tries to make the comparison as fair as possible.

Table 5.3: Implementaciones compactas del algoritmo Montgomery.

Work	bits	FPGA	slices	Frec. (MHz)	Throughput (Mbps)	Efficiencia (Mbps/slices)
prop. (Micro)	1024 (k=16)	Virtex 7	**26**	226.75	55.76	**2.145**
prop. (Micro)	1024 (k=32)	Virtex 7	**70**	118.72	114.96	**1.638**
prop. (FSM)	1024 (k=16)	Virtex 7	**30**	200.68	49.35	**1.645**
prop. (FSM)	1024 (k=32)	Virtex 7	80	103.49	99.98	1.250
prop. (FSM)	2048 (k=16)	Virtex 7	39	200.40	24.85	0.637
[57]	1024 (D-4)	Virtex 5	5702	222.22	868.50	0.152
[57]	2048 (D-2)	Virtex 5	6837	390.62	777.3	0.113
[3]	1024 (k=16)	xc5vlx50	**94**	102.42	24.45	0.260
[59]	1020	Spartan-3E500	1553	119	133.80	0.086
[93]	1024 (k=64)	Virtex 5	219	237.52	242.66	**1.108**
[94]	1024	Virtex II V2-600	3390	121	117.00	0.035
[58]	1024	XC2V3000	4512	114.2	113.40	0.025

In [57] it is presented a Montgomery multiplier that takes advantage of the Carry Save Adders (CSA) technique to improve the critical datapath in the partial operations. Furthermore, this multiplier uses a digit-serial approach to an area-speed trade-off. Different digit sizes were analyzed in [57] with the Virtex 2 and Virtex 5 FPGAs. When using the Virtex 5 FPGA, the multiplier presented in [57] achieved a high throughput (868 Mbps) at the cost of high area resources (5702), 219 times bigger than the multiplier proposed in this research. Furthermore, in this research, an efficiency of 2.145 Mbps/slice is achieved 14 times better than the multiplier presented in [57] (0.152 Mbps) for operand size of 1024. For a 2048 bit operand size, the $\mathbb{F}_p$ multiplier presented in [57] requires 6837 slices achieving an efficiency of 0.113 Mbps, the $\mathbb{F}_p$ multiplier proposed in this research is five times more efficient than the proposed in [57] and require only 39 slices.

In [59] is presented a RSA implementation, which require a $\mathbb{F}_p$ exponentiation and a $\mathbb{F}_p$ multiplier. For the $\mathbb{F}_p$ multiplication, it is used the Montgomery algorithm, specifically, the CIOS versions proposed in [64] using the Spartan 3 FPGA as implementation platform, the Montgomery multiplier presented in [59] require 1553 slices.

In [93] it is presented a digit-digit multiplier with a similar approach that this thesis.

However, due to the microprogramming technique and the pipeline registers used, the proposed hardware architecture outperforms the results obtained in [93] regarding area and efficiency. For example, the Montgomery multiplier presented in [93] require 2019 slices for a digit size of 64, and achieve an efficiency of 1.108 Mbps/slice.

The works presented in [94, 58] are reported in an old Xilinx FPGA the Virtex II, that a direct comparison with state of the art FPGAs (Virtex 5, Virtex 7 or Zynq 7000) is unfairly. It is easy to see that the proposed architectures in this thesis research achieve a better efficiency with an small amount of hardware resources than the works reported in [94, 58]. However this works are presented in Table 5.3 as a basis reference.

According to Table 5.3 the proposed Montgomery multiplier is one of the most compact finite field hardware operators with an efficiency similar to or better than others reported in the literature. This Montgomery multiplier has been used in this thesis research as a main component for $\mathbb{F}_p$ exponentiation.

5.7.2 $\mathbb{F}_p$ exponentiation

In this section, a comparison of state of the art $\mathbb{F}_p$ hardware exponentiation (Montgomery Powering Ladder (MPL), Most Significant Bit (MSB), Least Significant Bit (LSB)) is presented. Table 5.4 shows some of the most significant state of the art works for exponentiation in $\mathbb{F}_p$. It should be noted that a fair comparison is difficult due to the different technologies and implementation strategies used. It is not possible to compare all the works with the same metric since not all the designs exploit the FPGAs embedded blocks. However a comparison as fair as possible according to area (slices), throughput (kbps) and efficiency (kbps/slice) is presented.

It is remarked here the importance of using the embedded FPGA resources, mainly for efficiency improvement and power saving [95]. The comparison shown in Table 5.4 is in terms of the standard logic (slices) since the goal of the proposed design is compactness. Although a fair comparison against [57, 69, 66] is not possible using slices as a metric, it can be done in terms throughput and efficiency. Since [62, 59] also use FPGA embedded resources, a fairer comparison against those works is possible.

The hardware module for MPL implemented in [59] uses the CIOS Montgomery algorithm as $\mathbb{F}_p$ multiplier. The number of slices is 3899 plus 16 BRAMs, completing an exponentiation in 7.95 ms in a Spartan 3E. Compared to [59], using the same FPGA and operand size of 1024, our design with $k = 16$ is more compact (one-tenth the size), occupying only 375 slices. For $k = 32$, our design still remains with a lower area (one-fourth the size), using 900 slices. In terms of efficiency, our design is also better than [59], improving the efficiency by 48% (with $k = 16$) and 72% (with $k = 32$).

Table 5.4: Results and comparison for a 1024-bit exponentiation.

Work	Alg.	Op.Size (bits)	FPGA	Area (slices)	BRAMs	DSPs	Freq (MHz)	avg Cyc (x 1000)	avg T (ms)	Thrg (Kbps)	Efficiency (kbps/slice)
our.(k=16)	MPL	1024	Z-7010	**109**	3	6	106.38	4265	40.10	25.535	0.234
our.(k=32)	MPL	1024	Z-7010	**249**	5	22	68.49	1087	15.76	64.49	0.258
[59]	MPL	1024	Spartan3E	3899	16	20	119.05	946	7.95	128.84	0.033
our.(k=16)	MPL	1024	Spartan3E	**375**	6	6	77.16	4265	55.29	18.521	**0.049**
our.(k=32)	MPL	1024	Spartan3E	**900**	6	22	54.59	1087	19.93	51.387	**0.057**
[57](k=2)	MSB	1024	Virtex-5	7303	-	-	384.62	529	1.38	744.60	0.102
[57](k=4)	LSB	1024	Virtex-5	6217	-	-	222.11	397	1.79	572.50	0.092
[57](k=2)	LSB	1024	Virtex-5	4060	-	-	384.62	793	2.03	503.60	0.124
[69]	MPL	1024	Virtex-5	3218	-	-	346.02	1097	3.18	322.01	0.100
[66]	LSB	1024	Virtex-5	6776	-	-	401	-	1.37	747.4	0.110
[66]	MSB	1024	Virtex-5	12716	-	-	401	-	0.92	1113	0.087
our(k=16)	MPL	1024	Virtex-5	**160**	6	8	190.84	4265	22.35	45.809	**0.286**
our(k=32)	MPL	1024	Virtex-5	**266**	6	22	73.91	1087	14.71	69.605	**0.262**
[62](k=16)	LSB	512	Virtex-7	343	-	14	458	-	1.23	416.26	1.214
our(k=16)	MPL	512	Virtex-7	**91**	6	8	193.12	543	2.82	181.85	**1.998**
[62](k=32)	LSB	1024	Virtex-7	1060	-	26	485	-	2.33	439.48	0.415
our(k=64)	MPL	1024	Virtex-7	**574**	10	66	80.21	284	3.55	288.55	**0.503**
[62](k=64)	LSB	2048	Virtex-7	3558	-	54	399	-	5.68	360.56	0.101
our(k=64)	MPL	2048	Virtex-7	**602**	10	66	81.11	2174	26.82	76.37	**0.127**

The results reported in [57] are among the fastest in the literature, but the FPGA area resources (Virtex-5) consumed are too high, 4060 slices, with an execution time of 2.03 ms. Our design is more efficient than the MPL hardware module reported in [57]. For 1024-bit operands, our design with $k = 16$ has an efficiency of 0.286 kbps/slice twice the one achieved by the best version reported in [57].

The hardware module for $\mathbb{F}_p$ exponentiation reported in [69] for a Virtex-5 FPGA uses 3218 slices, with a throughput of 322.01 kbps and an efficiency of 0.100 kbps/slice. Our design with $k = 16$ uses only 10% of the resources reported in [69] with a better efficiency of 0.286 kbps/slice (more than double).

