

People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University M'Hamed BOUGARA – Boumerdès



Institute of Electrical and Electronic Engineering
Department of Electronics

Final Year Project Report Presented in Partial Fulfilment of
the Requirements for the Degree of

'Master'

In Electronics

Option: Computer Engineering

Title:

**Elliptic Curve Digital Signature System
Implementation on Embedded ARM
Processor**

Presented by:

BOURAHLA Amina
NEBHI Lynda

Supervisor:

Prof. KHOUAS Abdelhakim

Co-supervisor:

DR. OUDJIDA Abdelkrim Kamel

Registration Number:...../2023

Abstract

Elliptic Curve Digital Signature Algorithm (ECDSA) is a widely-used cryptographic algorithm that verifies the authenticity of digital messages. The main advantage of ECDSA over other signature algorithms is requiring smaller key sizes to achieve the same level of security. The smaller size of the key results in faster operations due to its employment in the most computationally heavy part of the ECDSA algorithm: the Elliptic Curve (EC) Point Multiplication (PM). Therefore, it is a target for optimization to achieve better speed, memory consumption, energy dissipation and security. Some approaches rely on hardware support such as the use of parallelism and more memory. Others, depend on a more efficient use of EC arithmetic as EC PM is built on EC operations: Addition (ADD), and Doubling (DBL). The Radix- 2^w method for EC PM relies on the recoding of the scalar k with fewer nonzero digits in a w -bit window serving to reduce the cost in terms of ADD operations used. This project focuses on the implementation of ECDSA using the Radix- 2^w method for EC point multiplication (PM) and Double Point Multiplication (DPM); DPM is the sum of two EC PMs. This implementation is realized in the context of national institute of standards and technology recommended binary ECs, and serves as proof of concept for the Radix- 2^w multiplication methods. The Zynq Evaluation and Development Board's processing system (PS) is used for the implementation as it allows for later integration of PL blocks for the Radix- 2^w multiplication making up a hardware/software solution. The project was carried out in three parts. First, the Radix- 2^w multiplication methods were implemented on computer, and their functionality validated. Subsequently, they were tested to reveal a 58.63% improvement in the cost of the Radix- 2^w EC PM method over the binary EC PM method in terms of ADDs, and a 47.509% improvement in the cost (in terms of ADDs) for the Radix- 2^w EC DPM method over the binary EC DPM method. Second, the ECDSA protocol was implemented on computer, along with an encryption protocol to complement the security provided by the signature (authentication and non repudiation). The reason behind this is that signature algorithms do not provide confidentiality. Thus, for the purpose of encryption, El-Gamal algorithm was used for its compatibility with ECs. Third and finally, the complete ECDSA/El-Gamal encryption protocol using Radix- 2^w methods for multiplication was adapted to the zedboard PS. This project resulted in the successful signature generation/verification and message encryption/decryption between the board and a computer using Radix- 2^w methods for multiplication.

Dedication

We would like to dedicate our effort to god first,
To our families, whose support we are immensely grateful for, whose patience, understanding,
and faith in us sustained us,
And to our friends who have been true companions in this endeavour.

Acknowledgement

We would like to express our sincere gratitude to our primary supervisor Professor A. KHOUAS, co-supervisor Director of Research A. K. OUDJIDA, and Researcher F. Nait-Abdesselam at The Center for the Development of Advanced Technologies, for their guidance, support, and valuable input throughout the duration of this project. Their expertise, patience, and constant encouragement have been instrumental in shaping our work and pushing us to strive for the better.

We would also like to extend our heartfelt appreciation to Ms. BENDAHMANE Meryem, a student our senior, and Mr. DAHMANI Abd El Madjid, a fellow student, for their assistance and valuable contributions to our project. Their knowledge, experience, and willingness to lend a helping hand have been truly valuable. Their insights and suggestions have helped in enhancing our work, and we are grateful for their collaborative spirit.

Finally, we would like to acknowledge our community at the Institute of Electrical & Electronic Engineering for fostering an environment conducive to learning and growth.

Table of Contents

Abstract	i
Dedication	ii
Acknowledgement	iii
List of Figures	vi
List of Tables	viii
List of Algorithms	ix
List of Abbreviations	x
General Introduction	1
1 Elliptic Curves	4
1.1 Introduction	4
1.2 Finite Fields	5
1.3 Binary Field Arithmetic	6
1.3.1 Binary Field Addition	6
1.3.2 Binary Field Multiplication	7
1.3.3 Binary Field Squaring	7
1.3.4 Binary Field Inversion	8
1.4 Elliptic Curve Group Law	9
1.5 Elliptic Curve Coordinate Systems	11
1.5.1 Projective Jacobian Coordinate System	11
1.5.2 Projective Lopez-Dahab Coordinate System	13
1.5.3 Coordinate Systems Comparison	13
1.6 Elliptic Curve Point Multiplication	14
1.7 Radix- 2^w Elliptic Curve Multiplication	14
1.7.1 Radix- 2^w Recoding	15
1.7.2 Radix- 2^w Elliptic Curve Point Multiplication Method	17
1.7.3 Radix- 2^w Elliptic Curve Double Point Multiplication Method	18
1.8 Conclusion	19
2 Elliptic Curve Cryptography	20
2.1 Introduction	20
2.2 Cryptographic Model	20

2.2.1	Symmetric-key Cryptography	21
2.2.2	Public-key Cryptography	22
2.3	Elliptic Curve Digital Signature Algorithm	23
2.3.1	Signature Generation	23
2.3.2	Signature Verification	25
2.4	Encryption Schemes	26
2.4.1	Joint Signature and Encryption System	27
2.4.2	EC-ElGamal Encryption Protocol	27
2.5	Conclusion	29
3	ECDSA System Design using Radix-2^w EC multiplication methods	30
3.1	Introduction	30
3.2	Radix- 2^w Elliptic Curve Point Multiplication Design	30
3.2.1	Elliptic Curve Doubling and Addition	31
3.2.2	Elliptic Curve Multiplication using Radix- 2^w Methods	33
3.2.3	Binary Method for Elliptic Curve Point Multiplication	37
3.3	Cryptographic System Design	38
3.4	Conclusion	39
4	EC Signature/Encryption System Implementation and Results	40
4.1	Introduction	40
4.2	Hardware Environment	41
4.2.1	Zynq Evaluation and Development Board	41
4.2.2	Development Tools	42
4.3	Software System Implementation	43
4.3.1	Global System Overview	43
4.3.2	Scalars and Points Representation	45
4.4	Radix 2^w Elliptic Curve Multiplication Implementation	47
4.4.1	Elliptic Curve Addition and Doubling Implementation	47
4.4.2	Elliptic Curve Point Multiplication Implementation	47
4.5	Elliptic Curve Multiplication Blocks Tests	52
4.5.1	Functionality Validation	52
4.5.2	Elliptic Curve Point Multiplication Comparison	54
4.6	Elliptic Curve based Cryptographic Protocol Implementation	56
4.6.1	Public-Key Implementation	56
4.6.2	ElGamal Elliptic Curve Encryption Implementation	56
4.6.3	Elliptic Curve Digital Signature Implementation	57
4.7	Joint Elliptic Curve Cryptographic Protocols Implementation	57
4.8	System Implementation on the ZedBoard	59
4.8.1	File Manipulation on the ZedBoard	59
	General Conclusion	61
	Bibliography	63

List of Figures

1.1	Point G multiples from 1 to 5 on an elliptic curve [1].	4
1.2	Pyramid showing the dependencies between field operations, elliptic curve operations and elliptic curve cryptographic protocols.	5
1.3	Squaring in \mathbb{F}_{2^m} using a polynomial-basis representation [2].	8
1.4	Geometric addition and doubling of EC point [3].	10
1.5	Decomposition of $k = (5892973)_{10}$ in Radix 2^4 [4].	17
2.1	Cryptographic Model.	21
2.2	Signature generation process ($m =$ message, $e =$ hash digest, $d =$ private key, $k =$ nonce, $(r, s) =$ signature components).	24
2.3	Signature verification process($m =$ message, $e =$ hash digest, $Q =$ public key, $(r, s) =$ signature components).	25
2.4	Joint ECDSA / Encryption Protocol workflow.	27
2.5	Mapping and encryption process for a single character.	28
3.1	Elliptic curve operations and their dependencies.	31
3.2	Block diagram of the Radix- 2^w EC PM precomputations.	35
3.3	Design of the block that retrieves a slice Q.	35
3.4	Block diagram of the Radix- 2^w EC DPM precomputations.	37
3.5	Block diagram of the joint EC cryptographic system.	39
4.1	ZedBoard layout[5].	42
4.2	Global system layout.	44
4.3	Result of the cross-compilation of GNU MP library for ARM Cortex-A9 processor.	46
4.4	B-283 NIST-recommended binary elliptic curve parameters [6].	46
4.5	Block diagram of the simplified Radix- 2^w EC PM.	48
4.6	Radix-w recoding test bench.	49
4.7	Block diagram of the Radix- 2^w EC PM.	50
4.8	Radix- 2^w DPM recoding test bench.	51
4.9	Block diagram of the Radix- 2^w EC DPM.	52
4.10	Radix- 2^w EC PM implementation validation flowchart.	53
4.11	EC PM validation test flowchart.	54
4.12	Diagram for comparison of binary and Radix 2-w methods for EC PM in terms of cost in ADDs.	54
4.13	Flowchart showing the user interface implemented to access the cryptographic protocols.	58
4.14	Produced signature of a message " <i>MSG_SIG.TXT</i> ".	58
4.15	Produced encryption of a message " <i>ENC_MSG.TXT</i> ".	59
4.16	Produced decryption of an encrypted message " <i>DEC_MSG.TXT</i> ".	59

4.17 Result of the example decryption and signature verification from the TUI view of software implementation.	59
--	----

List of Tables

1.1	Smallest number of multiplications for inversions over NIST fields and their corresponding algorithms [7].	11
1.2	$GF(2^m)$ operation count and number of temporary variables (finite field elements) for EC point addition and doubling (M = Multiplications, I = Inversions, T = number of Temporary Variables)[8].	14
3.1	Radix- 2^w configuration parameters for NIST recommended binary field elliptic curves [4].	34
4.1	Radix- 2^3 recoding set look up table.	50
4.2	Average cost of EC PM for each multiplication method in terms of EC ADDs for each NIST-recommended binary curve.	55
4.3	Average cost of EC DPM for each multiplication method in terms of EC ADDs for each NIST-recommended binary curve.	55
4.4	Comparative percentage for cost of operations.	56
4.5	Lookup table for encryption in ECC.	57

List of Algorithms

1.1	Addition in $\mathbb{F}_{2^m}[3]$	6
1.2	Right-to-left shift-and-add field multiplication in $\mathbb{F}_{2^m}[3]$	7
1.3	Binary algorithm for inversion in $\mathbb{F}_{2^m}[3]$	9
1.4	Computation of m_i and n_i	16
1.5	Radix- 2^w Point Multiplication [4].	18
1.6	Simultaneous Radix- 2^2 Double Point Multiplication[9].	19
2.1	Elliptic Curve Key Pair Generation	24
2.2	ECDSA signature generation	25
2.3	ECDSA signature verification	26
2.4	Encryption of single character	28
2.5	Decryption of single character	29
3.1	Point doubling ($y^2 + xy = x^3 + ax^2 + b, a \in 0,1$, LD coordinates)	32
3.2	Point addition ($y^2 + xy = x^3 + ax^2 + b, a \in 0,1$, LD-Affine coordinates)	33
3.3	Left-to-Right binary method for point multiplication	37
3.4	Right-to-Left binary method for point multiplication	38

List of Abbreviations

ADD: Point Addition
AP: All Programmable
ARM: Advanced Reduced Instruction Set Computer Machine
AXI: Advanced eXtensible Interface
DB: Double-base
DBL: Point Doubling
DDR3: Double Data Rate 3
DLP: Discrete Logarithm Problem
DPM: Double Point Multiplication
DSA: Digital Signature Algorithm
EC: Elliptic Curve
ECC: Elliptic Curve Cryptography
ECDSA: Elliptic Curve Digital Signature Algorithm
FAT: File Allocation Table
FATFS: File Allocation Table File System
FPGA: Field-Programmable Gate Array
GMP: GNU Multiple Precision
HMAC: Hash Message Authentication Code
HDL: Hardware Description Language
HLS: High Level Synthesis
IDE: Integrated Development Environment
IFP: Integer Factorization Problem
* IP: Intellectual Property
IT: Itoh-Tsujii
JTAG: Joint Test Action Group
LD: Lopez-Dahab
LSB: Least Significant Bit
MAC: Message Authentication Code
MNAF: Modified Non-Adjacent Form
* MOF: Mutual Opposite Form
* MSB: Most Significant Bit
NAF: Non-Adjacent Form
NIST: National Institute of Standards and Technology
PC: Personal Computer
PM: Point Multiplication
PL: Programmable Logic
PS: Processing System
RNG: Random Number Generator
RSA: Rivest, Shamir, and Adleman Algorithm
SD: Secure Digital
SDK: Software Development Kit
SDRAM: Synchronous Dynamic Random-Access Memory
SoC: System On Chip
TB: Triple-base
TUI: Terminal User Interface
* UART: Universal Asynchronous Receiver / Transmitter
USB: Universal Serial Bus

w-NAF: Window Non-Adjacent Form
XilFFS: Xilinx File System
ZedBoard: Zynq Evaluation and Development Board

General Introduction

Communication's important role in the progress of civilization led to the development of various methods for transmitting and receiving information. A wide range of communication practices emerged across time: the inscription on clay tablets for correspondence, the posting of letters, and most recently, the modern-day exchange of textual messages via mobile devices. However, these communication methods were plagued by a number of security concerns regarding privacy, fraudulent messages, and the such. In response, some practices that help ensure confidentiality and authenticity came into being. In letters, for example, envelopes can be sealed, letters signed, and messages coded. This demand for security remained unchanged despite the change in communication mediums.

Cryptography is a field of study that employs mathematical algorithms to keep a piece of information secure against unwanted interventions. However, the communication protocols emerging from the field of cryptography are subject to breaches provided suitable mathematical tools and computational power. In the context of modern day digital communication, cryptography is concerned with the conception of protocols applicable to machine computational power. The growing memory and speed capacities of machines entail the continual optimization and creation of cryptographic algorithms satisfying the security, data protection, and information integrity communication requirements.

In line with this need for secure digital communication, digital signatures come into play. Similar to traditional signatures on letters, digital signatures serve as means of authentication through the verification of: the message's integrity, and the identity of its sender. To achieve this aspect of security, a number of cryptographic protocols were developed. The National Institute of Standards and Technology (NIST) approved digital signature techniques are the Rivest, Shamir, and Adleman (RSA) algorithm, the Digital Signature Algorithm (DSA), and the Elliptic Curve Digital Signature Algorithm (ECDSA)[10]. RSA is widely supported, offers strong security, and is versatile in its functionality. DSA requires shorter key lengths, resulting in faster computations and efficient signature verification. ECDSA provides robust security with even shorter key lengths, making it suitable for resource-constrained environments and offering faster computations[11]. Each algorithm has its advantages, and the choice depends on factors such as security requirements and performance considerations. The distinguishing feature underlying each digital signature algorithm from the rest is the mathematical problem (typically a number theory problem) that is posed by the signature and solved by the verification.

Digital signature cryptographic schemes rely on the use of "keys" in their operation. Keys are sizable integer numbers used to achieve security. They are used in the computations of both signature generation and verification, and their size positively correlates with the level of security provided. However, keys of the same size do not guarantee equal security across different algorithms. ECDSA, for example, relies on elliptic curve (EC) mathematical properties to reduce the

size of the key while maintaining the level of security resulting in faster operations[12]. Optimizations for speed, memory, etc., go beyond just minimizing key sizes, but extend to enhancing the computational aspects of the signature algorithms.

EC point multiplication (PM) is used in the ECDSA and is considered to be the most costly operation of the protocol making it a target for optimization. It involves the multiplication of a point P belonging to an EC by a scalar k . The addition of two or more PMs being commonly required in cryptographic algorithms, Double Point Multiplication (DPM) and Multiple Point Multiplication, gave rise to further optimizations in EC Multiplication[9][13][14]. Basic EC point operations: point addition (ADD) and point doubling (DBL) are used to evaluate the multiplications. The double-and-add technique is the traditional binary algorithm in which the scalar k is represented as a binary number and the product is evaluated through accumulation. The point P is added to an accumulator based on the value of each bit in k , followed by a doubling of the accumulator. A more efficient technique used to reduce the overall cost of the multiplication in terms of ADDs and DBLs is the recoding of the scalar k . A well used method for computing ECPM that employs the recoding of k is the windowed Non-Adjacent Form (w-NAF). However, It achieves cost reduction at the expense of memory consumption in the form of intermediate variables and carries[3]. The Radix- 2^w method for EC multiplications overcomes the drawbacks of the w-NAF method using a different recoding of k . It eliminates the need for intermediate variables and carries using a minimal amount of precomputation, while reducing the cost of ADDs to a near-optimal limit in a sub-linear computational time without increasing the number of DBLs[4].

The primary objective of this project is to implement the ECDSA using the Radix- 2^w method for EC PM and DPM on ARM processor. The use of the Radix- 2^w method for recoding in the EC multiplications is central to this project since our aim is to provide its proof of concept. Since ECDSA does not target the confidentiality aspect of security, it will be implemented along with an encryption protocol to complement it. Due to its adaptability to elliptic curves, ElGamal Encryption algorithm will be used.

The implementation will be two-fold. First, the software program for the ECDSA/El-Gamal cryptographic system will be written to run on personal computers (PC) as it simplifies the debugging and the subsequent functionality testing of the different software blocks. Second, the resulting functional program will be adapted to the the processing system of the Zynq Evaluation and Development Board (ZedBoard). The end goal is to provide the cryptographic system software implementation into which hardware blocks for Radix- 2^w multiplications implemented on a Field Programmable Gate Array (FPGA) can be integrated.

The report is organized as follows. The first chapter introduces and explains the different mathematical prerequisites necessary for ECC. It starts by describing finite fields and their underlying arithmetic. Then, it delves into the basic EC operations, EC PM, and finally EC DPM. The second chapter deals with all ECC concepts relevant to this project. First, it introduces the fundamental concepts of cryptography and its types. Then, it reviews the ECDSA, and the steps involved in achieving signature generation, and verification. Finally, it provides a description of the encryption scheme protocol, and discusses the aspect of joining the ECDSA with the encryption protocol. Chapter three establishes the design of the Radix- 2^w EC multiplications and its underlying EC and finite field operations. Then, it delves into the details of building the cryptographic protocols (ECDSA and ElGamal encryption) The fourth chapter deals with the details of the entire EC cryptographic system software implementation. This includes descriptions of the libraries, languages, tools used throughout the implementation, as well as some problems faced and the solutions used to overcome them. It provides the details for the Radix- $2w$ EC multiplications

implementation. Subsequently, it describes the methods used for validating the implementation of the EC multiplications. Then, it showcases the results obtained from the comparison of the Radix- $2w$ EC PM/DPM and the binary EC PM. Finally, it presents some details on the ECDSA and ElGamal encryption system's implementation along with an overview of the Terminal User Interface (TUI) used. An example where both encryption and signing are used is given to illustrate the usage of the system on personal computers. Following this, it gives an overview on the hardware and the development tools included in the ZedBoard implementation along with a description of the necessary adaptations made to the software to enable it to run on the board's processing system. At the end, the report conclusion is presented, where the project results and recommendations for future improvements are discussed.

Chapter 1

Elliptic Curves

1.1 Introduction

Cryptography is built upon the idea of one-way (trapdoor) functions. They are characterized by straight-forward computation in one direction, and difficult reverse computation. An example of one way functions is the modulo function. The result of modulo a number n operation (denoted $\%n$) is equal to the remainder of the division over n . Thus, computation of 7 modulo 5 is simple and gives 2. However, given 2 as the result, all of 7, 12, 17, 22, etc, modulo 5 are possible solutions to the inverse-modulo. Therefore, retrieving 7 from the result and divisor alone is difficult.

The use of elliptic curves in cryptography also revolves around the concept of one-way-functions. However, instead of using scalars for the operations (as shown in the previous modulo example), elliptic curves operate on points $G(x, y)$ where x and y are coordinates of the point. The most important operation to elliptic curve cryptography is the elliptic curve point multiplication, where a point G belonging to an elliptic curve E is multiplied by a scalar k . Figure 1.1 shows an example of a point G represented on an elliptic curve, along with some of its multiples: $2xG$, $3xG, \dots, 5xG$. The elliptic curve point multiplication sets the foundation for elliptic curve based cryptographic systems.

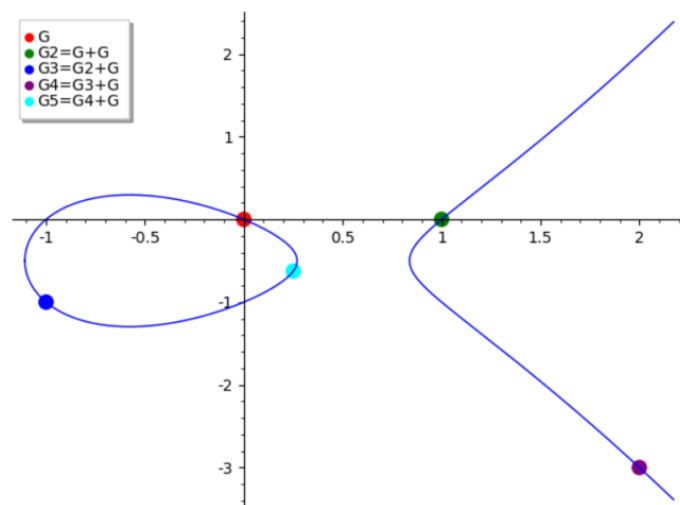


Figure 1.1: Point G multiples from 1 to 5 on an elliptic curve [1].

The elliptic curve point multiplication is a complex operation and is built atop other mathematical operations. The hierarchy depicted in Figure 1.2 shows the different layers necessary for understanding and implementing elliptic curve cryptographic systems.

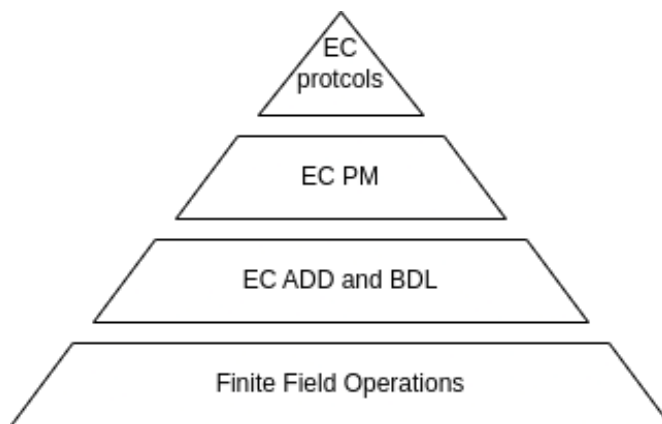


Figure 1.2: Pyramid showing the dependencies between field operations, elliptic curve operations and elliptic curve cryptographic protocols.

In this chapter, we will be introducing the necessary mathematical background needed for implementing the fundamental EC operations, and subsequently the EC PM. Finite fields are a necessary prerequisite for EC operations since ECs are defined over them. therefore, we will start by introducing them. Subsequently, We will delve into the basic EC operations: addition and doubling. Finally, we will review EC PM along with some methods to optimize this operation.

1.2 Finite Fields

A field $(\mathbb{F}, +, \cdot)$ consists of a set \mathbb{F} along with two operations: addition $(+)$, and multiplication (\cdot) . A given field must satisfy nine conditions called field axioms:

1. **Associativity of addition.**
2. **Existence of additive identity:** Denoted 0.
3. **Existence of additive inverses:** Any element x is invertible. The additive inverse is unique and denoted $-x$.
4. **Commutativity of multiplication.**
5. **Associativity of multiplication.**
6. **Existence of multiplicative identity:** Denoted 1.
7. **Existence of multiplicative inverses:** Any element x , except possibly for 0, is invertible for (\cdot) . The multiplicative inverse is unique, and denoted x^{-1} .
8. **Distributive law:** For all x_1, x_2, x_3 in the field: $x_1 \cdot (x_2 + x_3) = (x_1 \cdot x_2) + (x_1 \cdot x_3)$.
9. **Zero-one law:** The additive identity and multiplicative identity are distinct.

It is customary to call a field $(\mathbb{F}, +, \cdot)$, simply, the field \mathbb{F} . If the set \mathbb{F} is finite, then the field is said to be finite. The size q of a field \mathbb{F} , which is the number of elements present in the field, is denoted as $q = p^m$. Here, p is a prime number called the characteristic of \mathbb{F} , and m is a positive

integer. There are different types of finite fields, two being prime and extension fields; where, if $m = 1$, then \mathbb{F} is called a prime field, and if $m \geq 2$, then \mathbb{F} is called an extension field. A special case of extension fields are binary fields having $p = 2$ and $m \geq 2$. It is worth noting that, since the size q of a binary field is equal to 2^m , the bit-length¹ of an element belonging to \mathbb{F}_{2^m} does not exceed m .

In prime finite fields, the addition and multiplication operations are performed modulo p . However, in extension fields, operations are performed modulo a primitive element (or generator). The latter is an element that generates a multiplicative subgroup of nonzero elements in the field. More formally, let \mathbb{F}_q be a finite field with q elements. An element $\alpha \in \mathbb{F}_q$ is called a primitive element if the powers $\alpha^0, \alpha^1, \alpha^2, \dots, \alpha^{q-2}$ are all distinct and all nonzero elements of \mathbb{F}_q are generated by an expression these powers.

1.3 Binary Field Arithmetic

\mathbb{F}_{2^m} elements are represented as binary polynomials (polynomials with coefficients 0 or 1). In a field, there are 2^m such polynomials where the degree of each polynomial does not exceed $m-1$. Therefore, these elements can be represented as m -bit strings. An element x is represented as $x(z) = x_{m-1} \times z^{m-1} + x_{m-2} \times z^{m-2} + \dots + x_1 \times z^1 + x_0 \times z^0$ with z being the base, in this case $z = 2$. Operations on binary fields are defined using this polynomial representation, and are performed modulo a primitive polynomial (similar to a primitive element). The latter is an irreducible polynomial that generates the field itself. In other words, it is a polynomial that cannot be factored into lower-degree polynomials within the field, and its degree determines the size of the field. Modulo in binary field operations involves reducing the result of said operations by the field's primitive polynomial. This entails dividing the result of an operation by the primitive polynomial, using polynomial division, and keeping the remainder as the reduced result. The purpose of this process is to keep the result within the field size.

1.3.1 Binary Field Addition

As shown in Algorithm 1.1, the addition of two elements $a(z), b(z)$ of a binary field is equivalent to a bit-wise XOR operation. Consider two binary polynomials, $a(z)$ and $b(z)$, of degree at most $m - 1$ in the field \mathbb{F}_{2^m} . For each degree i , the corresponding coefficients in $a(z)$ and $b(z)$ are added. This addition is performed by applying the XOR operation to said coefficients. This allows for a carry free addition. The resulting polynomial $c(z)$ represents the sum of $a(z)$ and $b(z)$. Subtraction in this field is also achieved through the same bit-wise XOR operation.

Algorithm 1.1 Addition in $\mathbb{F}_{2^m}[3]$

Input: Binary polynomials $a(z)$ and $b(z)$ of degree at most $m-1$

Output: $c(z) = a(z) + b(z)$

- 1: **for** i **from** 0 **to** $m-1$ **do**
 - 2: $c_i \leftarrow a_i \oplus b_i$
 - 3: **Return** ($c(z)$)
-

¹**bit-length:** number of digits in the binary representation of a number.

1.3.2 Binary Field Multiplication

The multiplication of two elements of a binary field is performed through a combination of addition and left-shifts. Assuming a binary polynomial $f(z)$ is the primitive polynomial (primitive element), the multiplication of binary polynomials $a(z)$ and $b(z)$ is done according to Algorithm 1.2. This algorithm performs multiplication in the binary field by iteratively multiplying and adding polynomials based on the coefficients of the input polynomials $a(z)$ and $b(z)$. First, the lowest degree term in polynomial $a(z)$ is checked, and the product polynomial $c(z)$ is initialized accordingly. If the rightmost coefficient a_0 of $a(z)$ is 1, the product polynomial $c(z)$ is set as $b(z)$. If not, then it is set as 0. Next, a left-shift operation is performed on polynomial $b(z)$. This is achieved by multiplying it with z and taking the result modulo $f(z)$. Therefore, shifting the coefficients to the left by one position. Then, if the coefficient a_i of polynomial $a(z)$ is 1, the modified polynomial $b(z)$ is added to the product polynomial $c(z)$. This process is repeated for all $a(z)$ coefficients. The product polynomial $c(z)$ is then returned as the result of the binary field multiplication. The modulo operation with the primitive polynomial ensures that the resulting product polynomial $c(z)$ remains within the specified field.

Algorithm 1.2 Right-to-left shift-and-add field multiplication in \mathbb{F}_{2^m} [3]

Input: Binary polynomials $a(z)$ and $b(z)$ of degree at most $m-1$

Output: $c(z) = a(z) \cdot b(z) \pmod{f(z)}$

```

1: if  $a_0 = 1$  then  $c(z) \leftarrow b(z)$ 
2: else  $c(z) \leftarrow 0$ 
3:   for  $i$  from 1 to  $m-1$  do
4:      $b(z) \leftarrow b(z) \cdot z \pmod{f(z)}$ 
5:     if  $a_i = 1$  then  $c(z) \leftarrow c(z) + b(z)$ 
6: Return ( $c(z)$ )

```

1.3.3 Binary Field Squaring

Squaring an element of this field can be done through multiplication. This is equivalent to multiplying the element by itself. However, this operation can be optimized by inserting zeroes between all coefficients of the element, as shown in Figure 1.3.