Our results with the Virtex-5 FPGA can be compared with those of [66]. The best efficiency reported in [66] is 0.110 kbps/slice using an area of 6776 slices. In contrast, our proposed architecture for the same device achieves an efficiency of 0.286 kbps/slice using only 160 slices.

One of the most compact modular exponentiation architecture for FPGAs reported to date is the one presented in [62] for a Xilinx FPGA, using the binary algorithm for $\mathbb{F}_p$ exponentiation and Montgomery and Karatsuba algorithms for field multiplication. Our design outperforms [62] in terms of efficiency, due to the significant savings in area resources. For a 1024-bit modulus, our design uses half the slices with a better

efficiency of 0.503 kbps/slice, and for a 2048-bit modulus, our design is one-sixth the size, as well as having a better efficiency: 0.127 kbps/slice. [62] exploits 17-bit multipliers and 48-bit adder units in DSP blocks to compute the multiplication of high radix integers. The smaller digit size used there is 16, which fits the embedded multipliers in the Xilinx FPGAs.

The results obtained show that the proposed $\mathbb{F}_p$ exponentiation architecture is smaller than the state of the art in terms of slices, while the number of DSPs and memory blocks required is similar to or less than other works reported in the literature.

Table 5.5 shows the power estimation generated with Xilinx Power Analyzer (XPA). Dynamic Powers refers to the quantity and specific use of each resource, and it is considered signals toggling and capacitive loads charging and discharging. So, designs with higher required resources, as well as designs with higher clock frequency will consume more power. Also, big digits require more hardware resources, and as a result, more power consumption. So, in low power devices, it is preferably smaller hardware architectures. On the other hand, quiescent power (also called static power) is not affected by the activity of the design. For example, in Table 5.5 quiescent power is the same for all configurations. When small digits are used, BRAMs consume most of the power. However, when bigger digits are used, signals and DSPs require similar power than BRAMs.

Table 5.5: Supply power (W) of the MPL architecture.

Size	k	Clocks	Logic	Signals	BRAMs	DSPs	IOs	Dynamic	Quiescent	Total
1024	8	0.005	0.003	0.008	0.021	0.006	0.007	0.049	0.178	0.227
1024	16	0.007	0.004	0.012	0.017	0.008	0.013	0.061	0.178	0.239
1024	64	0.006	0.015	0.032	0.036	0.023	0.021	0.132	0.178	0.311
2048	16	0.007	0.004	0.015	0.021	0.008	0.013	0.069	0.178	0.247
2048	64	0.006	0.014	0.029	0.036	0.023	0.021	0.128	0.178	0.307

Although a high throughput is not the aim of the exponentiation architecture proposed in the present thesis research, it is worth noting that the throughput achieved by our design is better than representative software implementations, as is shown in Table 5.6. For example, our proposed architecture in Virtex-7 is 600 times faster than the timing achieved in [96], which is aimed at Wireless Sensor Network (WSN) applications.

The MSP430 and ATmega128 are two processors commonly used for sensor network research. The proposed design in the Zynq-7010 is 190x faster than the MSP430 implementation, and 697x faster than the ATmega128 implementation. This comparison

Table 5.6: $\mathbb{F}_p$ exponentiation in software vs. proposed $\mathbb{F}_p$ exponentiation compact hardware architecture.

Ref.	Imp.	Time
[97]	MSP430 @ 8MHz	$\approx$ 3 s
[86]	ATmega128 8MHz	10.99 s
[36]	WSN Software	22.03 s
our(k=64)	Virtex-7	3.55 ms
our(k=32)	Virtex-5	14.71 ms
our(k=32)	Zynq-Z7010	15.76 ms

Table 5.7: State of the art works for finite field hardware operators over $\mathbb{F}_{2^m}$.

Work	FPGA	Size	Ciclos	Slices	FF	LUTS	Frecuencia	Throughput (mbps)	Eficiencia (mbps/slice)
Prop (v3,k=16)	s3	233	240	137	62	246	223.36	216.84	1.58
Prop (v3,k=32)	s3	233	72	423	94	816	241.13	780.35	1.84
Prop (v3,k=16)	s3	409	702	140	62	252	218.00	127.01	0.90
Prop (v3,k=32)	s3	409	702	434	107	818	241.89	543.60	1.25
Prop (v3,k=16)	v7	233	240	**82**	62	153	638.16	619.54	**7.55**
Prop (v3,k=16)	v7	409	702	**85**	62	170	663.13	386.35	**4.54**
[71](g=1)	v5	233	233	178(aprox.)	710	714	561.79	561.79	3.15
[71](g=16)	v5	233	15	587(aprox.)	705	2351	423.72	6581.78	**11.21**
[71](g=1)	v5	409	409	310(aprox.)	1240	1244	549.45	549.45	3.15
[71](g=1)	v5	571	571	432(aprox.)	1727	1731	540.54	540.54	1.25
[76](k=16)	v6	233	1643	410(aprox)	x	1643	338	47.93	0.82
[76](k=16)	v6	409	452	806(aprox)	x	3224	414	374.61	0.46
[72](D=64, d=233)	s3	233	10	3458	x	x	172.41	4017.2	1.2
[79](D=64, d=8)	s3	233	80	406	x	x	363.76	1059.47	2.60

is only provided to show that the proposed architecture is faster than the software implementations, and to show the proposed hardware accelerates the multiplication and exponentiation in prime fields even using fewer area resources that other hardware implementations in the literature.

5.7.3 $\mathbb{F}_{2^m}$ multiplication

This section presents a comparison of state of the art finite field multipliers over $\mathbb{F}_{2^m}$. Table 5.7 shows some of the most significant works on hardware architectures for multiplication over $\mathbb{F}_{2^m}$ in FPGAs.

Sutter et al., [71] present the design and implementation of an elliptic curve point multiplication over the binary field $\mathbb{F}_{2^m}$. To improve elliptic curve point multiplication a finite field multiplier over $\mathbb{F}_{2^m}$ is studied and analyzed. So, a digit-serial multiplier

is presented in [71] for different digit sizes: 1 (bit-serial), 2, 4, 8, 16, 24, and 32, and operand size of 163, 233, 409 and 571, which are used in recommended finite fields proposed by NIST. In [71] the results are reported in registers and LookUp Table (LUT)s, since a slice contains 4 LUTs an approximation has been made to a fair comparison assuming full utilization of LUTs and slices. The smaller multipliers proposed by [71] is the bit serial ($d = 1$). So, compared to our proposed hardware architecture for an operand size of 233 the bit serial implementation in [71] require 178 slices, while the proposed in this work require 21 (d=4). For a digit size $d = 16$, the proposed multiplier requires 82 slices achieving an efficiency of 7.55 Mbps/slice. In [71] as the digit size get bigger a better efficiency is achieved at the cost of more hardware resources. For example, the digit-serial with $(d = 16)$ achieve an efficiency of 11.21 Mbps/slice (assuming full utilization of slices and LUTs which is not commonly achieved). However, with the digit-serial implementation, the smaller architecture proposed in [71] is for $(d = 1)$ requiring 178 slices, and the hardware resources cannot be lower.

In [76] is presented a digit-serial multiplier over $\mathbb{F}_{2^m}$, with digit sizes 16 and 32. Area results presented in [76] are in LUTs, since in this work it is used the slices metric, slices where approximated according to the specification that 1 slice is conformed by 4 LUTs. For a digit size of 16 the multiplier over $\mathbb{F}_{2^m}$ proposed in this thesis achieves an efficiency of 1.58 Mbps/slice and requires 137 slices. The most compact architecture reported in [76] for an operand size of 233 bits requires 410 slices (almost three times the proposed in this thesis) and achieves an efficiency of 0.82 Mbps/slice (nearly half of the proposed in this thesis). And for an operand size of 409 the multiplier reported in [76] require 806 slices (9 times the proposed in this thesis, 82) and achieve an efficiency of 0.49 Mbps/slice (the achieved in this thesis is 7.55, 16 times better than [76]).

The $\mathbb{F}_{2^m}$ multipliers proposed in this thesis research were implemented in the old FPGA Spartan 3 for comparison with the works reported in [72, 79]. Compared with the $\mathbb{F}_{2^m}$ multiplier proposed in [72], the multiplier proposed in this thesis research is 25 times smaller and 1.3 more efficient. In [79] is proposed a digit-digit hardware multiplier over $\mathbb{F}_{2^m}$. Compared to the $\mathbb{F}_{2^m}$ multiplier proposed in this thesis research the reported one in [79] achieves a high efficiency 2.60 Mbps/slice against 1.84 Mbps/slices achieved in this thesis research. Both architectures are compact and process the operands and partial results in a digit fashion. However, in [79] only is reported the required slices for the datapath without taking in to account the necessary logic to the control module to address memory blocks. The main difference in this

thesis research compared to [79] is the elimination of one of the partial multipliers in the datapath (Figure 4.13) which reduces hardware resources.

5.7.4 $\mathbb{E}(\mathbb{F}_{2^m})$ scalar multiplication

This section presents a comparison of state of the art finite field scalar multipliers over $\mathbb{E}(\mathbb{F}_{2^m})$. Table 5.8 shows the most significant state of the art work results for scalar multipliers over $\mathbb{E}(\mathbb{F}_{2^m})$ in FPGAs.

Table 5.8: State of the art works for finite field hardware operator over $\mathbb{E}(\mathbb{F}_{2^m})$.

Work	FPGA	Size	Cycles	Slices	Frequency (MHz)	Throughput (kbps)	Efficiency (kbps/slice)
Prop. (k=8)	z7000	233	1553782	**442**	190.04	28.49	0.064
Prop. (k=16)	z7000	233	408547	**626**	149.20	85.09	0.136
Prop. (k=32)	z7000	233	128820	1170	135.31	244.75	**0.209**
Prop. (k=8)	z7000	409	7504232	**453**	190.94	10.40	0.023
Prop. (k=16)	z7000	409	1926426	653	154.44	32.78	0.050
Prop. (k=32)	z7000	409	511493	1183	132.59	106.02	0.090
[71](g=16,d=2)	v5	233	8193	3939	263.15	7483.69	**1.899**
[71](g=8, d=1)	v5	409	45513	5395	181.81	1633.82	0.030
[83]	k7	233	679776	**3016**	255.66	87.63	0.029
[83]	k7	283	1395312	4625	251.98	51.10	0.011
[81]	v7	233	5929	2647	370.00	14540.39	**5.498**
[81]	v7	409	10354	6888	316.00	12482.51	**1.812**
[85]	v5	163	1396	3513	147.00	17.16	0.004

The results presented in [71] are proposed for a digit-serial approach for multiplication and inversion over $\mathbb{F}_{2^m}$, and square and addition over $\mathbb{E}(\mathbb{F}_{2^m})$ are computed fully combinational in only one clock cycle. In [71] the digit size for the digit-serial $\mathbb{F}_{2^m}$ multiplication is represented by g and for inversion by d. So, the comparisons with [71] is with the smaller digit size reported. For the digit size 233 the scalar multiplier presented in [71] requires 3939 slices achieving an efficiency of 1.899 kbps/slice. Compared to the proposed scalar multiplier in this work, the results presented in [71] are almost ten times better according to efficiency. However, the designs proposed in this work are smaller than the reported in [71], for example, for a digit size of 8, 16, and 32 the required area is 442, 626 and 1170 slices respectively. Furthermore, for an operand size of 409 bits, the required hardware results grow very fast compared to the 233 bits in the results reported in [71]. Compared to the proposed scalar multiplier over $\mathbb{E}(\mathbb{F}_{2^m})$ in this work for operand size of 409 bits, the proposed design requires only 445 slices while the reported in [71] require 5395. Moreover, the design proposed in this work for a digit size of 409 bits achieves an efficiency of 0.50 and 0.90 kbps/slice

for a digit size of 16 and 32 respectively, while in [71] the efficiency for a digit size of 409 is 0.30 kbps/slice.

In [83] it is presented a hardware architecture for elliptic curve scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$ implemented for the NIST-recommended binary fields $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{283}}$. Table 5.8 shows that the scalar multiplier hardware architecture presented in [83] requires 3016 and 4625 slices for the operand size 233 and 283 respectively. Compared with the scalar multiplier proposed in this thesis research for operand size of 233 the scalar multiplier presented in [83] requires 6.8 times more slices. Furthermore, in [83] an efficiency of 0.029 Mbps/slice is achieved whilst in this thesis research is achieved an efficiency of 0.064 Mbps/slice (2.2 times better than the one achieved in [83]). The proposed scalar multiplier for 283 bits operand size was not implemented as in [83]. However, It was implemented for 409 bits operand size, and is 10 times smaller than the one proposed in [83] for 283 bits operand size achieving an efficiency of 0.023 Mbps/slice, twice the one achieved in [83] (0.011 Mbps/slice).

In [81] it is presented a throughput/area-efficient elliptic curve scalar point multiplier processor that use the Lopez-Dahab Montgomery algorithm. The hardware presented in [81] was implemented for all the five NIST-recommended binary fields. However for comparison reasons Table 5.8 shows the results presented in [81] for the binary fields $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{409}}$. The scalar multiplier over $\mathbb{E}(\mathbb{F}_{2^m})$ presented in [81] for an operand size of 233 requires 2647 slices and achieves an efficiency of 5.498 kbps/slice, 26 times better than the efficiency results achieved in this research (0.209). And, for an operand size of 409, the hardware architecture presented in [81] required 6888 slices and achieved an efficiency of 1.812 kbps/size, 20 times better than the efficiency (0.090 kbps/slice) achieved with the proposed scalar multiplier. However, even though the results presented in [81] regarding to higher efficiency, the area results (slices) are smaller in the scalar multiplier proposed in this research. Furthermore, for an operand size of 409 bits, the are results presented in [81] grows significantly, from 2647 to 6888 slices for 233 and 409 bits operand size respectively. The area results (slices) for the proposed scalar multiplier depend on the digit size, not in the operand size. So, for operand size of 233 or 409 is almost the same area resources required as can be seen in Figure 5.5.

In [85] it is presented a scalar multiplier over the binary fields $\mathbb{F}_{2^{163}}$ that requires 3789 slices and achieve an efficiency of 0.004 kbps/slice. Compared with the proposed scalar multiplier, the scalar multiplier presented in [85] requires seven times hardware resources with low efficiency.

Even though the results presented in [81] achieve an efficiency superior to the

achieved in work, the proposed scalar multiplier is smaller than all the state of the art hardware architectures. These results give evidence that the proposed design could be used as a small, high-performance hardware accelerator for security in embedded systems getting high speed than the one achieve with a general purpose processor (software implementations).

5.8 In-circuit verification of FPGA finite field operators

This section describes an in-circuit verification approach for the proposed finite field operators $\mathbb{F}_p$ exponentiator, and $\mathbb{E}(\mathbb{F}_{2^m})$ scalar multiplier, by means of a hardware-software co-design. Under this context, the finite field operators are used as a co-processor commanded by a general purpose processor via a bus interface. The finite field operator for exponentiation over $\mathbb{F}_p$ has been evaluated in a hardware-software co-design for the RSA digital signature, and the finite field operator for scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$ has been evaluated in the Elliptic Curve Diffie-Hellman (ECDH) key exchange.

The software used for carried out the hardware-software co-design was:

- Simulation: Model Technology ModelSim Altera vsim 10.3d Simulator 2014.10 Oct 7 2014.
- Synthesis and implementation in: Vivado v2016.1 (64-bit) from Xilinx.
- Implementation platform: MicroZed low-cost development board based on the Xilinx Zynq®-7000 All Programmable SoC which combines ARM dual-core Cortex-A9 MPCore processing system (PS) and 28 nm Xilinx programmable logic (PL) in a single device.
- $\mathbb{F}_p$ Test vectors : BigInteger Java library.
- $\mathbb{E}(\mathbb{F}_{2^m})$ Test vectors: C++ Number Theory Library (NTL).
- Software implementations of the RSA Digital Signature and ECDH key exchange were implemented in pure software in the MicroZed Board under Linux with the MIRALC library for comparisons.