To understand how this optimization works, let's consider a binary field element represented by the following polynomial:

$$A(z) = a_{m-1} \times z^{m-1} + a_{m-2} \times z^{m-2} + \dots + a_1 \times z + a_0$$

When squaring this element, we need to multiply it by itself:

$$A(z)^2 = (a_{m-1} \times z^{m-1} + a_{m-2} \times z^{m-2} + \dots + a_1 \times z + a_0) \cdot (a_{m-1} \times z^{m-1} + a_{m-2} \times z^{m-2} + \dots + a_1 \times z + a_0)$$

Expanding this multiplication, we get:

$$A(z)^2 = a_0 \times a_0 + (a_0 \times a_1 + a_1 \times a_0) \times z + (a_0 \times a_2 + a_1 \times a_1 + a_2 \times a_0) \times z^2 + \dots$$

Here, the cross-product terms $a_i \times a_j$ and $a_j \times a_i$, arise from multiplying the coefficients from different terms together. These cross-product terms introduce additional terms and complexity in the expanded polynomial. However, by inserting zeroes between the coefficients before the squaring, the cross-product terms are eliminated. The resulting polynomial after squaring only

includes terms with products of coefficients from the same term. This results in a more efficient and concise representation of the squared polynomial.

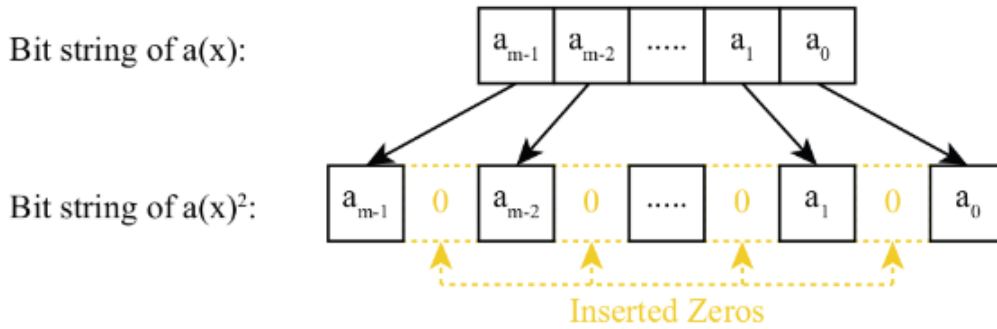


Figure 1.3: Squaring in \mathbb{F}_{2^m} using a polynomial-basis representation [2].

1.3.4 Binary Field Inversion

Division by an element is performed through a multiplication by the inverse of that element. The inverse of an element is obtained using the Algorithm 1.3 for binary inversion, where the modular inverse of a nonzero binary polynomial $a(z)$ modulo another polynomial $f(z)$ is calculated.

The algorithm begins by initializing variables u and v : u is set to the input polynomial $a(z)$, and v is set to the primitive polynomial $f(z)$. The variables g_1 and g_2 are then initialized: g_1 , an intermediate variable used in the subsequent calculations, is set to 1; and g_2 , representing the resulting inverse of $a(z)$ modulo $f(z)$, is set to 0. The algorithm then enters a loop that continues until either u or v becomes equal to 1. Within this loop, there are two more inner loops. The first inner loop divides u by the variable z as long as z divides u . Depending on whether z divides g_1 , it adjusts g_1 accordingly, where: if z divides g_1 , z is removed from it. If it does not, g_1 is updated by adding $f(z)$ and dividing the result by z . The same process is also applied on the second inner loop operating on the variable v and g_2 . After each pass through the two inner loops, the algorithm checks the degrees of u and v . If the degree of u is greater than that of v , it updates u and g_1 by adding v and g_2 to them, respectively. Otherwise, it updates v and g_2 by adding u and g_1 to them, respectively. Once the outer loop finishes, the algorithm checks if u is equal to 1. If so, it assigns g_2 the value of g_1 . Finally, the algorithm returns the computed value of g_2 , which represents the modular inverse of $a(z)$ modulo f .

Algorithm 1.3 Binary algorithm for inversion in $\mathbb{F}_{2^m}[3]$

Input: A nonzero binary polynomial $a(z)$ of degree at most $m-1$
Output: $g_2 = a(z)^{-1} \bmod f$

- 1: $u \leftarrow a(z), v \leftarrow f(z).$
- 2: $g_1 \leftarrow 1, g_2 \leftarrow 0.$
- 3: **while** $u \neq 1$ and $v \neq 1$ **do**
- 4: **while** z divides u **do**
- 5: $u \leftarrow u/z.$
- 6: **if** z divides g_1 **then**
- 7: $g_1 \leftarrow g_1/z.$
- 8: **else**
- 9: $g_1 \leftarrow (g_1 + f(z))/z.$
- 10: **while** z divides v **do**
- 11: $v \leftarrow v/z.$
- 12: **if** z divides g_2 **then**
- 13: $g_2 \leftarrow g_2/z.$
- 14: **else**
- 15: $g_2 \leftarrow (g_2 + f(z))/z.$
- 16: **if** $\deg(u) > \deg(v)$ **then**
- 17: $u \leftarrow u + v.$
- 18: $g_1 \leftarrow g_1 + g_2.$
- 19: **else**
- 20: $v \leftarrow u + v.$
- 21: $g_2 \leftarrow g_1 + g_2.$
- 22: **if** $u=1$ **then**
- 23: $g_2 \leftarrow g_1.$
- 24: **Return** $g_2.$

1.4 Elliptic Curve Group Law

An elliptic curve E is a set of points $P(x, y)$ defined over a field \mathbb{F} . $E(\mathbb{F})$ satisfies the Weierstrass equation shown in eq.(1.1). However, in cryptographic applications, eq.(1.2) is generally used. It is a simplified Weierstrass equation. It requires for the discriminant² $\Delta = -16(4a^3 + 27b^2)$ to be nonzero, and for the polynomial $x^3 + ax + b$ to have distinct roots. The National Institute of Standards and Technology (NIST) recommends, in document [6], a set of ECs to be used in cryptography.

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \quad (1.1)$$

$$E : y^2 = x^3 + ax + b \quad (1.2)$$

²A mathematical quantity that determines the properties of an elliptic curve and helps classify them into different types based on its value

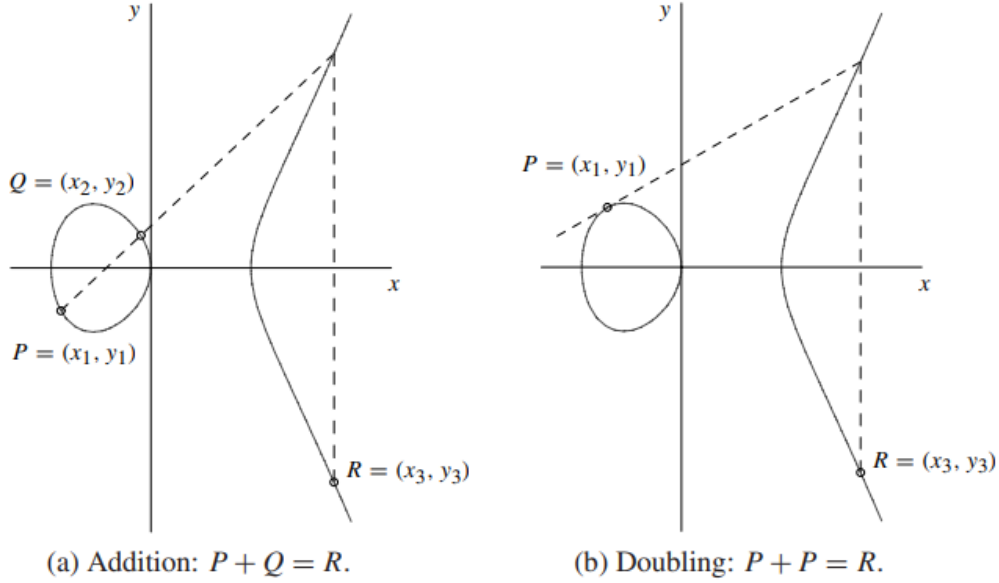


Figure 1.4: Geometric addition and doubling of EC point [3].

An EC $E(\mathbb{F})$ is naturally a group and its group law³ is constructed geometrically. Figure 1.4 shows the geometric definition of addition and doubling, of point elements, in this group. Addition is defined as the negative of the point at which the line that passes by both points being added intersects with the curve. Since doubling is the addition of a point to itself, it is defined similarly: with the two points being added converging to the same point, the line drawn is the tangent to this point. Each EC has a generator point G from which the entire set of points belonging to $E(\mathbb{F})$ can be generated using the EC defined operations. The group law for the curve $E(\mathbb{F})$ is as follows:

1. **Identity:** $P + \infty = \infty + P = P$ for all $P \in E(\mathbb{F})$.
2. **Negatives:** The negative of a point $P(x,y)$ is a point $-P$ with coordinates $(x,-y)$ in prime fields. In ECs defined over binary fields, however, the negative is $-P$ with coordinates $(x,x+y)$. The sum of P and its negative gives the point at infinity.
3. **Point Addition (ADD):** For two points $P(x_1, y_1)$ and $Q(x_2, y_2)$ on the curve, addition results in a point $R(x_3, y_3)$ such that:

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1} \right)^2 - x_1 - x_2 \quad (1.3)$$

$$y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1} \right) (x_1 - x_3) - y_1 \quad (1.4)$$

4. **Point Doubling (DBL):** For a point $P(x_1, y_1)$ on the curve, doubling results in a point $2 \times P = R(x_3, y_3)$

$$x_3 = \left(\frac{3 \times x_1^2 + a}{2 \times y_1} \right)^2 - 2 \times x_1 \quad (1.5)$$

$$y_3 = \left(\frac{3 \times x_1^2 + a}{2 \times y_1} \right) (x_1 - x_3) - y_1 \quad (1.6)$$

³Group law refers to the operations defined on a group. It specifies how elements of the set combine or interact with each other under the operations.

In the algebraic definitions for ADD and DBL, the Affine coordinate system is used to represent a point P as $P(x, y)$. The problem with the Affine coordinate point representation is the use of division in the EC ADD and DBL operations. Since division requires inversion, which is a costly operation, implementing the ADD and DBL as defined in Affine can be expensive. Table 1.1 shows the cost of the field inversion operation in term of multiplications for different binary fields. The inversion costs were calculated according to the binary inversion algorithms: double-base (DB), Itoh-Tsujii (IT), triple-base (TB) algorithms. Due to the high cost of inversion, different coordinate representations were developed to minimize the use of this field operation.

Field	Multiplication	Algorithms
$\mathbb{F}(2^{163})$	9	IT / DB / TB
$\mathbb{F}(2^{233})$	10	IT / DB / TB
$\mathbb{F}(2^{283})$	11	IT
$\mathbb{F}(2^{409})$	10	DB / TB
$\mathbb{F}(2^{571})$	12	TB

Table 1.1: *Smallest number of multiplications for inversions over NIST fields and their corresponding algorithms [7].*

1.5 Elliptic Curve Coordinate Systems

A number of Projective coordinate systems were developed with the purpose of optimizing EC arithmetic for ADD and DBL through minimizing the use of the costly field inversion. They are alternative representations for points on an EC and an extension of the more commonly used Affine coordinate system. In a Projective coordinate system, a point P is represented using three coordinates: X , Y , and Z . The conversion between $P(X, Y, Z)$ and $P(x, y)$ depends on the type of Projective coordinates. In this section, we will be looking at Projective Jacobian and Lopez-Dahab (LD) coordinate systems as they are the most efficient and commonly used Projective coordinates. We will be assuming a simplified EC of eq.(1.2) defined over a binary finite field.

1.5.1 Projective Jacobian Coordinate System

In Projective Jacobian coordinate system, $P(X, Y, Z)$ with $Z \neq 0$, corresponds to the Affine $P(x, y) = (X/Z^2, Y/Z^3)$. This implies that for $Z = 1$, $x = X$ and $y = Y$. Therefore, $P(x, y)$ in Affine coordinates automatically corresponds to $P(X, Y, 1)$ in Projective Jacobian coordinates, producing a 1-1 correspondence between the two coordinate systems for a fixed Z . In Projective Jacobian coordinate system, the point at infinity corresponds to $(1, 1, 0)$, while the negative of (X, Y, Z) is $(X, X + Y, Z)$. The equations for ADD and DBL can be obtained by replacing x by X/Z^2 and y by Y/Z^3 in eqs.(1.3), (1.4), (1.5), and (1.6). For the doubling of a point $P(X_1, Y_1, Z_1)$, eqs.(1.7), and (1.8) are obtained for a resulting point $R'(X'_3, Y'_3, Z'_3)$, simplifying to eqs.(1.9), and (1.10).

$$X'_3 = \left(\frac{3 \times (X_1/Z_1^2)^2 + a}{2 \times (Y_1/Z_1^3)} \right)^2 - 2 \times \left(\frac{X_1}{Z_1^2} \right) \quad (1.7)$$

$$Y'_3 = \left(\frac{3 \times (X_1/Z_1^2)^2 + a}{2 \times (Y_1/Z_1^3)} \right) \left(\frac{X_1}{Z_1^2} - X_3 \right) - \left(\frac{Y_1}{Z_1^3} \right) \quad (1.8)$$

$$X'_3 = \frac{(3X_1^2 + aZ_1^4)^2 - 8X_1Y_1^2}{4Y_1^2Z_1^2} \quad (1.9)$$

$$Y'_3 = \left(\frac{3X_1^2 + aZ_1^4}{2Y_1Z_1} \right) \left(\frac{X_1}{Z_1^2} - X'_3 \right) - \frac{Y_1}{Z_1^3} \quad (1.10)$$

In order to eliminate inversion, the values of the coordinates of $2P(X_3, Y_3, Z_3)$ are set to $X_3 = X'_3 \cdot Z_3^2$ and $Y_3 = Y'_3 \cdot Z_3^3$ where $Z_3 = 2Y_1Z_1$. The resulting set of eqs.(1.11),(1.12), and (1.13) define the DBL operation in the Projective Jacobian coordinate system.

$$X_3 = (3X_1^2 + aZ_1^4)2 - 8X_1Y_1^2. \quad (1.11)$$

$$Y_3 = (3X_1 + aZ_1)(4X_1Y_1 - X_3) - 8Y_1^4. \quad (1.12)$$

$$Z_3 = 2Y_1Z_1. \quad (1.13)$$

By storing some intermediate elements, X_3 , Y_3 and Z_3 can be computed using six field squarings and four field multiplications: $A \leftarrow Y_1^2$, $B \leftarrow 4X_1 \cdot A$, $X_3 \leftarrow D^2 - 2B$, $C \leftarrow 8A^2$, $D \leftarrow 3X_1^2 + a \cdot Z_1^4$, $Y_3 \leftarrow D \cdot (B - X_3) - C$, $Z_3 \leftarrow 2Y_1 \cdot Z_1$.