5.8.1 HW/SW co-design for operators over $\mathbb{F}_p$

The finite field operator for exponentiation in $\mathbb{F}_p$ was evaluated in a hardware-software co-design using the RSA digital signature presented in section 2.1.5. The main opera-

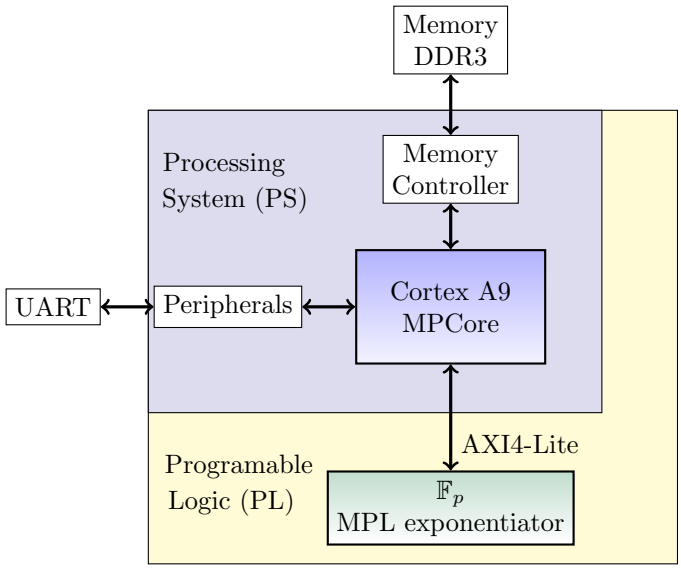

Figure 5.6: Hardware-software co-design for exponentiation over $\mathbb{F}_p$.

tion in the RSA digital signature is the exponentiation over $\mathbb{F}_p$. This operation is used to sign a document D with a private parameter s and public parameter N:

$$S \equiv D^s \bmod N. \tag{5.3}$$

The exponentiation over $\mathbb{F}_p$ is the operation used to verify the validity of a digital signature too:

$$V \equiv S^v \bmod N \tag{5.4}$$

So, in this hardware-software co-design, the exponentiation over $\mathbb{F}_p$ is accelerated through a hardware design. Figure 5.6 shows a general view of the proposed hardware-software co-design. The $\mathbb{F}_p$ module is implemented according to the proposed MPL exponentiator presented in Section 4.3.

Table 5.9 shows the obtained results for the MPL exponentiator module architecture implemented in the MicroZed FPGA. Table 5.10 shows the implementation results of the full hardware-software co-design in the MicroZed. Furthermore, a full software implementation of the RSA digital signature was implemented in the Cortex A9 of the MicroZed FPGA using the MIRACL library. The processing time of the two implementations, full software and the hardware-software co-design, are presented in Table 5.10.

The software implementation with the MIRACL library requires 490 ms to achieve a digital sign/verification scheme with the RSA digital signature algorithm. However,

Table 5.9: Implementation results for MPL exponentiation over $\mathbb{F}_p$ in the MicroZed.

Size	k	d	Area (slice)	Freq. (MHz)	Throughput (kbps)	Efficiency (kbps/slice)	Time (ms)
1024	32	32	318	90.9	85.42	0.268	11.98

Table 5.10: Implementation results for the RSA digital signature hw-sw co-design in the MicroZed.

Size	k	Area (slices)	Freq. (MHz)	Time (ms)	MIRACL TIME (ms)
1024	32	530	81.5	26.2033	490

with the proposed compact hardware architecture the required time is 26.20 ms, that is 18 times less. The finite field operator for $\mathbb{F}_p$ exponentiation (MPL) only requires 318 slices which is almost the 7.22% of the disposal slices. This 7% of the slices in the FPGAs can be used to provide speed in the security of an IoT application while clearly another 92% can be used the implement other requirements of the application.

The parameters and test vector of the hardware-software co-design for the RSA digital signature algorithm are shown in Appendix A.1.

5.8.2 HW/SW co-design for operators over $\mathbb{E}(\mathbb{F}_{2^m})$

The finite field operator for scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$ was evaluated in a hardware-software co-design using the Diffie-Hellman key exchange Elliptic Curve version presented in Section 2.2.8. In the ECDH key exchange the parties, for example, an FPGA server and an FPGA sensor, agree on an elliptic curve $\mathbb{E}(\mathbb{F}_{2^m})$ and a point $P \in \mathbb{E}(\mathbb{F}_{2^m})$. Then the parties select a secret integer each, for example, n_A and n_B. And, using the scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$ they compute the values.

$$\underbrace{Q_A = n_A P}_{\mathrm{SensorFPGA}} \qquad \text{and} \qquad \underbrace{Q_B = n_B P}_{\mathrm{ServerFPGA}} \tag{5.5}$$

The values $Q_A = Q_B$ is the shared secret key exchange between the FPGA server and the sensor.

In elliptic curve cryptosystems, the main operation is the scalar multiplication as in the Elliptic Curve Diffie-Hellman key exchange. So, in this hardware software co-design, the scalar multiplication over $\mathbb{E}(\mathbb{F}_{2^m})$ is implemented directly in hardware to accelerate the computation. Figure 5.7 shows the proposed hardware-software co-design for the scalar multiplier over $\mathbb{E}(\mathbb{F}_{2^m})$. The $\mathbb{E}(\mathbb{F}_{2^m})$ scalar multiplier module is

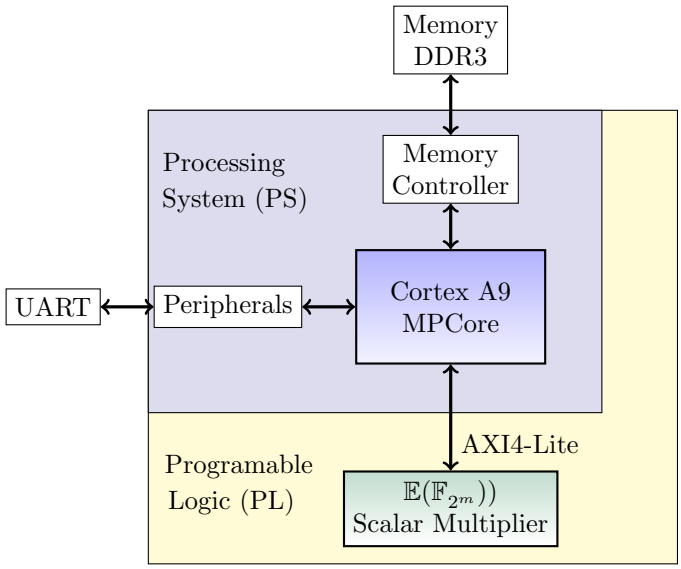

Figure 5.7: Hardware-software co-design for $\mathbb{E}(\mathbb{F}_{2^m})$ scalar multiplier.

Table 5.11: Scalar multiplier over $\mathbb{E}(\mathbb{GF}(2^m))$ implementation results in the MicroZed board.

Size	k	digits	Area (slcies)	Freq. (MHz)	Throughput (kbps)	Efficiency (kbps/slice)	Time (ms)
233	32	8	1619	62.5	115.146	0.071	2.024

implemented according to the proposed hardware architecture presented in Section 4.6.

Implementation results for this module implementation in the MicroZed board are shown in Table 5.11. The scalar multiplier module requires 1619 slices for a digit size of 32 bits, achieving a throughput of 115.146 kbps and an efficiency of 0.071 (kbps/slice). Table 5.12 shows the implementation results for the hardware-software co-design for the ECDH in the MicroZed board. In Table 5.12 results for the software implementation with the MIRACL library are presented too. The hardware-software co-design requires 1809 slices running a maximum frequency of 62.5 (MHz). The proposed hardware-software co-design requires 4.13 ms to compute an Elliptic Curve Diffie Hellman key exchange. The pure software implementation in the MicroZed with the MIRACL library requires 70 ms for projective coordinates and 200 ms for affine coordinates. The proposed hardware-software co-design is 17 times faster than the pure software implementation. However, the proposed scalar multiplier module requires 36% of the slices resources in the MicroZed, leaving 66% of the slices for others application requirements.

Table 5.12: ECDH hw-sw co-design implementation results.

Size	k	Area (slices)	Freq. (MHz)	Time (ms)	MIRACL TIME Proj (ms)	MIRACL TIME Aff (ms)
233	32	1809	62.5	4.1352	70	200

5.9 Summary

In this section obtained results for the finite field operators discussed in Chapter 4 have been presented. Furthermore, a comparison of the proposed finite field operators with state of the art works has been discussed. The evaluation of the finite field operators has been carried out with the MicroZed board, a target used in Industrial Internet of Things applications. Software implementations have been presented in the MicroZed and compared with the hardware-software co-design shows a considerable speed up of the RSA Digital Signature and Elliptic Curves Diffie-Hellman key exchange algorithm.

The proposed finite field operators require less hardware are resources than state of the art hardware architectures. For example, for 1024 bits exponentiation in the prime field $\mathbb{F}_p$, the proposed hardware finite field operator requires 574 slices only and achieves and efficiency of 0.503 kbps (k=64), the 54% of slice and the 121% of efficiency of the hardware architecture reported in [62] (one of the most compact works reported in the literature). For 233 bits scalar multiplication in elliptic curves over $\mathbb{E}(\mathbb{F}_{2^m})$, the proposed hardware operator requires 1170 slices and achieves a efficiency of 0.209 kbps, the 44% of slices reported in [81]. However, the $\mathbb{E}(\mathbb{F}_{2^m})$ operator proposed in this research achieves 26% of the efficiency achieved in [81]. Despite the proposed operator for scalar multiplication in elliptic curves $\mathbb{E}(\mathbb{F}_{2^m})$ have lower efficiency than the hardware architecture implementations presented in the state of the art, the proposed operators improve the execution time of software implementations, such as the MIRACL library. For example, for a RSA digital signature a software implementation with the MIRACL library in the MicroZed board require 4090 ms for RSA digital signature and 70 ms for ECDH key exchange while using the proposed hardware operator $\mathbb{F}_p$ requires 26.2 ms and 4.1 ms respectively.

In view of the experimental results, it can be concluded that the digit-digit approach allows more compact hardware implementations than implementations with other approaches such as bit-serial and digit-serial, and outperforms software implementations. The presented results give evidence that the initial hypothesis proposed in chapter 1 is correct.

In the next section, the conclusions of this thesis research and new directions for

this project is presented.

Chapter 6

Conclusions and directions

This research work presented the design and implementation of hardware finite field operators for public key cryptography on resource-constrained devices. The studied operators are multiplication in finite fields $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$, and exponentiation in the multiplicative group $\mathbb{F}_p^*$ and the additive elliptic curve group $\mathbb{E}(\mathbb{F}_{2^m})$. Different from most of the state of the art works that implement hardware finite field operators for high speed, this works is focused on compact implementations.

Algorithms for multiplications in finite field $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$ were adapted to implement this operation with a small area footprint compared to state of the art works. Furthermore, in spite of a reduction of hardware resources, the proposed multipliers achieve better performance than software counterparts. FPGAs is the technology used to implement the proposed hardware finite field operators. To validate the proposed operators, a hardware-software co-design was implemented in a small FPGA targeted for Industrial Internet of Things, the MicroZed. The RSA digital signature and Elliptic Curve Diffie-Hellman (ECDH) algorithms were used to evaluate the proposed finite field and group operators in the hardware-software co-design. Despite the few hardware resources required for the implementation compared with state of the art works, the proposed co-design achieves a better throughput compared with software implementations.

The proposed hardware architectures would allow implementing security services (confidentiality, authentication, integrity and non-repudiation) in IoT applications with a small amount of hardware where time execution is crucial. For example, a vehicular network (also known as VANETs) will contribute to safer and more efficient roads by providing timely information to drivers and specific authorities that are concerned by enabling vehicles to communicate with each other via Inter-Vehicle Communication (IVC) as well as with roadside base stations via Roadside-to-Vehicle Communication (RVC).

6.1 Summary of contributions

The main contributions achieved in this thesis research are:

- Novel digit-digit Montgomery multiplier algorithm.
- Novel digit-digit $\mathbb{F}_{2^m}$ multiplier algorithm.
- Compact finite field hardware operators for:
  - Proposed Montgomery multiplier algorithm.
  - Proposed multiplication over $\mathbb{F}_{2^m}$.
  - Montgomery Powering Ladder exponentiation algorithm over $\mathbb{F}_p$.
  - Elliptic curve Montgomery scalar multiplier algorithm.
- Hardware-software co-design for:
  - RSA Digital Signature algorithm.
  - Elliptic Curve Diffie-Hellman key exchange algorithm.

The proposed hardware finite field operators require fewer hardware resources (slices) in FPGAs compared with state of the art works. For example, for 1024 bits operand size with a digit size $k = 32$ the proposed hardware exponentiation operator in prime field $\mathbb{F}_p$ requires 318 slices, that is only a third of the 1060 slices reported in [62], and for 233 bits with a digit size $k = 32$ the proposed hardware scalar multiplier over $\mathbb{E}(\mathbb{F}_{2^m})$ requires 1619 slices only the 61% of the 2647 slices required in the hardware architecture reported in [81].

Furthermore, these operators achieve higher throughput than software implementations and can be used by a hardware designer to speed up cryptography requirements with only a few hardware resources in FPGAs. For example, for a RSA digital signature a hardware-software co-design requires 530 slices in the MicroZed board, and the digital signature is accelerated from 490 ms (MIRACL software implementation) to 26.20 ms (proposed finite field operator). For a ECDH key exchange a hardware-software co-design requires 1809 slices, and the key exchange is accelerated from 70 ms (MIRACL software implementation) to 4.13 ms (proposed finite field operator).

Also, the proposed operators can handle several levels of security (key size) with almost the same hardware resources, since hardware resources required for operators depends on the selected digit size instead of the operand size (key size). In the case of the exponentiation in $\mathbb{F}_p$ using the digit size $k = 32$ the proposed hardware operator

for 512, 1024 and 2048 requires 236, 228 and 246 slices respectively. Although different key sizes are used the hardware architecture requires similar hardware resources since area depends on the digit size not on the operand size (key length).

6.2 Publications

As contributions of this thesis research the obtained results have been published in:

- International conferences:
  - [98] L. Rodriguez-Flores, M. Morales-Sandoval, R. Cumplido, C. Feregrino-Uribe, and I. Algredo-Badillo, "A compact FPGA-based microcoded coprocessor for exponentiation in asymmetric encryption," in *2017 IEEE 8th Latin American Symposium on Circuits Systems (LASCAS)*, pp. 1–4, Feb 2017
- Scientific journals:
  - [99] L. Rodríguez-Flores, M. Morales-Sandoval, R. Cumplido, C. Feregrino-Uribe, and I. Algredo-Badillo, "Compact FPGA hardware architecture for public key encryption in embedded devices," *PLOS ONE*, vol. 13, pp. 1–21, 01 2018

6.3 Future work

As future work, hardware resources required to implement the control module for the scalar multiplier for elliptic curves can be reduced using a microprogramming approach, similar to the one used in the proposed Montgomery Powering Ladder exponentiation [98]. In some cases, it is required to use RSA and ECC for ubiquitous applications, and configure the FPGA to execute both public key cryptosystems would require a considerable amount of hardware resources. So, for future work partial reconfiguration would be implemented, where the part in common of both algorithms (inputs and output) could be configured first, and the FPGA could be partially reconfigured according to the required public key cryptosystem (RSA or ECC) using the same hardware resources. Further, frequency performance can be improved in the digit-digit Elliptic Curve Cryptosystem, a possible solution could be to explore a pipeline architecture for the Karatsuba algorithm in the partial $\mathbb{F}_{2^m}$, and using a microprogramming approach for the control module.

Appendix A

Test vectors

A.1 RSA digital signature scheme

A.2 Elliptic Curve Diffie-Hellam key exchange

Table A.1: Selected parameters for in-circuit verification of the RSA digital signature scheme.

1	• Secret primes p and q. p (511) = 5469912423305568567094794773858602122684237254961747880093981997316855161742613466358995126733325429931110509592017447931679634071960799533767695972929567 q (511) = 4639978112936603647786606685334759356804090437428374323297913574251805543834158080534230794320862369480602413257682993057259495834558466289466966270656683
2	Verification exponent v with $\gcd(v(p-1)(q-1)) = 1$ v (1022)= 2484263065763093369215393017449450416820872966309007234628644562892066697221531011819944927836324104220791732928693913365207959522067392284683375534320891915888840461664375793625014741434856227491040491199036037943173562933337045797234359697175141270152975302041971531841927816407906637247587546794669354412 1
3	Compute s with $sv \equiv 1(\bmod (p-1)(q-1))$ s (1020) = 6276597309394928732834039851507921383298415966136342518245053565366137742299828910753833423164941887611417572791677334327358062790756059452705353356504508972201697335745511427523024406542029600332268106106583596508803120382513798664696130702854073375191020798250847114165984317272887598238756454530180074073
4	Publish $N = pq$ and v N (1022) = 2538027392381785676777632417555326700928721895657088121369769219905967092070282224591259533213764094561352053324242997522674068289964426253220466266355728322525979352713285137847165123207309706674809308679350522003228004958648254852572620113757679319999480436182568662653391705578477149607297576929639684626 1 v (1022)= 2484263065763093369215393017449450416820872966309007234628644562892066697221531011819944927836324104220791732928693913365207959522067392284683375534320891915888840461664375793625014741434856227491040491199036037943173562933337045797234359697175141270152975302041971531841927816407906637247587546794669354412 1