Similar steps are followed for the ADD operation. For the addition of two points $P(X_1, Y_1, Z_1)$ and $Q(X_2, Y_2, Z_2)$, the point $R(X_3, Y_3, Z_3)$ is obtained as shown in eqs.(1.14), (1.15), and (1.16).

$$X_3 = (X_1Y_2 + X_2Y_1)(Y_1Y_2 - a(X_1Z_2 + X_2Z_1) - 3bZ_1Z_2) \quad (1.14)$$

$$- (Y_1Z_2 + Y_2Z_1)(aX_1X_2 + 3b(X_1Z_2 + X_2Z_1) - a^2Z_1Z_2)$$

$$Y_3 = (Y_1Y_2 + a(X_1Z_2 + X_2Z_1) + 3bZ_1Z_2)(Y_1Y_2 - a(X_1Z_2 + X_2Z_1) - 3bZ_1Z_2) \quad (1.15)$$

$$+ (3X_1X_2 + aZ_1Z_2)(aX_1X_2 + 3b(X_1Z_2 + X_2Z_1) - a^2Z_1Z_2)$$

$$Z_3 = (Y_1Z_2 + Y_2Z_1)(Y_1Y_2 + a(X_1Z_2 + X_2Z_1) + 3bZ_1Z_2) \quad (1.16)$$

$$+ (X_1Y_2 + X_2Y_1)(3X_1X_2 + aZ_1Z_2)$$

Although this Projective coordinate system successfully avoids field inversions, it does not optimally optimize the overall cost of field operations. By introducing mixed coordinates in the ADD operation, representing one point in Affine coordinates (x, y) and the other point in Projective coordinates (X, Y, Z) , a balance between the advantages of Affine and Projective coordinate systems can be achieved. The addition of two points $(P(X_1, Y_1, Z_1)$ in Jacobian Projective coordinates, and $Q(X_2, Y_2, 1)$ with (X_2, Y_2) being the Affine coordinates for Q) is obtained by replacing x by X/Z^2 and y by Y/Z^3 in eqs.(1.3), (1.4) as well as replacing $Z_2 = 1$. Thus, eqs.(1.17) and (1.18) are obtained for a resulting point $R'(X'_3, Y'_3, Z'_3)$.

$$X'_3 = \left(\frac{Y_2 - (Y_1/Z_1^3)}{X_2 - (X_1/Z_1^2)} \right)^2 - (X_1/Z_1^2) - X_2 \quad (1.17)$$

$$= \left(\frac{Y_2Z_1^3 - Y_1}{(X_2Z_1^2 - X_1)Z_1} \right)^2 - (X_1/Z_1^2) - X_2$$

$$Y'_3 = \left(\frac{Y_2 - (Y_1/Z_1^3)}{X_2 - (X_1/Z_1^2)} \right) ((X_1/Z_1^2) - X'_3) - (Y_1/Z_1^3) \quad (1.18)$$

$$= \left(\frac{Y_2Z_1^3 - Y_1}{(X_2Z_1^2 - X_1)Z_1} \right) ((X_1/Z_1^2) - X'_3) - (Y_1/Z_1^3)$$

To eliminate inversion, the coordinates for the point $R(X_3, Y_3, Z_3)$ are set to $X_3 = X'_3 \cdot Z_3^2$ and $Y_3 = Y'_3 \cdot Z_3^3$ where $Z_3 = (X_2Z_1^2 - X_1)Z_1$. The resulting set of eqs.(1.19),(1.20), and (1.21) define

the ADD operation in mixed Projective Jacobian and Affine coordinate systems.

$$X_3 = (Y_2 Z_1^3 - Y_1)^2 - (X_2 Z_1^3 - X_1)^2 (X_2 Z_1^2 + X_1). \quad (1.19)$$

$$Y_3 = (Y_2 Z_1^3 - Y_1)(X_1(X_2 Z_1^3 - X_1)^2 - X_3) - Y_1(X_2 Z_1^3 - X_1)^3. \quad (1.20)$$

$$Z_3 = (X_2 Z_1^2 - X_1) Z_1. \quad (1.21)$$

By storing some intermediate elements, X_3 , Y_3 and Z_3 can be computed using three field squarings and eight field multiplications as follows:

$$A \leftarrow Z_1^2, B \leftarrow Z_1 \cdot A, F \leftarrow D - Y_1, X_3 \leftarrow F^2 - (H + 2I), C \leftarrow X_2 \cdot A, G \leftarrow E^2, D \leftarrow Y_2 \cdot B, H \leftarrow G \cdot E, E \leftarrow C - X_1, I \leftarrow X_1 \cdot G, Y_3 \leftarrow F \cdot (I - X_3) - Y_1 \cdot H, Z_3 \leftarrow Z_1 \cdot E.$$

1.5.2 Projective Lopez-Dahab Coordinate System

In Projective LD-coordinate system, $P(X, Y, Z)$ with $Z \neq 0$, corresponds to the Affine $P(x, y) = (X/Z, Y/Z^2)$. Thus, the implication that for $Z = 1$, $x = X$ and $y = Y$ also holds for Projective LD-coordinates. Therefore, $P(x, y)$ in Affine coordinates automatically corresponds to $P(X, Y, 1)$ in Projective LD-coordinates, producing a 1-1 correspondence between the two coordinate systems for a fixed Z . The point at infinity in Projective LD-coordinates corresponds to $(1, 0, 0)$, while the negative of (X, Y, Z) is $(X, X + Y, Z)$. Further details on Projective LD-coordinate system can be found in [15]. The DL coordinate DBL operation of a point $P(X_1, Y_1, Z_1)$ results in the point $2P(X_2, Y_2, Z_2)$ obtained through eqs. (1.22), (1.23), and (1.24).

$$X_2 = X_1^4 + b \cdot Z_1^4. \quad (1.22)$$

$$Y_2 = b Z_1^4 \cdot Z_2 + X_2 \cdot (a Z_2 + Y_1^2 + b Z_1^4). \quad (1.23)$$

$$Z_2 = Z_1^2 \cdot X_1^2. \quad (1.24)$$

The LD-coordinate ADD operation of two points $P(X_0, Y_0, Z_0)$ and $Q(X_1, Y_1, Z_1)$, giving the point $R(X_2, Y_2, Z_2)$, is defined as follows:

$$A_0 \leftarrow Y_1 \cdot Z_0^2, A_1 \leftarrow Y_0 \cdot Z_1^2, B_0 \leftarrow X_1 \cdot Z_0, B_1 \leftarrow X_0 \cdot Z_1, C \leftarrow A_0 + A_1, D \leftarrow B_0 + B_1, E \leftarrow Z_0 \cdot Z_1, F \leftarrow D \cdot E, Z_2 \leftarrow F^2, G \leftarrow D^2 \cdot (F + a E^2), H \leftarrow C \cdot F, X_2 \leftarrow C^2 + H + G, I \leftarrow D^2 \cdot B_0 \cdot E + X_2, J \leftarrow D^2 \cdot A_0 + X_2, Y_2 \leftarrow H \cdot I + Z_2 \cdot J.$$

On the other hand, the mixed Affine-LD-coordinates ADD of two points $P(X_0, Y_0, Z_0)$ and $Q(X_1, Y_1, 1)$, giving the point $R(X_2, Y_2, Z_2)$, is defined as follows:

$$A = Y_1 \cdot Z_0^2 + Y_0, B = X_1 \cdot Z_0 + X_0, C = Z_0 \cdot B, D = B^2 \cdot (C + a Z_0^2), Z_2 = C^2, E = A \cdot C, X_2 = A^2 + D + E, F = X_2 + X_1 \cdot Z_2, G = X_2 + Y_1 \cdot Z_2, Y_2 = E \cdot F + Z_2 \cdot G$$

1.5.3 Coordinate Systems Comparison

The cost of the EC operations ADD and DBL in terms of binary field operations change according to the coordinate system used to represent points on the EC. Table 1.2 shows the overall costs of EC DBL and ADD for Affine, Jacobian and LD-coordinate systems in terms of field operations. In EC ADD, both Projective coordinate systems (Jacobian and Lopez-Dahab) avoid the use of field inversion. A field inversion is equivalent to 9-12 field multiplications according to Table 1.1. Therefore, the total cost of EC DBL in Affine is equal to 11-14 field multiplications (depending on the inversion algorithm used). Compared to the costs of 5 and 2 field multiplications for Projective Jacobian and LD-coordinate systems respectively, it is clear that the Affine representation is too costly. The EC ADD in Affine costs from 11 to 15 field multiplications depending on the inversion algorithm used. It is comparable to the costs of EC ADD for both Projective coordinates. However, it is clear that the EC mixed Affine-LD-coordinate system is the

most cost efficient. Overall, LD-Projective coordinates are most cost effective in the case of EC DBL and mixed EC ADD.

Coordinates	EC point add			EC mixed point add			EC point double		
	M	I	T	M	I	T	M	I	T
Affine	2	1	2	-	-	-	2	1	2
Projective Jacobian	14	-	6	10	-	6	5	-	2
Projective Lopez-Dahab	14	-	7	9	-	3	4	-	2

Table 1.2: $GF(2^m)$ operation count and number of temporary variables (finite field elements) for EC point addition and doubling ($M =$ Multiplications, $I =$ Inversions, $T =$ number of Temporary Variables)[8].

The ADD and DBL are the fundamental operations in EC arithmetic upon which the concept of EC PM is defined. EC PM is an essential operation in EC cryptographic protocols. It is performed on a point P belonging to $E(\mathbb{F})$ to obtain a product point kP . With k being the scalar in the field \mathbb{F} , the product is evaluated as follows: $kP = P + P + \dots + P$, k times.

1.6 Elliptic Curve Point Multiplication

In the context of cryptography, EC PM is a computationally demanding operation. It is achieved using EC ADDs and DBLs. The classical method, known as "DBL and ADD," performs point multiplication based on the binary format of the scalar k . This method uses an accumulator to store the sum kP . By parsing k from left to right, it performs ADD and DBL operations based on the digits of k : for every "1" digit, P is added (ADD) followed by a doubling of the content of the accumulator (DBL), and for every "0" digit, only a DBL is performed. However, this method is vulnerable to side-channel attacks. It is also inefficient as the binary representation has a high density of 0.5 (Hamming weight⁴/Total number of bits). This density correlates with the number of ADDs used in the computation of EC PM. Therefore, to reduce the cost of EC PM, various representations of the scalar k have been developed. To reduce the cost of EC PM, various representations of the scalar k have been developed. One commonly used method is the Non-Adjacent Form (NAF), which reduces the representation density to 0.33. Using w -bit windowing methods on NAF, (i.e., w -NAF), aims to make the density even sparser, but they require additional memory for precomputations. The w -NAF method employs a sliding window approach but has right-to-left carry-overs, making it unsuitable for memory-constrained devices. The Radix- 2^w representation, introduced by Homayoon and Gupta in 1990, was applied to EC PM in [4]. This use of the Radix- 2^w representation relies on recoding the scalar k with fewer nonzero digits, achieving a density of 0.19. Therefore, successfully optimizing the cost of EC ADDs while minimizing memory expenses.

1.7 Radix- 2^w Elliptic Curve Multiplication

The Radix- 2^w methods for ECPM and EC DPM rely on the recoding of the scalar k with fewer nonzero digits using a w -bit window. They utilize a set of precomputed points corresponding

⁴Hamming weight of a given string is the number of symbols that are different from the zero-symbol of the alphabet used.

to the possible values of the non-zero digits in order to evaluate the multiplication. Both the precomputed set of points, and the possible values of the non-zero digits depend on the size of the w -bit window used, which is defined based on the bit-length of k . In this section, we will only be introducing the mathematical concepts necessary for our implementation according to the papers [4], and [9]. Further details on the Radix- 2^w method for EC PM, and EC DPM can be found in the same papers.

1.7.1 Radix- 2^w Recoding

The Radix- 2^w EC PM represents the scalar k as the sum shown in eq.(1.25). The number is sliced into $w+1$ sized integer values. Each slice is multiplied by its corresponding weight ($2^{w \times i}$). The slice itself is signed, and the $k_{w \times i + w - 1}$ bit (Most Significant Bit (MSB)) of this slice determines the sign. The value of the bit $k_{w \times i - 1}$ is also added to the slice. This bit is shared with a different slice. This overlap allows for the overall value of the scalar to be evaluated through these slices without the use of carries. This property is what enables the EC PM to be performed from either right-to-left or left-to-right. The scalar k can, therefore, be represented in terms of slices Q_i as shown in eq.(1.26). The slices Q_i are, therefore, computed as shown in eq.(1.27).

$$k = \sum_{i=0}^{\lceil (l+1)/w \rceil - 1} \left(-2^{w-1} k_{w \times i + w - 1} + 2^{w-2} k_{w \times i + w - 2} + \dots + 2^2 k_{w \times i + 2} + 2^1 k_{w \times i + 1} + 2^0 k_{w \times i} + k_{w \times i - 1} \right) \times 2^{w \times i} \quad (1.25)$$

$$k = \sum_{i=0}^{\lceil (l+1)/w \rceil - 1} Q_i \times 2^{w \times i} \quad (1.26)$$

$$Q_i = -2^{w-1} k_{w \times i + w - 1} + 2^{w-2} k_{w \times i + w - 2} + \dots + 2^2 k_{w \times i + 2} + 2^1 k_{w \times i + 1} + 2^0 k_{w \times i} + k_{w \times i - 1} \quad (1.27)$$

These slices Q_i should be of equal bit-length. To ensure that, a number of zero bits are concatenated with the scalar k of length l . Specifically, an additional zero to the right of the scalar k , and a number Nz of zeros to its left are added. Nz is determined by eq.(1.28).

$$Nz = w - (l \bmod w) \quad (1.28)$$

Each slice Q_i is, therefore, of fixed bit-length $w+1$. They are, as previously mentioned, signed. They also overlap at the extremities, sharing the $k_{w \times i + w - 1}$, and $k_{w \times i - 1}$ bits, with the neighboring slices. This implies that the set of values a slice Q_i can take is finite and dependent on the value of w . This set of values is represented by the Digit Set $Ds(2^w)$ and is shown in eq.(1.29). The unsigned value of a slice Q_i , like any number, can be represented as a product of an odd integer and a power of two. Using the MSB of the slice to represent the sign, a formula $Q_i = (-1)^{k_{w \times i + w - 1}} \times m_i \times 2^{n_i}$ is found, where m_i is the odd valued integer, 2^{n_i} is the even part of the slice, and $k_{w \times i + w - 1}$ is the MSB of this slice.

$$Q_i \in Ds(2^w) = \left\{ -2^{w-1}, -2^{w-1} + 1, \dots, -1, 0, 1, \dots, 2^{w-1} - 1, 2^{w-1} \right\} \quad (1.29)$$

$$k = \sum_{i=0}^{\lceil (l+1)/w \rceil - 1} (-1)^{k_{w \times i + w - 1}} \times m_i \times 2^{w \times i + n_i} \quad (1.30)$$

This makes the representation shown in eq.(1.30) possible, where n_i is merged with the weight of the slice ($2^{w \times i}$). $m_i \in Os(2^w) \cup \{0, 1\}$, where the Odd set $Os(2^w)$ is as shown in eq.(1.31).

$$Os(2^w) = \left\{ 3, 5, 7, \dots, 2^{w-1} - 1 \right\} \quad (1.31)$$

Finally, to use this in EC PM, all translation information from binary to Radix- 2^w format is grouped in the recoding set $Rs(2^w)$ shown in (1.32).

$$Rs(2^w) = \left\{ (k_l, m_{\lceil(l+1)/w\rceil-1}, n_{\lceil(l+1)/w\rceil-1}), (k_{l-w}, m_{\lceil(l+1)/w\rceil-2}, n_{\lceil(l+1)/w\rceil-2}), \right. \\ \left. \dots, (k_{2 \times w-1}, m_1, n_1), (k_{w-1}, m_0, n_0) \right\} \quad (1.32)$$

The values of m_i and n_i can be invoked either by a call to Algorithm 1.4, or by making an appropriate look-up-table.