Table A.2: Test vectors for the RSA digital signature

1	• Digital message D D (1020) = 1030742523126596155613651427610589444524308860190830519347316753401154575376812404053192841158192051108383367762004122707268780503059963916823037164347555132639703882095418910600658383331269975028604593293808035121858463426382611337662752535881686865120567940094788901028903617205055616315858940258120577834 3
2	• Signature $S \equiv D^s \bmod N$ S (1020) = 9894014987019166052753873363608809729766777110039352398269912948582504387395830546632051856168117130059698458457201928137113934881374597584020484837181684826528800954014039510249639768691252973100462682728987490809026860200710658335528373893882083607795130184143784343765932355311566265725320881480735183798

Table A.3: Test vectors for the RSA digital signature verification

1	• Digital message D D (1020) = 1030742523126596155613651427610589444524308860190830519347316753401154575376812404053192841158192051108383367762004122707268780503059963916823037164347555132639703882095418910600658383331269975028604593293808035121858463426382611337662752535881686865120567940094788901028903617205055616315858940258120577834 3
2	• Verification $V \equiv S^v \bmod N$ V (1020) = 1030742523126596155613651427610589444524308860190830519347316753401154575376812404053192841158192051108383367762004122707268780503059963916823037164347555132639703882095418910600658383331269975028604593293808035121858463426382611337662752535881686865120567940094788901028903617205055616315858940258120577834 3

Table A.4: Test vector for ECDH in the hw-sw co-design (hexadecimal base)

Public	• Base point $P = (G_X, G_Y)$ G_x = 00000172 32ba853a 7e731af1 29f22ff4 149563a4 19c26bf5 0a4c9d6e efad6126 G_y = 000001db 537dece8 19b7f70f 555a67c4 27a8cd9b f18aeb9b 56e0c110 56fae6a3
Sensor	• Chose a secret integer n_A n_A = 00000080 00000000 00000000 00000000 00069d5b b915bcd4 6efb1ad5 f173abcb • Compute the point $Q_A = n_A P$ Q_{Ax} = 00000111 2DBC0A61 3B981F29 128690DD 072BA010 63991B08 FC5706B6 BAA234FA Q_{Ay} = 00000007 CD042450 6011DD2F 130EEE07 C2AFC012 4FE4FA6E C2C2980B 8B235C0C • Alice send $Q_A(Q_{Ax}, Q_{Ay})$ to Bob.
Server	• Chose a secret integer n_B n_B = 00000080 00000000 00000000 00000000 00069d5b b915bcd4 6efb1ad5 f173abdb • Compute the point $Q_B = n_B P$ Q_{Bx} = 000000C1 27A0AAB6 AE3AE1E4 206B5483 0E8D1DAC C79AD742 ED00E8FD 6C9849E6 Q_{By} = 00000018 7E3AA562 86523706 DFD281A5 BDBAB72A 3B2FC40B 9D78C787 FC847465 • Bob send $Q_B(Q_{Bx}, Q_{By})$ to Alice
Sensor	• Alice compute the point $R = n_A Q_B$ R_x = 000001A1 7DBD1F69 608A8B7B 9E3833A7 0E538175 7F843622 F07765FE E79FAF4D R_y = 00000033 76EA5CD0 ABD23067 BBE71A7B 608F84D5 98A38DD7 F8E60ECC 48C5763D
Server	• Bob compute the point $R = n_B Q_A$ R_x = 000001A1 7DBD1F69 608A8B7B 9E3833A7 0E538175 7F843622 F07765FE E79FAF4D R_y = 00000033 76EA5CD0 ABD23067 BBE71A7B 608F84D5 98A38DD7 F8E60ECC 48C5763D

Appendix B

List of Acronyms

IoT Internet of Things

AmI Ambient Intelligence

PC Personal Computer

RFID Radio-frequency identification

WSN Wireless Sensor Network

BAN Body Area Network

SKC Symmetric Key Cryptography

PKC Public Key Cryptography

ECC Elliptic Curve Cryptography

DH Diffie-Hellam

ECDH Elliptic Curve Diffie-Hellam

RSA Rivest–Shamir–Adleman

ECC Elliptic Curve Cryptography

RAM Random-access memory

NIST National Institute of Standards and Technology

LWC lightweight cryptography

FPGA Field-Programmable Gate Array

HECC Hyperelliptic Curve Cryptosystems

DLP Discrete Logarithm Problem

ECDLP Elliptic Curve Discrete Logarithm Problem

ECDH Elliptic Curve Diffie-Hellman

ASICs Application Specific Integrated Circuits

DSA Digital Signature Algorithm

FLT Fermat's Little Theorem

MPL Montgomery Powering Ladder

NAF Non-adjacent form

CSA Carry Save Adders

PE Process Elements

SOS Separated Operand Scanning

CIOS Coarsely Integrated Operand Scanning

FIOS Finely Integrated Operand Scanning

FIPS Finely Integrated Product Scanning

CIHS Coarsely Integrated Hybrid Scanning

CIHS Coarsely Integrated Hybrid Scanning

DSP Digital Signal Processing

LUT LookUp Table

LSB Least Significant Bit

MSB Most Significant Bit

L2R Left to Right

R2L Right to Left

MSE Most Significant Element first

LSE Least Significant Element first

BRAM Block RAM

MMA Montgomery Multiplication Algorithm

FSM Finite State Machine

SPA Simple Power Analysis

DPA Differential Power Analysis

CLB Configurable Logic Block

XPA Xilinx Power Analyzer

Bibliography

[1] T. Eisenbarth and S. Kumar, "A survey of lightweight-cryptography implementations," *Design & Test of Computers, IEEE*, vol. 24, no. 6, pp. 522–533, 2007.

[2] M. N. Hassan and M. Benaissa, "Low area-scalable hardware/software co-design for elliptic curve cryptography," in *2009 3rd International Conference on New Technologies, Mobility and Security*, pp. 1–5, Dec 2009.

[3] M. Morales-Sandoval and A. Diaz-Perez, "A compact FPGA-based Montgomery multiplier over prime fields," in *Proceedings of the 23rd ACM International Conference on Great Lakes Symposium on VLSI*, GLSVLSI '13, (New York, NY, USA), pp. 245–250, ACM, 2013.

[4] C. D. Walter, *Montgomery's Multiplication Technique: How to Make It Smaller and Faster*, pp. 80–93. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999.

[5] G. O'Regan, *A Brief History of Computing*. London: Springer London, 2012.

[6] M. Weiser, "The computer for the 21st century," *Scientific American*, vol. 265, no. 3, pp. 94–104, 1991.

[7] M. Weiser, "Ubiquitous computing," *IEEE Computer Hot Topics*, Oct. 1993.

[8] D. Jain, P. V. Krishna, and V. Saritha, "A study on internet of things based applications," June 18 2012.

[9] D. J. Cook, J. C. Augusto, and V. R. Jakkula, "Ambient intelligence: Technologies, applications, and opportunities," *Pervasive and Mobile Computing*, vol. 5, no. 4, pp. 277 – 298, 2009.

[10] Y. Venkataramana, "Pervasive computing: Implications, opportunities and challenges for the society," in *Pervasive Computing and Applications, 2006 1st International Symposium on*, pp. 5–5, Aug 2006.

[11] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J. Burgelman, "Scenarios for ambient intelligence in 2010," 01 2001.

[12] L. Z. Cai and M. F. Zuhairi, "Security challenges for open embedded systems," in *Engineering Technology and Technopreneurship (ICE2T), 2017 International Conference on*, pp. 1–6, IEEE, 2017.

[13] E. Bertino and N. Islam, "Botnets and internet of things security," *Computer*, vol. 50, no. 2, pp. 76–79, 2017.

[14] Ahmad Sharifi, Mohsen Khosravi and Dr. Asadullah Shah, "Security Attacks And Solutions On Ubiquitous Computing Networks," *International Journal of Engineering and Innovative Technology (IJEIT)*, vol. 3, pp. 40–45, Oct. 2013.

[15] F. Stajano, "Security issues in ubiquitous computing*," in *Handbook of Ambient Intelligence and Smart Environments* (H. Nakashima, H. Aghajan, and J. Augusto, eds.), pp. 281–314, Springer US, 2010.

[16] D. G. Padmavathi, M. Shanmugapriya, and others, "A survey of attacks, security mechanisms and challenges in wireless sensor networks," *arXiv preprint arXiv:0909.0576*, 2009.

[17] D. Martins and H. Guyennet, "Wireless sensor network attacks and security mechanisms: A short survey," in *Network-Based Information Systems (NBiS), 2010 13th International Conference on*, pp. 313–320, Sept 2010.

[18] C. Douligeris and D. Serpanos, *Network Security: Current Status and Future Directions*. Wiley, 2007.

[19] A. Alkalbani, T. Mantoro, and A. Tap, "Comparison between rsa hardware and software implementation for wsns security schemes," in *Information and Communication Technology for the Muslim World (ICT4M), 2010 International Conference on*, pp. E84–E89, Dec 2010.

[20] H. Wang and Q. Li, "Efficient implementation of public key cryptosystems on mote sensors (short paper)," in *Information and Communications Security* (P. Ning, S. Qing, and N. Li, eds.), vol. 4307 of *Lecture Notes in Computer Science*, pp. 519–528, Springer Berlin Heidelberg, 2006.

[21] W. Stallings, *Cryptography and Network Security: Principles and Practice (6th Edition)*. Prentice Hall, 6 ed., 3 2013.

[22] NIST, *Recommended elliptic curves for federal government use*, 1999. http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf.

[23] C. Manifavas, G. Hatzivasilis, K. Fysarakis, and K. Rantos, "Lightweight cryptography for embedded systems a comparative analysis," in *Data Privacy Management and Autonomous Spontaneous Security* (J. Garcia-Alfaro, G. Lioudakis, N. Cuppens-Boulahia, S. Foley, and W. M. Fitzgerald, eds.), vol. 8247 of *Lecture Notes in Computer Science*, pp. 333–349, Springer Berlin Heidelberg, March 2014.

[24] C. Swenson, *Modern Cryptanalysis: Techniques for Advanced Code Breaking*. Wiley Publishing, 2008.

[25] C. Paar and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer-Verlag New York Inc, 2010.

[26] P. S. D. Galbraith, *Mathematics of Public Key Cryptography*. Cambridge University Press, 0 ed., 4 2012.

[27] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, pp. 644–654, Nov. 1976.

[28] J. Hoffstein, J. Pipher, and J. H. Silverman, *An Introduction to Mathematical Cryptography*. Springer Publishing Company, Incorporated, 2nd ed., 2014.

[29] A. R. Meijer, *Algebra for Cryptologists*. Springer Publishing Company, Incorporated, 1st ed., 2016.

[30] F. Rodriguez-Henriquez, N. A. Saqib, A. D. Prez, and C. K. Koc, *Cryptographic Algorithms on Reconfigurable Hardware*. Springer Publishing Company, Incorporated, 1st ed., 2010.

[31] R. Schoof, "The discrete logarithm problem," *Open problems in mathematics, Springer*, pp. 403–416, 2016.

[32] K. S. McCurley, "The discrete logarithm problem," in *Cryptology and Computational Number Theory* (C. Pomerance, ed.), vol. 42 of *Proceedings of Symposia in Applied Mathematics*, (Providence), pp. 49–74, American Mathematical Society, 1990.

[33] J. Zhang and L.-Q. Chen, "An improved algorithm for discrete logarithm problem," in *Environmental Science and Information Application Technology, 2009. ESIAT 2009. International Conference on*, vol. 2, pp. 658–661, July 2009.

[34] M. Xiaojing, L. ZhiTang, and T. Hao, "Solving the discrete logarithm problem using p systems," in *Networks Security, Wireless Communications and Trusted Computing, 2009. NSWCTC '09. International Conference on*, vol. 2, pp. 438–441, April 2009.

[35] R. L. Rivest, A. Shamir, and L. Adelman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[36] N. Nedjah, L. Mourelle, M. Santana, and S. Raposo, "Massively parallel modular exponentiation method and its implementation in software and hardware for high-performance cryptographic systems," *Computers Digital Techniques, IET*, vol. 6, pp. 290–301, September 2012.

[37] S. Kumar, T. Wollinger, and C. Paar, "Optimum digit serial $\mathbb{GF}(2^m)$ multipliers for curve-based cryptography," *IEEE Transactions on Computers*, vol. 55, pp. 1306–1311, Oct 2006.

[38] V. S. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology — CRYPTO '85 Proceedings* (H. C. Williams, ed.), (Berlin, Heidelberg), pp. 417–426, Springer Berlin Heidelberg, 1986.

[39] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987.

[40] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.

[41] Z. U. A. Khan and M. Benaissa, "High-speed and low-latency ecc processor implementation over $gf(2^m)$ on fpga," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, pp. 165–176, Jan 2017.

[42] V. Miller, "Use of elliptic curves in cryptography.," in *CRYPTO: Proceedings of Crypto*, pp. 417–426, 01 1985.

[43] M. Joye and S.-M. Yen, "The montgomery powering ladder," in *Cryptographic Hardware and Embedded Systems-CHES 2002*, pp. 291–302, Springer, 2003.

[44] A. Coman and R. Fratila, "Cryptographic applications using fpga technology," *Journal of Mobile, Embedded and Distributed Systems*, vol. 3, no. 1, pp. 10–16, 2011.

[45] M. Rao, J. Coleman, and T. Newe, "An fpga based reconfigurable ipsec esp core suitable for iot applications," in *2016 10th International Conference on Sensing Technology (ICST)*, pp. 1–5, Nov 2016.

[46] N. Shylashree and V. Sridhar, "Efficient implementation of scalar multiplication for ecc in gf (2m) on fpga," in *2015 International Conference on Emerging Research in Electronics, Computer Science and Technology (ICERECT)*, pp. 472–476, Dec 2015.

[47] X. Guo, Z. Chen, and P. Schaumont, *Energy and Performance Evaluation of an FPGA-Based SoC Platform with AES and PRESENT Coprocessors*, pp. 106–115. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[48] F. Gebali and A. Ibrahim, "Low space-coplexity and low power semi-systolic multiplier architecture over gf(2m) based on irreducible trinomial," *Microprocessors and Microsystems*, 2016.

[49] J.-L. Beuchat, T. Miyoshi, Y. Oyama, and E. Okamoto, "Multiplication over fpm on fpga: A survey," in *ARC*, 2007.

[50] C. K. KOC, "High-speed rsa implementation version 2.0," *RSA Security*, 1994.

[51] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[52] P. C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Proceedings of the 19th Annual International Conference on Advances in Cryptology*, (London, UK, UK), pp. 388–397, Springer-Verlag, 1999.

[53] J. López and R. Dahab, "Fast multiplication on elliptic curves over gf(2m) without precomputation," in *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '99, (London, UK, UK), pp. 316–327, Springer-Verlag, 1999.

[54] P. L. Montgomery, "Speeding the Pollard and Elliptic Curve Methods of Factorization," *Mathematics of Computation*, vol. 48, no. 177, pp. 243–264, 1987.

[55] P. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[56] M. Shieh, J. Chen, H. Wu, and W. Lin, "A new modular exponentiation architecture for efficient design of rsa cryptosystem," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, pp. 1151–1161, Sept 2008.

[57] G. Sutter, J. Deschamps, and J. Imana, "Modular multiplication and exponentiation architectures for fast RSA cryptosystem based on digit serial computation," *IEEE Transactions on Industrial Electronics*, vol. 58, pp. 3101–3109, July 2011.

[58] Y.-Y. Zhang, Z. Li, L. Yang, and S.-W. Zhang, "An efficient csa architecture for montgomery modular multiplication," *Microprocessors and Microsystems*, vol. 31, no. 7, pp. 456 – 459, 2007.

[59] E. Oksuzoglu and E. Savas, "Parametric, secure and compact implementation of RSA on FPGA," in *International Conference on Reconfigurable Computing and FPGAs*, pp. 391–396, 2008.

[60] S. B. Ors, L. Batina, B. Preneel, and J. Vandewalle, "Hardware implementation of a montgomery modular multiplier in a systolic array," in *Proceedings International Parallel and Distributed Processing Symposium*, pp. 8 pp.–, April 2003.

[61] A. P. Fournaris and O. Koufopavlou, "A new rsa encryption architecture and hardware implementation based on optimized montgomery multiplication," in *2005 IEEE International Symposium on Circuits and Systems*, pp. 4645–4648 Vol. 5, May 2005.