Algorithm 1.4 Computation of m_i and n_i

Input: Q_i

Output: m_i and n_i

- 1: Compute Q_i according to eq.(1.25)
 - 2: $n_i = 0$
 - 3: **while** Q_i is even **do**
 - 4: $Q_i = Q_i/2$
 - 5: $n_i = n_i + 1$
 - 6: $m_i = Q_i$
 - 7: **Return** m_i, n_i
-

The maximum number of ADDs for the whole EC PM $k \times P$ is given by (1.33).

$$\lceil(l+1)/w\rceil + 2^{w-2} - 1 \quad (1.33)$$

Through this formula, the value w resulting in the minimum number of ADDs is obtained. The derivative of eq.(1.33) with respect to w , is equated to 0 giving eq.(1.34); with W being the Lambert function. It is defined as ($y = x \cdot e \iff x = W(y)$). For our purposes, an integer value of w is required. Since the resulting w has a real value, both its ceiling and floor functions are computed to find its integer value. Their maximum number of ADDs is compared through eq.(1.33). Then, the value of w resulting in the lesser amount of maximum ADDs is used.

$$w = \frac{2 \times W((l+1) \times \log(2))}{\log(2)} \quad (1.34)$$

To better understand how the Radix- 2^w recoding process works, let's consider an example. Let us take $k = (5892973)_{10}$ having the binary representation: $(10110011110101101101101)_2$, where $l = 23$. Using eq.(1.34), w is found:

$$w = \frac{2 \times W((23+1) \times \log(2))}{\log(2)} = 3.4995.$$

According to eq.(1.33), a maximum of 9 ADDs are required for both $w = 3$ and $w = 4$. It is preferable to use a smaller size for w in order to reduce the size of the precomputation set. However, we will fix w to 4 in order to show a larger precomputation set for the sake of this example. Having set $w = 4$, the scalar k is split into 6 slices Q_i , considering that: $\lceil(l+1)/w\rceil = \lceil(23+1)/4\rceil = 6$. Algorithm 1.4 is used to determine the $Rs(2^4)$ set corresponding to k , which gives:

$$Rs(2^4) = \left\{ (0, 3, 1), (1, 3, 1), (1, 1, 0), (1, 5, 0), (0, 7, 0), (1, 3, 0) \right\}$$

As for finding the number of zeros to be concatenated, besides the zero added to the right of the scalar k , we use eq.(1.28) to determine that $Nz = 1$. These Q_i slices are depicted in Figure 1.5

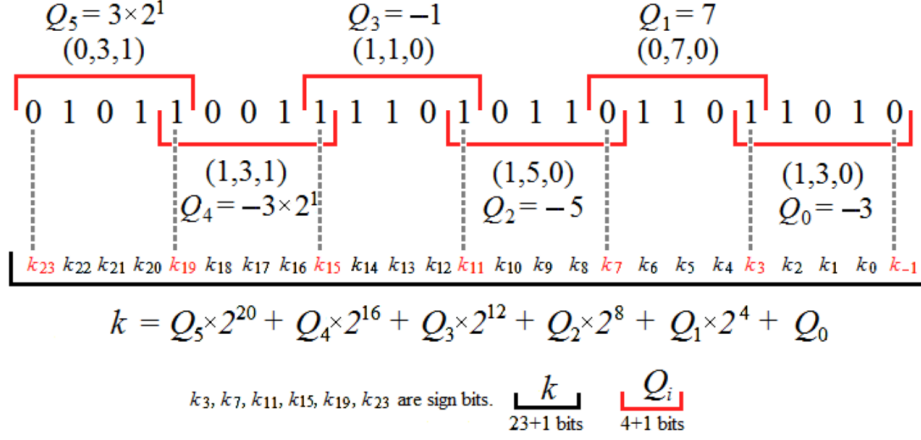


Figure 1.5: Decomposition of $k = (5892973)_{10}$ in Radix $2^4[4]$.

Using eq.(1.30), k is translated in Radix 2^4 as:

$$k = 3 \times 2^{21} - 3 \times 2^{17} - 1 \times 2^{12} - 5 \times 2^8 + 7 \times 2^4 - 3 \times 2^0 = (03000\bar{3}0000\bar{1}000\bar{5}0007000\bar{3})_{24}$$

Where \bar{X} represents a negative digit. The concepts dealing with the Radix- 2^w apply to both EC PM and DPM methods.

1.7.2 Radix- 2^w Elliptic Curve Point Multiplication Method

Given an elliptic curve E defined over either a binary or prime field F , and a point P defined on E . The EC PM of the scalar k , as represented in eq.(1.30), and the point P , is shown in eq.(1.35).

$$kP = \sum_{i=0}^{\lceil (l+1)/w \rceil - 1} (-1)^{w \times i + w - 1} \times (m_i \times P) \times 2^{w \times i + n_i} \quad (1.35)$$

Since multiplication by 2 is a point doubling operation DBL, it is necessary that the terms $(-1)^{w \times i + w - 1} \times m_i \times P$ be added to the sum for this EC PM evaluation. From the properties of elliptic curves, we know $m_i \times P$ is a point that belongs to E . We also know that the inverse of the point also belongs to the same EC. Therefore, $(-1)^{w \times i + w - 1} \times m_i \times P$ is a point defined on E , and can be added to the sum by an ADD operation. The set of points $m_i \times P$ is precomputed for the EC PM. This gives the precomputation set $Ps(2^w)$ given in eq.(1.36).

$$Ps(2^w) = \left\{ 3 \times P, 5 \times P, 7 \times P, \dots, (2^{w-1} - 1) \times P \right\} \quad (1.36)$$

This set is obtained recursively. Starting off with the points P and $2 \times P$, and using eq.(1.37) with the counter j from 0 to $j = |Os(2^w)| - 1$, the $Ps(2^w)$ is found.

$$[2 \times (j + 1) + 1] \times P = [2 \times j + 1] \times P + 2 \times P \quad (1.37)$$

Finally, the computation for the EC PM method is shown in Algorithm 1.5. It starts by finding the different parameters and fixed values required for the multiplication: the value of w and therefore the values of m_i , n_i , Nz , and the precomputation set. Next, the resulting point R

Algorithm 1.5 Radix- 2^w Point Multiplication [4].

Input: $P \in E(\mathbb{F})$ and scalar k
Output: $R = k.P \in E(\mathbb{F})$

- 1: Compute w_{temp} according to eq.(1.34)
- 2: Determine w among ceiling and floor functions of w_{temp} according to (1.33)
- 3: Compute and store $Ps(2^w)$ set
- 4: Concatenate a zero to k_o according to eq.(1.25)
- 5: Concatenate Nz zeros to k_{l-1} according to eq.(1.28)
- 6: $R \leftarrow P_\infty$ $\triangleright P_\infty$ is the point at infinity
- 7: **for** $i = (1+Nz)/w - 1$ **downto** 0 **do**
- 8: $Q_i[1] = \overline{Q_i[0]}$
- 9: $Q_i[0] = Q_i[k_{w \times i + w - 1}]$
- 10: Load (m_i, n_i) according to $Q_i[0]$ from the indexation table
- 11: **for** $j = w-1$ **downto** 0 **do**
- 12: $R = R + R$ \triangleright Point Doubling (DBL)
- 13: **if** $m_i \neq 0$ **then**
- 14: **if** $j = n_i$ **then**
- 15: $R = R + (-1)^{k_{w \times i + w - 1}} \times (m_i \times P)$
- 16: **Return** R

is initialized to the point at infinity. From here, two loops perform the summation according to the values of the slices Q_i : an outer loop serves to double R as required, and an inner loop adding $(-1)^{w \times i + w - 1} \times m_i \times P$ at the appropriate locations. By the end of the iterations, R would be equal to $k \times P$. It is, therefore, the returned value.

1.7.3 Radix- 2^w Elliptic Curve Double Point Multiplication Method

For the EC DPM method, we assume two scalars u and v , along with two points on the elliptic curve P and Q . The scalars u and v are represented in the same manner as the scalar k . The slices in EC DPM are denoted du and dv for u and v respectively. The sum of the multiplications $u \cdot P$ and $v \cdot Q$ is therefore as shown in eq.(1.38).

$$u \cdot P + v \cdot Q = \sum_{i=\lceil(l+1)/w\rceil-1}^0 (du_i \cdot P)2^{w \times i} + \sum_{i=\lceil(l+1)/w\rceil-1}^0 (dv_i \cdot Q)2^{w \times i} \quad (1.38)$$

The previous definition for odd values m is also applied here for slices du and dv producing mu and mv as shown in eq.(1.39). Therefore, $mu, mv \in Os(2^w) \cup 0, 1$ and $du, dv \in Ds(2^w)$.

$$\begin{aligned} u \cdot P + v \cdot Q = & \sum_{i=\lceil(l+1)/w\rceil-1}^0 [(-1)^{w \times i + w - 1} \times (mu_i \times P) \times 2^{w \times i + n_i}] \\ & + \sum_{i=\lceil(l+1)/w\rceil-1}^0 [(-1)^{w \times i + w - 1} \times (mv_i \times Q) \times 2^{w \times i + n_i}] \end{aligned} \quad (1.39)$$

For the simultaneous EC DPM method, a window size of 2 is fixed. This implies that the pre-

computation set $Ps(2^w)$ is limited to the set shown in eq.(1.40).

$$Ps(2^w) = P_\infty, \pm Q, \pm 2Q, P, P + Q, P - Q, P + 2Q, P - 2Q, 2P, 2P + Q, 2P - Q, 2(P + Q), 2(P - Q) \quad (1.40)$$

The EC DPM is therefore performed according to Algorithm 1.6.

Algorithm 1.6 Simultaneous Radix-2² Double Point Multiplication[9].

Input: : P and $Q \in E(\mathbb{F})$ and scalars u and v of bit-length l .
Output: $QR = u.P + v.Q \in E(\mathbb{F})$.
// Points of line 1 must be computed in the indicated order
1: $P + Q; P + 2Q; 2P + Q; P - Q; P - Q; 2P - Q$.
2: Concatenate a zero to the right of u_0 and v_0 .
3: Concatenate Nz zeros to u_{l-1} and v_{l-1} according to eq.(1.28).
4: $R = P_\infty$ // P_∞ is the point at infinity
5: **for** $i = (l + Nz) - 2 - 1$ **downto** 0 **do**
6: **Case** ($|du_i|, |dv_i|$)
7: (0,0) : $M = P_\infty$
8: (1,0) (2,0) : $M = P$
9: (0,1) (0,2) : $M = Q$
10: (1,1) (2,2) : $M = P + (-1)^{v_{2 \times i+1} - u_{2 \times i+1}} . Q$
11: (1,2) : $M = P + (-1)^{v_{2 \times i+1} - u_{2 \times i+1}} . 2.Q$
12: (2,1) : $M = 2.P + (-1)^{v_{2 \times i+1} - u_{2 \times i+1}} . Q$
13: $R = R + R$ // DBL
14: **Case** ($|du_i|, |dv_i|$)
15: (0,0) (0,1) : $R = R + R$; $R = R + (-1)^{v_{2 \times i+1}} . M$
16: (0,2) : $R = R + (-1)^{v_{2 \times i+1}} . M$; $R = R + R$
17: (1,0) (1,1) (1,2) (2,1) : $R = R + R$; $R = R + (-1)^{u_{2 \times i+1}} . M$
18: (2,0) (2,2) : $R = R + (-1)^{u_{2 \times i+1}} . M$; $R = R + R$
19: **Return** R

1.8 Conclusion

In this chapter, we have presented an introduction to ECs and the associated Finite Fields in which they are defined. We have delved into the binary field operations necessary to the construction of EC operations. Following that, we have introduced the fundamental EC operations. We also have discussed the the use of EC projective coordinate systems and their subsequent effect on the cost of EC ADD and DBL. Then, we have considered some methods of constructing EC PM and their drawbacks. Finally, we introduced the Radix-2^w and its method for optimizing the evaluation of EC PM as it is the focus of this project.

To conclude this chapter, we highlight the relationship between the operations discussed in it through Figure 1.2 which shows how EC cryptography relies on EC PM, which builds upon EC ADD and DBL operations. These operations, in turn, depend on the field operations of the field the elliptic curve is defined over.

Chapter 2

Elliptic Curve Cryptography

2.1 Introduction

Cryptography refers to the design of mechanisms based on mathematical algorithms that enable secure communications in the presence of malicious adversaries. The primary objective of using cryptography is to provide the following four fundamental security services:

1. **Confidentiality:** Keeping data secret from all but those authorised to access it. It is the fundamental security service provided by crypto-systems.
2. **Data Integrity:** Detecting when data is manipulated in unauthorized manners.
3. **Authentication:** Verifying a communicating entity's identity, and ensuring they are indeed who they claim to be.
4. **Non-repudiation:** Ensuring that the creation and transmission of data by a sender, to a recipient, can be proven by a third party. It is a property that is most desirable in situations where there are chances of a dispute over the exchange of data.

In this chapter, we will explain how communication happens in a cryptosystem and how the aforementioned security services can be achieved. First, we will introduce the concept of keys in cryptography and their crucial role in cryptographic protocols. Then, we will discuss the elliptic curve digital signature cryptographic system and its dependency on Public-key cryptography. Lastly, we will introduce the encryption/ decryption system, how it can be adapted to ECs and joint to the ECDSA.

2.2 Cryptographic Model

The basic cryptographic model is a fundamental concept in cryptography that describes the communication process between two (or more) parties. At its core, this model is a cryptographic system consisting of said two parties, typically referred to as the sender and the receiver, whom want to interact with one another securely. This model is applicable to different protocols, prioritizing different aspects of security. Figure 2.1 focuses on confidentiality. It consists of the following components:

- **Plaintext:** It is the data to be protected during transmission.

- **Ciphertext:** It is the cipher version of the plaintext produced by the encryption algorithm. A format that is unintelligible to an interceptor.
- **Encryption Key:** It is a value that the sender inputs into the encryption algorithm, along with the plaintext, in order to compute the ciphertext.
- **Decryption Key:** It is a value that the receiver inputs into the decryption algorithm, along with the ciphertext, in order to recompute the plaintext. The decryption key is related to the encryption key, but is not always identical to it.
- **Encryption Algorithm:** It is a cryptographic algorithm that takes plaintext and an encryption key as input to produce a ciphertext.
- **Decryption Algorithm:** It is a cryptographic algorithm that takes a ciphertext and a decryption key as input, and outputs a plaintext. Essentially the inverse operation of the encryption algorithm.

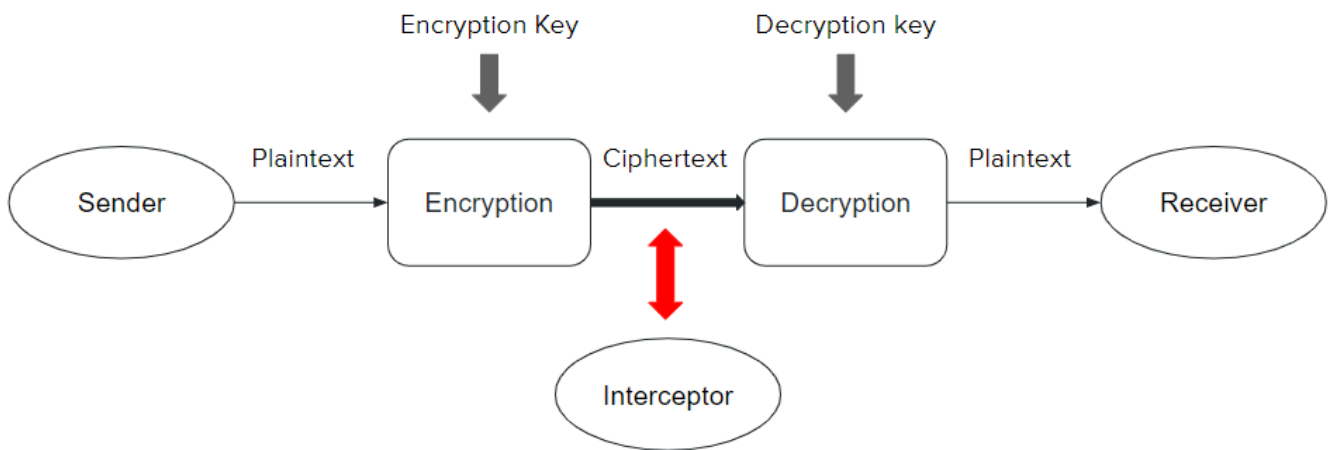


Figure 2.1: *Cryptographic Model.*

In order to achieve secure communication, the sender, in Figure 2.1, converts their message into ciphertext using the encryption algorithm. The ciphertext is then forwarded to the receiver through a publicly accessible channel. The receiver would then input it into the decryption algorithm to recover the plaintext.

Fundamentally, cryptosystems can be classified into two types: Symmetric and Asymmetric-Key Encryption cryptosystems. This classification is made based on the symmetry, or lack thereof, pertaining to the keys used in the encryption and decryption processes.

2.2.1 Symmetric-key Cryptography

Symmetric Key Cryptography is a cryptographic system where a common secret key is leveraged for both the encryption and decryption processes. This makes it so that when data is converted to ciphertext, it cannot be read or inspected by anyone who does not have the secret key that was used to encrypt it.

For example, take a secret key k shared by sender A and receiver B . A would encrypt a plaintext message m using k . They would then transmit the resulting ciphertext c to B . Upon receiving this, B would use the same key k to recover m . Some common symmetric encryption

algorithms to achieve this confidentiality include the Advanced Encryption Standard and the Data Encryption Standard. As for data integrity and origin authentication, then A would compute the authentication tag t of m . This is done using a message authentication code (MAC) and k . A would then send m and t to B . Upon receiving this, B would use the MAC algorithm and the same k to recompute the tag t' of m . The message is accepted as having originated from A if $t' = t$. A common MAC algorithm used is the Hash Message Authentication Code (HMAC).

This form of cryptography is fast and of high efficiency, however it requires both the sender and recipient of the data to have access to the secret key. This requirement is one of the main drawbacks of symmetric key encryption. It is referred to as the key distribution problem, where there would be a need for a secret and authenticated channel in order to distribute keying material. Another drawback is the key management problem. Where in a network of N parties, each party may have to maintain different keying material with each of the other $N-1$ parties. In addition, since keying material is shared between two (or more) parties, it is impossible to distinguish between the actions taken by the different holders of a secret key. Therefore, symmetric-key techniques cannot be used to devise elegant digital signature schemes that provide non-repudiation services.

2.2.2 Public-key Cryptography

Public-key cryptography, also known as Asymmetric-key cryptography, was introduced to overcome the shortcomings of symmetric-key cryptography. The difference between these two types of cryptography is that the former requires for the keying material exchanged to be authentic but not secret. Each entity selects a single key pair consisting of a public key and a corresponding private key that the entity keeps secret. The key pair is selected such that the problem of deriving the private key, solely from knowledge of its corresponding public key, is equivalent to solving a computational problem that is believed to be intractable. Some number-theoretic problems used in popular public-key schemes are:

- Integer Factorization Problem (IFP).
- Discrete Logarithm Problem (DLP).
- Elliptic Curve DLP.

Since public-key encryption uses a key pair, it takes a different approach to achieve the fundamental security services. To explain the scheme, take as example the entity A . To establish confidentiality, A obtains an authentic copy of B 's public key Q_B . This key, along with A 's private key d_A , is used to encrypt a message m and transmit the resulting cyphertext c to B . Upon receiving c , B would then use it along with their private key d_B to recover m .

As for data integrity and origin authentication, A would use d_A to compute the signature s of m by way of a signature generation algorithm. A would then forward s and m to B . Since d_A is known only to A , it is assured that m indeed originates from A . Then B uses an authentic copy of A 's public key Q_A , along with m , to confirm that s was indeed generated by A . This is done using a signature verification algorithm.

As discernible, this verification requires only the non-secret quantities m and Q_A . This means that the signature for a message m can also be verified by a third party in case A denies having signed m . Therefore ensuring non-repudiation. Additionally, given that s changes entirely depending on the message m being signed, a third party can not simply append s to a different message and claim that A signed it.

Building upon these concepts, various public-key cryptography based systems have emerged, each with its own strengths and weaknesses. One widely known cryptosystem is RSA, which is named after its creators Rivest, Shamir, and Adleman. It is based on the Integer factorization problem. RSA is known for its versatility and is widely used for secure data transmission, digital signatures, and key exchange protocols. However, RSA key generation and encryption/decryption operations can be computationally expensive, particularly for large key sizes. DSA, which stands for Digital Signature Algorithm, is another popular public key cryptosystem that is specifically designed, as the name suggests, for digital signatures. It is based on the mathematical properties of the DLP in a finite field. It provides a mechanism to verify the authenticity and integrity of messages. Compared to RSA, DSA generally offers faster signature generation and verification. However, DSA does not provide encryption capabilities and is mainly focused on signature-related operations. ECC is an asymmetric cryptosystem that has gained significant popularity due to its strong security and computational efficiency. It relies on the EC DLP to provide robust encryption and digital signature capabilities. Compared to RSA, ECC offers equivalent security with shorter key lengths, resulting in faster computation and reduced memory requirements. This makes ECC particularly suitable for resource-constrained environments such as embedded systems. Additionally, ECC signatures, such as ECDSA, are generally shorter than DSA signatures for the same level of security.

2.3 Elliptic Curve Digital Signature Algorithm

ECDSA, developed for use with elliptic curves, is an evolution of the briefly aforementioned DSA. The latter is based on modular exponentiation and the discrete logarithm problem in finite fields. On the other hand, ECDSA utilizes elliptic curves to provide the same functionality as DSA but with more efficient computations and shorter signature lengths. ECDSA operates on elliptic curves over finite fields, using the EC DLP. The latter involves finding the exponent, or private key, in the equation $P = kG$. P here is a point on an elliptic curve, G is the generator point of the curve, and k is the private key. The difficulty lies in determining the corresponding private key k , given P and G . The security of ECDSA relies on the assumption that no algorithm exists to efficiently solve this computational problem for sufficiently large keys. Examples of the best-known algorithms capable of solving the EC DLP are Pollard's rho algorithm or the index calculus method. Then again, these algorithms have exponential time complexities, making them impractical for said large keys. Therefore, the choice of the elliptic curve parameters used in the ECDSA, including the curve coefficients, the field size, and the generator point G is necessary to ensure the algorithm's security against potential attacks. ECDSA consists of a multi-step process involving signature generation and signature verification which will be addressed in the next subsections.

2.3.1 Signature Generation

To create a digital signature for a message m , an asymmetric key pair is to first be generated. An asymmetric key pair, (Q, d) , consists of the private key d and its corresponding public key Q . The private key is a randomly generated integer within a specific range. The range is determined according to the elliptic curve selected and its parameters. The public key is a point belonging to the elliptic curve obtained by performing an EC PM operation. This operation consists of multiplying d by the generator point G to obtain the corresponding public key. This process is assimilated in Algorithm 2.1, where it takes in as input a set D of domain parameters and generates key pairs (Q, d) . The set D consists of (q, E, G, n) , where q is the field size; E is the

elliptic curve used; G is the generator point, and n is the order of G .

Algorithm 2.1 Elliptic Curve Key Pair Generation

Input: Elliptic curve domain parameters (q, E, G, n)

Output: Public key Q and private key d

- 1: Generate $d \in_R [1, n-1]$
 - 2: Compute $Q = d \cdot G$
 - 3: **Return** (Q, d)
-

Before proceeding to signing the message, ECDSA typically applies a cryptographic hash function, that takes the entire message as input, and produces a message digest. It is also known as a hash value or hash digest. This serves as a condensed representation of the original message and is used during both signature generation and verification. The hash digest produced possesses the following properties:

- **Deterministic:** The same input message will always produce the same hash digest.
- **Fixed Length:** Regardless of the size of the input message, the hash function produces a hash digest of a fixed length.
- **One-Way:** It is computationally infeasible to derive the original message from its hash digest.
- **Collision-Resistance:** It is highly unlikely that two different messages will produce the same hash digest.

This offers multiple benefits, including efficiency, standardized representation, and resistance to potential attacks.

After obtaining the hash digest, a random number known as a nonce (number used once) is generated. A new nonce is chosen for each signing operation, meaning it is independent of the private key and the message being signed. This ensures that each signature produced by the same private key for different messages will be unique. As depicted in Figure 2.2, these parameters calculated (the private key d , the hash digest e , and the nonce k) are then used to compute two components: the r and the s values, which form the digital signature. These values are calculated as follows:

- Calculating the r value: k is multiplied by the generator point G and the x -coordinate of the resulting point is taken as the r value.
- Calculating the s value: k , d , e , and the r value are used in modular arithmetic operations to compute s .

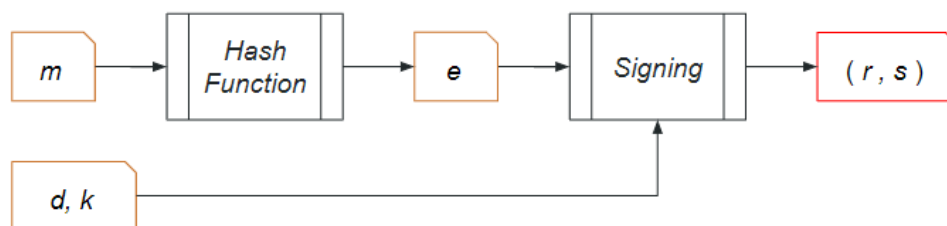


Figure 2.2: Signature generation process ($m =$ message, $e =$ hash digest, $d =$ private key, $k =$ nonce, $(r, s) =$ signature components).

This process is assimilated in Algorithm 2.2, where it takes as input the private key d , a message m and the set of domain parameters D to produce the signature. Domain parameters for ECDSA are of the form (q, FR, a, b, G, n) , where q , again, is the field size; FR is an indication of the basis used; a and b are two field elements that define the equation of the curve; G is the generator point, and n is the order of G . The condition in steps 4 and 8 is a precautionary measure to ensure that the signature generation process follows the required guidelines and produces valid signatures. The r and s values forming the digital signature are later used, along with the public key Q and the original message m , for signature verification.

Algorithm 2.2 ECDSA signature generation

Input: Domain parameters $D = (q, FR, a, b, G, n)$, private key d , message m

Output: Signature (r, s)

- 1: Select $k \in_R [1, n-1]$
 - 2: Compute $kG = (x_1, y_1)$ and convert x_1 to an integer \bar{x}_1
 - 3: Compute $r = \bar{x}_1 \bmod n$
 - 4: **if** $r = 0$ **then**
 - 5: go to step 1
 - 6: Compute $e = H(m)$
 - 7: Compute $s = k^{-1}(e + dr) \bmod n$
 - 8: **if** $s = 0$ **then**
 - 9: go to step 1
 - 10: **Return** (r, s)
-

2.3.2 Signature Verification

To verify a signature, the verifier obtains the public key Q of the signer, which is typically available through a trusted source. Additionally, the verifier receives the message and its associated signature consisting of the r and s values. Similar to the signature generation process, the same cryptographic hash function is applied to the message, to obtain a fixed-length hash digest.

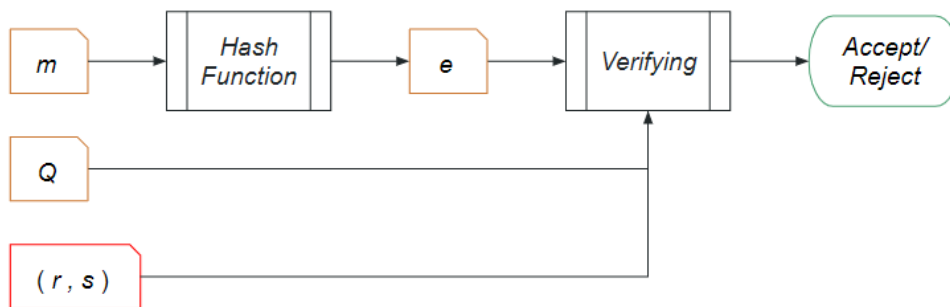


Figure 2.3: Signature verification process ($m =$ message, $e =$ hash digest, $Q =$ public key, $(r, s) =$ signature components).

This ensures the standardized representation of the message. By applying a series of mathematical calculations, involving modular arithmetic and elliptic curve point operations on the parameters received (the public key Q , hash digest e , and the received signature), the ECDSA confirms the integrity and authenticity of the signature. This process is depicted in Figure 2.3, the signature is authenticated by checking if the r and s values fall within certain ranges specified by the elliptic

curve parameters. If either value is out of range, the signature is rejected. The inverse of the s value modulo the curve's order is then computed. This is used to derive two elliptic curve points:

- One point is obtained by multiplying the inverse of s by the hash digest e .
- The other point is obtained by multiplying the r value by the inverse of s .

These points are added together using EC ADD. The resulting point's x-coordinate is then compared to the original r value's x-coordinate. If the values match, the signature is accepted.