[62] I. San and N. At, "Improving the computational efficiency of modular operations for embedded systems," *Journal of Systems Architecture*, vol. 60, no. 5, pp. 440 – 451, 2014.

[63] X. Yan, G. Wu, D. Wu, F. Zheng, and X. Xie, "An implementation of montgomery modular multiplication on fpgas," in *2013 International Conference on Information Science and Cloud Computing*, pp. 32–38, Dec 2013.

[64] C. K. Koc, T. Acar, and B. S. Kaliski, "Analyzing and comparing montgomery multiplication algorithms," *IEEE Micro*, vol. 16, pp. 26–33, June 1996.

[65] Y. S. Kim, W. S. Kang, and J. R. Choi, "Asynchronous implementation of 1024-bit modular processor for rsa cryptosystem," in *Proceedings of Second IEEE Asia Pacific Conference on ASICs. AP-ASIC 2000 (Cat. No.00EX434)*, pp. 187–190, Aug 2000.

[66] A. Rezai and P. Keshavarzi, "High-throughput modular multiplication and exponentiation algorithms using multibit-scan-multibit-shift technique," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, pp. 1710–1719, Sept 2015.

[67] K. Javeed, X. Wang, and M. Scott, "Serial and parallel interleaved modular multipliers on fpga platform," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, Sept 2015.

[68] D. M. Gordon, "A survey of fast exponentiation methods," *J. Algorithms*, vol. 27, pp. 129–146, 1998.

[69] T. Wu, S. Li, and L. Liu, "Fast, compact and symmetric modular exponentiation architecture by common-multiplicand Montgomery modular multiplications," *Integration, the VLSI Journal*, vol. 46, no. 4, pp. 323 – 332, 2013.

[70] J.-C. Ha and S.-J. Moon, "A common-multiplicand method to the montgomery algorithm for speeding up exponentiation," *Information Processing Letters*, vol. 66, no. 2, pp. 105 – 107, 1998.

[71] G. D. Sutter, J. Deschamps, and J. L. Imana, "Efficient elliptic curve point multiplication using digit-serial binary field operations," *IEEE Transactions on Industrial Electronics*, vol. 60, pp. 217–225, Jan 2013.

[72] J.-L. Beuchat, T. Miyoshi, Y. Oyama, and E. Okamoto, "Multiplication over f_{p^m} on fpga: A survey," in *Reconfigurable Computing: Architectures, Tools and Applications, Third International Workshop, ARC 2007, Mangaratiba, Brazil, March 27-29, 2007* (P. C. Diniz, E. Marques, K. Bertels, M. M. Fernandes, and J. M. P. Cardoso, eds.), vol. 4419 of *Lecture Notes in Computer Science*, pp. 214–225, Springer, 2007.

[73] G. Bertoni, J. Guajuargo, S. Kumar, G. Orlando, C. Paar, and T. Wollinger, "Efficient gf(pm) arithmetic architectures for cryptographic applications," *Topics in Cryptography*, 2003.

[74] C. Shu, S. Kwon, and K. Gaj, "Fpga accelerated tate pairing based cryptosystems over binary fields," in *2006 IEEE International Conference on Field Programmable Technology*, pp. 173–180, Dec 2006.

[75] L. Song and K. K. Parhi, "Low-energy digit-serial/parallel finite field multipliers," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 19, pp. 149–166, Jul 1998.

[76] D. Pamula and E. Hrynkiewicz, "Area-speed efficient modular architecture for gf(2 lt;sup gt;m lt;/sup gt;) multipliers dedicated for cryptographic applications," in *2013 IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 30–35, April 2013.

[77] D. Pamula, *Arithmetic operators on GF(2m) for cryptographic applications: performance - power consumption - security tradeoffs*. Theses, Université Rennes 1, Dec. 2012.

[78] J.-P. Deschamps, *Hardware Implementation of Finite-Field Arithmetic*. New York, NY, USA: McGraw-Hill, Inc., 1 ed., 2009.

[79] M. Morales-Sandoval and A. Diaz-Perez, "Area/performance evaluation of digit-digit $\mathbb{GF}(2^k)$ multipliers on fpgas," in *FPL*, pp. 1–6, IEEE, 2013.

[80] A. P. Fournaris and O. Koufopavlou, "Low area elliptic curve arithmetic unit," in *2009 IEEE International Symposium on Circuits and Systems*, pp. 1397–1400, May 2009.

[81] Z. Khan and M. Benaissa, "Throughput/area-efficient ecc processor using montgomery point multiplication on fpga," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, pp. 1078–1082, Nov 2015.

[82] W. Wei, L. Zhang, and C. Chang, "A modular design of elliptic-curve point multiplication for resource constrained devices," in *2014 International Symposium on Integrated Circuits (ISIC)*, pp. 596–599, Dec 2014.

[83] M. S. Hossain, E. Saeedi, and Y. Kong, "High-speed, area-efficient, fpga-based elliptic curve cryptographic processor over nist binary fields," in *2015 IEEE International Conference on Data Science and Data Intensive Systems*, pp. 175–181, Dec 2015.

[84] J.-H. Guo and C.-L. Wang, "Systolic array implementation of euclid's algorithm for inversion and division in gf(2/sup m/)," in *1996 IEEE International Symposium on Circuits and Systems. Circuits and Systems Connecting the World. ISCAS 96*, vol. 2, pp. 481–484 vol.2, May 1996.

[85] S. S. Roy, C. Rebeiro, and D. Mukhopadhyay, "Theoretical modeling of elliptic curve scalar multiplier on lut-based fpgas for area and speed," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, pp. 901–909, May 2013.

[86] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, *Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs*, pp. 119–132. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[87] L. Song and K. K. Parhi, "Low-energy digit-serial/parallel finite field multipliers," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 19, pp. 149–166, Jul 1998.

[88] P. L. Montgomery, "Speeding the Pollard and Elliptic Curve Methods of Factorization," *Mathematics of Computation*, vol. 48, no. 177, pp. 243–264, 1987.

[89] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, 2011.

[90] N. Homma, A. Miyamoto, T. Aoki, A. Satoh, and A. Samir, "Comparative power analysis of modular exponentiation algorithms," *IEEE Transactions on Computers*, vol. 59, pp. 795–807, June 2010.

[91] T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel, "A survey of lightweight-cryptography implementations," *IEEE Design Test of Computers*, vol. 24, pp. 522–533, Nov 2007.

[92] T. Itoh and S. Tsujii, "A fast algorithm for computing multiplicative inverses in gf(2m) using normal bases," *Information and Computation*, vol. 78, no. 3, pp. 171 – 177, 1988.

[93] M. Morales-Sandoval and A. Diaz-Perez, "Scalable gf(p) montgomery multiplier based on adigit–digitcomputation approach," *IET Computers Digital Techniques*, vol. 10, no. 3, pp. 102–109, 2016.

[94] Z. Liu, L. Xia, J. Jing, and P. Liu, "A tiny rsa coprocessor based on optimized systolic montgomery architecture," in *Proceedings of the International Conference on Security and Cryptography*, pp. 105–113, July 2011.

[95] T. Tuan and S. Trimberger, "The power of FPGA architectures-the present and future of low-power FPGA design.," *Xcell Journal*, pp. 12–15, Secont Quarter, 2007.

[96] A. S. Alkalbani, T. Mantoro, and A. O. M. Tap, "Comparison between RSA hardware and software implementation for WSNs security schemes," in *Information and Communication Technology for the Muslim World ICT4M, 2010 International Conference on*, pp. E84–E89, IEEE, 2010.

[97] L. Qiu, Z. Liu, G. C. C. F. Pereira, and H. Seo, "Implementing rsa for sensor nodes in smart cities," *Personal and Ubiquitous Computing*, vol. 21, pp. 807–813, Oct 2017.

[98] L. Rodriguez-Flores, M. Morales-Sandoval, R. Cumplido, C. Feregrino-Uribe, and I. Algredo-Badillo, "A compact FPGA-based microcoded coprocessor for exponentiation in asymmetric encryption," in *2017 IEEE 8th Latin American Symposium on Circuits Systems (LASCAS)*, pp. 1–4, Feb 2017.

[99] L. Rodríguez-Flores, M. Morales-Sandoval, R. Cumplido, C. Feregrino-Uribe, and I. Algredo-Badillo, "Compact FPGA hardware architecture for public key encryption in embedded devices," *PLOS ONE*, vol. 13, pp. 1–21, 01 2018.