This process is assimilated in Algorithm 2.3, where it takes as input the domain parameters D , the public key Q , the message m , and the purported signature. It then accepts or rejects said signature. Step 8 of the algorithm serves as an immediate rejection of the signature if the computed point X is the point at infinity. This is on account that the point at infinity is a special point on the elliptic curve. It represents the result of adding a point to its inverse. Therefore, if X is ∞ , it means the computation in step 7 produced an invalid result. It ensures that any subsequent steps or calculations based on an invalid point are skipped, saving computational resources.

Algorithm 2.3 ECDSA signature verification

Input: Domain parameters $D = (q, FR, a, b, G, n)$, public key Q , message m , signature (r, s)
Output: Acceptance or rejection of the signature

- 1: Verify that r and s are integers in the interval $[1, n-1]$
 - 2: **if** any verification fails **then**
 - 3: return("Reject the signature")
 - 4: Compute $e = H(m)$.
 - 5: Compute $w = s^{-1} \pmod n$.
 - 6: Compute $u_1 = ew \pmod n$ and $u_2 = rw \pmod n$
 - 7: Compute $X = u_1G + u_2Q$
 - 8: **if** $X = \infty$ **then**
 - 9: return("Reject the signature")
 - 10: Convert the x -coordinate x_1 of X to an integer x_1 ;
 - 11: Compute $v = x_1 \pmod n$.
 - 12: **if** $v = r$ **then**
 - 13: return("Accept the signature")
 - 14: **else**
 - 15: return("Reject the signature")
-

While ECDSA is a widely used cryptographic algorithm for generating and verifying digital signatures, it is not concerned with the functionality of encrypting data. Its main purpose is to ensure the authenticity, integrity, and non-repudiation of messages through signature-related operations. It does not inherently provide confidentiality of messages. To achieve this confidentiality, in conjunction with ECDSA, additional cryptographic mechanisms such as symmetric, asymmetric, or hybrid-key encryption schemes are employed.

2.4 Encryption Schemes

Encryption schemes are cryptographic methods used to transform plaintext data into ciphertext, ensuring confidentiality. These encryption schemes are used to encrypt the message itself

before applying the ECDSA signature, providing a comprehensive security solution.

2.4.1 Joint Signature and Encryption System

Figure 2.4 presents an overview of the high-level blocks forming the elliptic curve digital signature and encryption scheme system. This serves as a guide to understanding the connections between said blocks, where the message to be sent is used for signature generation and verification. Before sending it to the recipient, the message is signed and encrypted. The result of the encryption process, m_enc , is then forwarded along with the signature. Then the message passes through the decryption process and its signature is verified.

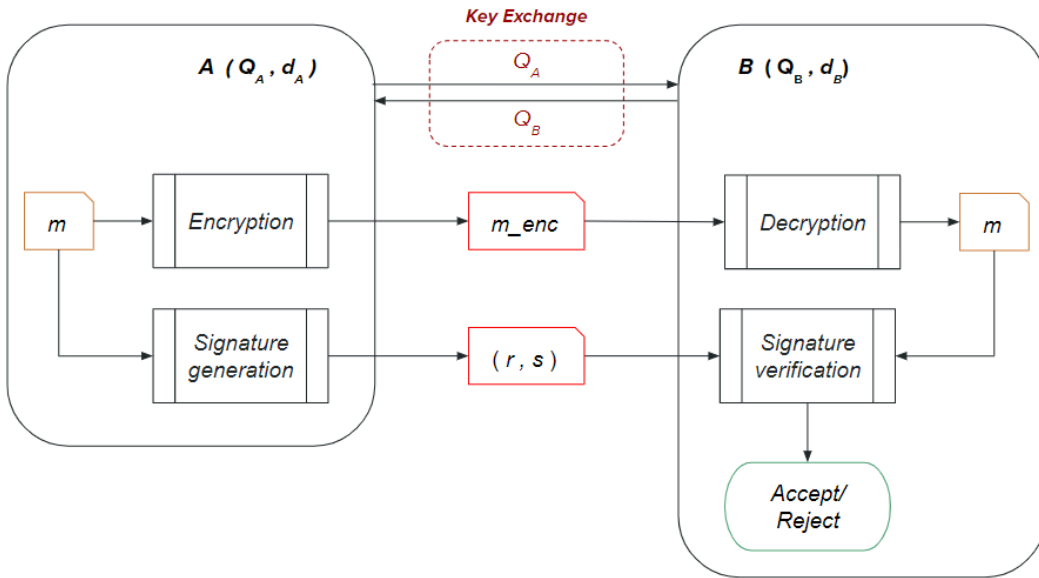


Figure 2.4: Joint ECDSA / Encryption Protocol workflow.

Various encryption schemes exist, each employing different algorithms and techniques. One popular public-key encryption protocol is ElGamal's encryption protocol. It is a public key encryption protocol proposed by Taher ElGamal in 1985. It is based on the DLP. There exist variations and enhancements to the ElGamal encryption protocol. A notable variant is the EC-ElGamal encryption protocol. This variation utilizes ECC instead of the DLP to achieve the same security level with smaller key sizes.

2.4.2 EC-ElGamal Encryption Protocol

As operations in ECC are performed on points on the elliptic curve, EC encryption requires some adaptations. Typically, encryption involves manipulating the characters composing the message via mathematical means to obtain the coded text. As EC operations do not operate directly on characters, a process called mapping is used to convert each character into a distinct point on the elliptic curve. In EC-ElGamal encryption, each character m_i in a message m is individually encrypted through manipulations performed on its mapped point P_m . P_m is generally chosen as a multiple of the EC generator point G . The coordinates of the resulting point of the encryption P_c represent the encrypted character. This process is shown in Figure 2.5. Thus, a sequence of these coordinates would represent the encrypted message. The decryption is later done on the other end by a reverse process. This means that the result of the decryption is a set

of points P_m , where each point would then need to be mapped back into the original character m_i . Therefore, it is necessary that all parties have access to the same mapping.

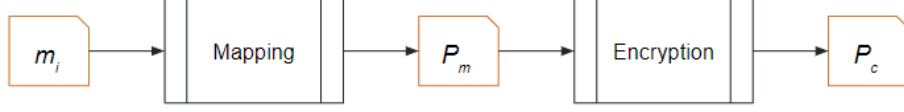


Figure 2.5: Mapping and encryption process for a single character.

To achieve this encryption scheme, the communicating parties need to exchange their public keys which are obtained as previously described in Algorithm 2.1. These exchanged keys are then used both to obtain P_c (Message Encryption) and retrieve P_m (Message Decryption).

2.4.2.1 Message Encryption

Let two parties A and B , having keys (Q_A, d_A) and (Q_B, d_B) respectively, be the communicating parties. The encryption process is done according to Algorithm 2.4; where, for A to encrypt the message m , each character m_i of m must be encrypted. This is achieved by first retrieving the point P_m corresponding to m_i from a lookup table containing all the character mappings. Next, A 's private key d_A is multiplied, using EC PM, by B 's public key Q_B . Finally, P_m , d_A and Q_B are used to obtain P_c through the following operation: $P_c = P_m + d_A \times Q_B$. P_c 's coordinates x_c and y_c are then stored in m_enc . This process is repeated for every character in m .

Algorithm 2.4 Encryption of single character

Input: m, Q_B, d_A

Output: P_c

- 1: **if** $m \in C_s$ **then** $\triangleright C_s$ is character set of m
 - 2: Retrieve P_m from mapping table
 - 3: Compute $d_A \times Q_B$
 - 4: $P_c \leftarrow P_m + d_A \times Q_B$
 - 5: **Return** P_c
-

2.4.2.2 Message Decryption

This process is done according to Algorithm 2.5; where, for B to decrypt the message m_enc received from A , P_c is to be retrieved from m_enc for every character. This is achieved by first multiplying Q_A , using EC PM, by d_B . Then, P_c , d_B and Q_A are used to obtain P_m through the following operation: $P_m = P_c - d_B \times Q_A$. Finally, m_i is retrieved from the lookup table according to the resulting P_m . This process is repeated for every point P_c corresponding to a character in m_enc .

Algorithm 2.5 Decryption of single character

Input: P_c, Q_A, d_B

Output: m_i

- 1: Compute $d_B \times Q_A$
 - 2: $P_m \leftarrow P_c - d_B \times Q_A$
 - 3: Retrieve m_i from mapping table
 - 4: **Return** m_i
-

2.5 Conclusion

In this chapter, we have introduced the ECC concepts most relevant to our project. We started by introducing Public-key cryptography as it is foundational to both digital signature algorithms and cryptographic encryption schemes we aim to implement. Then, building on the mathematical notions given in chapter 1, both ECDSA and ElGamal encryption algorithms were explained and the utility of their joining highlighted. The combination of ECDSA's signature generation and verification with ElGamal's encryption scheme presents a comprehensive security solution for text message communication. The next chapter will delve into our complete design of this joint cryptography system based on the Radix- 2^w method for EC multiplications.

Chapter 3

ECDSA System Design using Radix- 2^w EC multiplication methods

3.1 Introduction

In this chapter, we outline the methods and materials used in our design of the cryptographic system composed of ECDSA and EC-ElGamal encryption algorithm. We will provide a description of the cryptographic system design, and the hardware it was devised for. Given the protocols described in chapter 2 and their mathematical dependencies discussed in chapter 1, we will define the software blocks necessary at every level building up.

First, we will establish the design of the Radix- 2^w EC multiplications. The underlying EC operations, and therefore the finite field operations will be addressed in the context of the EC multiplications. We will also propose the binary EC PM method for the purpose of: cost comparison with the Radix- 2^w EC multiplications, and verification of functionality for EC ADD and DBL. Finally, we will delve into the details of building the cryptographic protocols (Public-key, ECDSA and EC-ElGamal encryption) around our design of the EC PM and EC DPM.

3.2 Radix- 2^w Elliptic Curve Point Multiplication Design

The design of the EC multiplications using the Radix- 2^w method for recoding is composed of three layers. Figure 3.1 shows a comprehensive diagram encompassing the blocks needed on each of the layers. Both EC PM and DPM rely on the ADD and DBL operations, which in turn rely on finite field operations. The order in which these blocks are to be implemented follows from these dependencies: starting from the bottom and going up. Therefore, the design of the blocks will be presented from the bottom up as well.

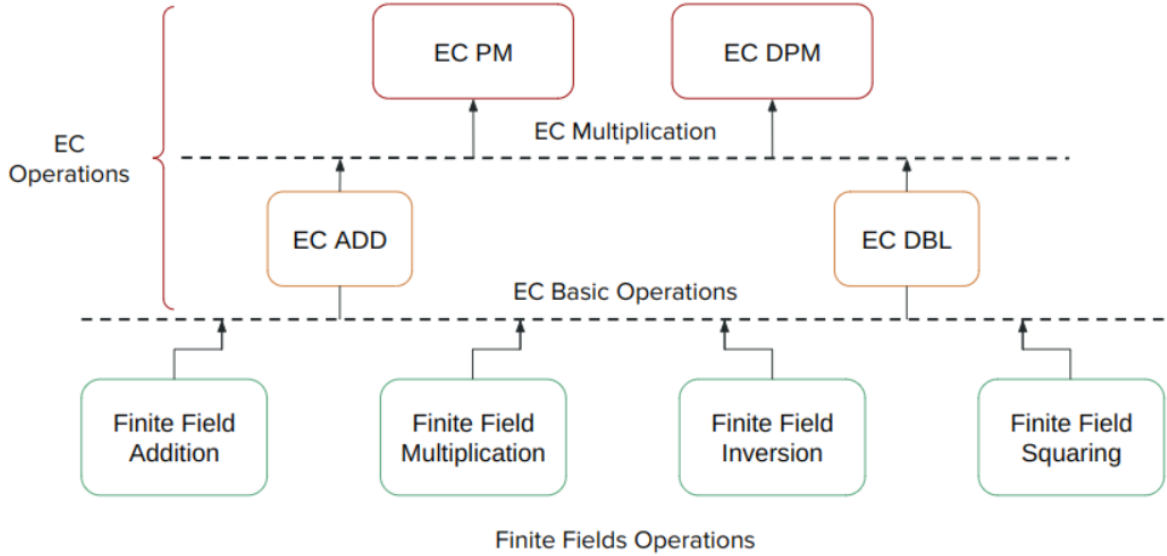


Figure 3.1: *Elliptic curve operations and their dependencies.*

The ECs recommended by NIST in [6] fall under two categories: curves defined over binary fields (a special case of extension fields), and curves defined over prime fields. The arithmetic of the operations over each type of field differs. According to [2], binary fields are faster for hardware implementations. Since our application is intended for later integration of hardware blocks for the EC multiplication, binary fields were chosen. Therefore, the EC operations, and subsequently, the cryptographic system is designed to operate on the NIST recommended binary curves: B-163, B-233, B-283, B-409, B-571.

The design of the addition, multiplication, and inversion binary field operations was based on the algorithms provided in [3] and previously explained in chapter 1. Field squaring can be achieved by multiplying the scalar by itself. Therefore, the design of an optimized finite field squaring operation is not necessary but would affect the cost of the EC operations in terms of speed and memory consumption.

3.2.1 Elliptic Curve Doubling and Addition

The design of The EC ADD and DBL operations depends on the coordinate system used to represent the points on the EC. As established in the chapter 1 through Tables 1.2, and 1.1, using DL coordinates for EC DBL, and mixed coordinates for EC ADD, best optimizes the cost in terms of binary field operations. Therefore, our design of these blocks will be based on Projective DL-coordinate system formulas for these operations. The algorithms we will present for EC DBL and EC ADD can be found in the appendix of the paper [15]. The binary field operation blocks were introduced as building blocks for the EC DBL and ADD operations.

3.2.1.1 Point Doubling

The DBL operation is done according to Algorithm 3.1. Starting by checking the input point P to determine if it is the point at infinity (∞). If so, the result of the doubling operation is also the point at infinity. Next, the slope of the tangent line at the input point is calculated using the curve equation and the coordinates of P . This slope is then used to find the intersection point of the tangent line with the elliptic curve. The coordinates of the doubled point are computed based

on this intersection point as specified by eqs.(1.22), (1.23), (1.24) in section 1.5.2. The resulting coordinates of the doubled point, are returned as the output of the point doubling operation.

Algorithm 3.1 Point doubling ($y^2 + xy = x^3 + ax^2 + b$, $a \neq 0, 1$, LD coordinates)

Input: $P = (X_1 : Y_1 : Z_1)$ in LD coordinates on $E/K : y^2 + xy = x^3 + ax^2 + b$

Output: $2P = (X_3 : Y_3 : Z_3)$ in LD coordinates

```

1: if  $P = \infty$  then
2:   return( $\infty$ ).
3:  $T_1 \leftarrow Z_1^2$ .
4:  $T_2 \leftarrow X_1^2$ .
5:  $Z_3 \leftarrow T_1 \cdot T_2$ .
6:  $X_3 \leftarrow T_2^2$ .
7:  $T_1 \leftarrow T_1^2$ .
8:  $T_2 \leftarrow T_1 \cdot b$ .
9:  $X_3 \leftarrow X_3 + T_2$ .
10:  $T_1 \leftarrow Y_1^2$ .
11: if  $a = 1$  then
12:    $T_1 \leftarrow T_1 + Z_3$ .
13:  $T_1 \leftarrow T_1 + T_2$ .
14:  $Y_3 \leftarrow X_3 \cdot T_1$ .
15:  $T_1 \leftarrow T_2 \cdot Z_3$ .
16:  $Y_3 \leftarrow Y_3 + T_1$ .
17: Return( $X_3 : Y_3 : Z_3$ ).

```

3.2.1.2 Point Addition

The process of point addition involves combining two input points, P and Q , to obtain a new resulting point. The coordinates of the input points are represented as $(X1, Y1, Z1)$ and $(X2, Y2, 1)$ respectively. The Arithmetic behind this algorithm is explained in chapter 1, section 1.5.2, according to the work done in [15].

Algorithm 3.2 Point addition ($y^2 + xy = x^3 + ax^2 + b, a \in 0, 1$, LD-Affine coordinates)

Input: $P = (X_1 : Y_1 : Z_1)$ in LD coordinates, $Q = (x_2, y_2)$ in Affine coordinates on $E/K : y^2 + xy = x^3 + ax^2 + b$

Output: $P + Q = (X_3 : Y_3 : Z_3)$ in LD coordinates

- 1: **if** $Q = \infty$ **then**
- 2: return(P)
- 3: **if** $P = \infty$ **then**
- 4: return($x_2 : y_2 : 1$)
- 5: $T_1 \leftarrow Z_1 \cdot x_2$.
- 6: $T_2 \leftarrow Z_1^2$.
- 7: $X_3 \leftarrow X_1 + T_1$.
- 8: $T_1 \leftarrow Z_1 \cdot X_3$.
- 9: $T_3 \leftarrow T_2 \cdot y_2$.
- 10: **if** $X_3 = 0$ **then**
- 11: **if** $Y_3 = 0$ **then**
- 12: use Algorithm 3.1 to compute:
- 13: $(X_3 : Y_3 : Z_3) = 2(x_2 : y_2 : 1)$ and return $(X_3 : Y_3 : Z_3)$.
- 14: **else** return(∞).
- 15: $Z_3 \leftarrow T_1^2$.
- 16: $T_3 \leftarrow T_1 \cdot Y_3$.
- 17: **if** $a = 1$ **then**
- 18: $T_1 \leftarrow T_1 + T_2$.
- 19: $T_2 \leftarrow X_3^2$.
- 20: $X_3 \leftarrow T_2 \cdot T_1$.
- 21: $T_2 \leftarrow Y_3^2$.
- 22: $X_3 \leftarrow X_3 + T_2$.
- 23: $X_3 \leftarrow X_3 + T_3$.
- 24: $T_2 \leftarrow x_2 \cdot Z_3$.
- 25: $T_2 \leftarrow T_2 + X_3$.
- 26: $T_3 \leftarrow T_3 + Z_3$.
- 27: $Y_3 \leftarrow T_3 \cdot T_2$.
- 28: $T_2 \leftarrow x_2 + y_2$.
- 29: $T_3 \leftarrow T_1 \cdot T_2$.
- 30: $Y_3 \leftarrow Y_3 + T_3$.
- 31: **Return**($X_3 : Y_3 : Z_3$)

3.2.2 Elliptic Curve Multiplication using Radix- 2^w Methods

In order to describe the design for Radix- 2^w EC PM and DPM, we will rely on the Radix- 2^w recoding method given in chapter 1.

3.2.2.1 Radix- 2^w Elliptic Curve Point Multiplication Design

Both the EC and Radix- 2^w EC PM parameters can be determined from the multiplication scalar k bit-length. According to eqs.(1.36), and (1.32), the bit-size of the window w determines Radix- 2^w recoding parameters. In addition, w is computed from the scalar k bit-length l according to eq.(1.34). The bit-length l of the multiplication scalar k is fixed for each of the

NIST-recommended binary curves, therefore, the sizes of w were computed for each one of the curves. The results are shown in Table 3.1. The optimal value for w is 5 bits for keys of bit-sizes 163 bits and 233 bits, and 6 bits for keys of bit-sizes 283 bits, 409 bits and 571 bits. These key sizes correspond to NIST EC curves B-163, B-233, B-283, B-409 and B-571 respectively. It follows that the relationship between the EC parameters, Radix- 2^w EC PM parameters, and multiplication scalar k 's size allows the set-up of all necessary parameters given one variable only.

bit-length(l)	163	233	283	409	571
w_{min}	5 bits		6 bits		

Table 3.1: Radix- 2^w configuration parameters for NIST recommended binary field elliptic curves [4].

In the context of EC cryptographic systems, the field scalars bit-length would be determined by the desired key size. The size of the key is expected to be chosen and introduced to the cryptographic system. Therefore, it entails that all EC multiplications are to be performed on that scalar length.

Figure 3.2 shows how the key size is involved with the blocks making-up and surrounding Radix- 2^w EC PM. The multiplication relies on the key size to determine the EC curve parameters and w window size which in turn determine the Radix- 2^w recoding parameters. Since the NIST-recommended binary ECs require window sizes of either 5 or 6 bits, it was more efficient to compute and store a recoding set 1.32 look-up table for each w size. The Look-up table used for the multiplication would be determined according to the size of the window w .

Similarly, given a multiplication point P , the precomputation set can be determined using the key size. Precomputation process block generates the precomputation set according eq.(1.37) as explained in chapter 1. This method for precomputation aims to minimize the number of ADD operations. It is noteworthy that since the precomputation process depends on the multiplication point P , it will be required prior to the multiplication process. However, in EC digital signature generation, the multiplication is always performed on the generator point of the EC. Thus, only running this process once suffices to perform signature generation for as long as the key size and curve parameters remain fixed. Therefore, not including the precomputation block in the multiplication block reduces the cost of the overall EC PM in this manner. The actual Radix- 2^w EC PM of k and P can be performed given the precomputation set and the recoding set shown in green in Figure 3.2.

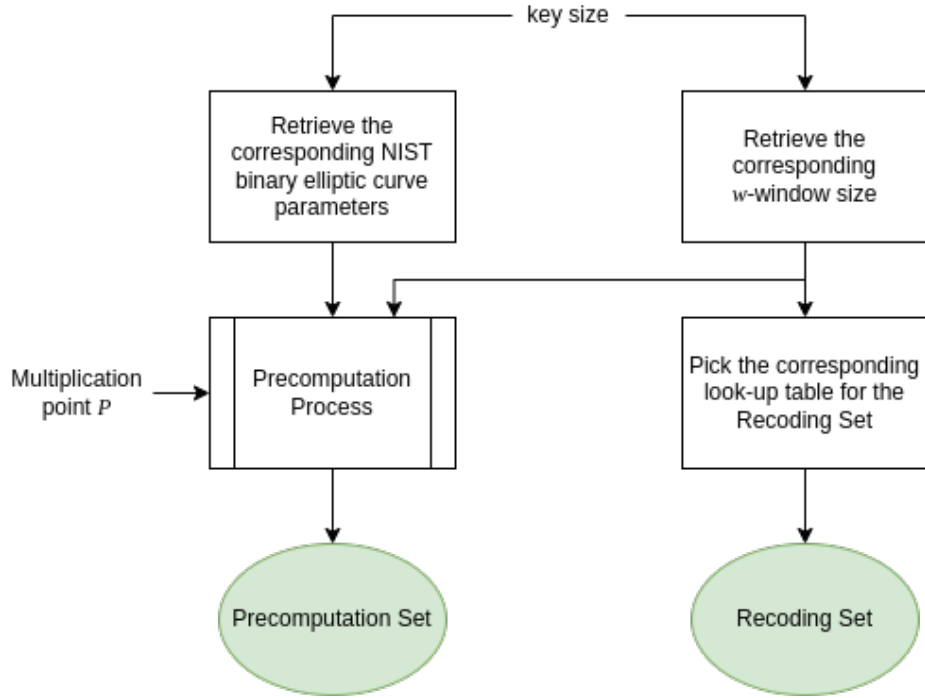


Figure 3.2: Block diagram of the Radix- 2^w EC PM precomputations.

An important feature of this EC PM method is the recoding and multiplication being performed on-the-fly as described in Algorithm 1.5. First, a slice Q must be obtained and subsequently used to retrieve m , and n from the look-up table which dictate the employment of EC ADD and DBL with respect to an accumulator. This process is repeated for all slices and the accumulator would equal kP by the end. The block shown in Figure 3.3 was designed to retrieve each slice Q by windowing the scalar k . A scalar whose binary value from the right equals w ones. This scalar is a slice sized window. It is shifted such that its right most "1" matches slice Q 's Least Significant Bit (LSB). An AND operation is performed on k and the scalar to obtain a shifted slice Q . The initial shifting is then reversed to obtain a slice Q . The value of Q is therefore used to retrieve m and n from the look-up table. EC operations are then performed according to the values of m and n . For non-zero values of m , an EC ADD is performed, the point added to the accumulator depends on the value of m . For non-zero values of n , an EC DBL of the accumulator is performed. In this manner, the recoding of k does not require to be stored then fetched again for the multiplication.

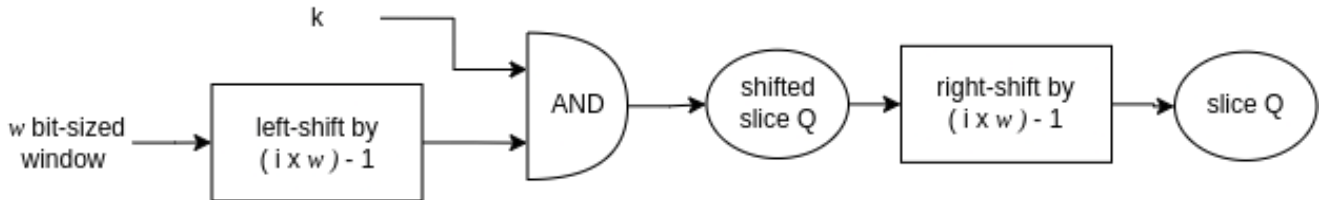


Figure 3.3: Design of the block that retrieves a slice Q .

The resulting point of the EC PM must be converted to Affine coordinates since our design of EC ADD and DBL output points in DL-coordinate system for optimization purposes. The conversion involves inversion field operations. This is due to a point $P(X, Y, Z)$ with $Z \neq 0$

in projective LD coordinate system corresponding to the Affine $P(x, y) = (X/Z, Y/Z^2)$. Based on this formula, our design of the full conversion block requires an inversion, a squaring, and 2 multiplications. First, the inverse of Z , Z^{-1} is found and stored in a temporary variable. Then, the inverse is squared to obtain Z^{-2} . Finally, X is multiplied by Z^{-1} and Y by Z^{-2} , and Z either set to 1 or eliminated completely. Thus, the result of EC PM can be obtained in Affine coordinates using the Radix- 2^w method.

As EC ADD takes mixed DL-Affine coordinates, one of the input points must always be in Affine coordinates before the operation. This may lead to extensive use of the DL-Affine conversion block within the multiplication. To avoid this, the precomputed values must all be converted to Affine within the precomputation process before being stored. This is due to the multiplication algorithm 1.5 only using ADDs with the precomputed values. This ensures at least one point in Affine coordinates at every ADD.

3.2.2.2 Radix- 2^w Elliptic Curve Double Point Multiplication Design

The Radix- 2^w EC DPM method proposed in [9] and discussed in chapter 1, section 1.7.3 has a fixed window size w of 2 unlike the EC PM method. Another difference is its use of the values of the slices directly without requiring to retrieve values from the recoding set. This means that the same method for obtaining the slices from the scalar shown in Figure 3.3 applies. The multiplication process depends on the key size only in obtaining the EC parameters as shown in Figure 4.9.

The precomputation set for EC DPM depends on the two points involved in the multiplication, and is performed in the order described in line 1 of Algorithm 1.6. However, unlike the EC PM, there is no benefit to excluding the precomputation process from the EC DPM block as the points are not fixed for any of the system protocols. The precomputed values are converted to Affine coordinates before storage for the same reasons mentioned in the EC PM design discussion and the result of the multiplication is given in Affine coordinates. In the Radix- 2^w EC DPM, operations are performed on each iteration according to a pair of slices, each corresponding to a point. Similarly to EC PM, the recoding and multiplication must be performed on-the-fly and by the end of the multiplication, the result must be converted to Affine coordinates.

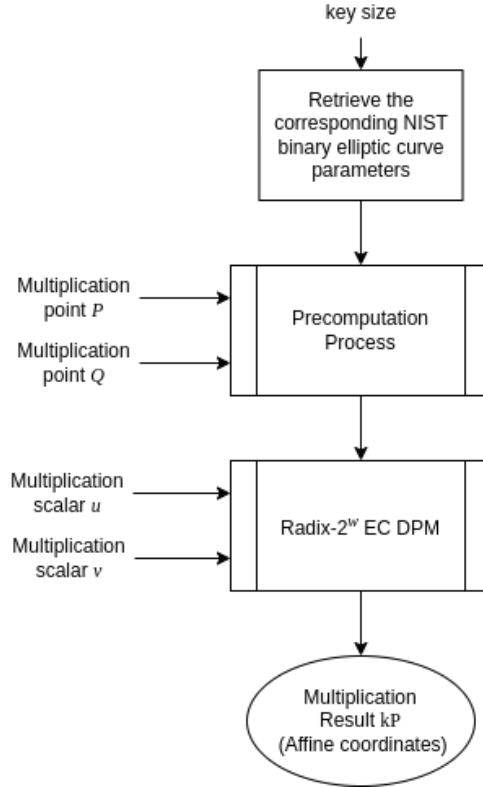


Figure 3.4: Block diagram of the Radix- 2^w EC DPM precomputations.

3.2.3 Binary Method for Elliptic Curve Point Multiplication

The classical binary method for EC PM will be used for cost comparison with the Radix- 2^w EC PM and EC DPM methods. The binary EC PM uses the binary representation of the scalar k in the multiplication. It can be performed from either left-to-right or right-to-left. Algorithms 3.3 and 3.4 show the methods used to achieve binary multiplication. In both algorithms an accumulator is used in a loop to collect the sum of the points representing each significant digit "1" in the number's binary representation. The difference between the two binary multiplication strategies is the DBL operation is performed on different points. In both instances the DBL operation serves to account for the analogous shift operation for the binary represented scalar k . In the right-to-left method, the added point P gets doubled with every iteration, accounting for the power of the "1" digits that are added to the sum Q . On the other hand, in the left-to-right method, the accumulator point Q gets doubled with every iteration

Algorithm 3.3 Left-to-Right binary method for point multiplication

Input: $k = (k_{t-1}, \dots, k_1, k_0)$, and $P \in E(\mathbb{F}_{2^m})$

Output: $kP \in E(\mathbb{F}_{2^m})$

- 1: $Q \leftarrow \infty$
 - 2: **for** $i = t - 1$ **downto** 0 **do**
 - 3: $Q \leftarrow 2Q$ ▷ Point DBL
 - 4: **if** $k_i = 1$ **then**
 - 5: $Q \leftarrow Q + P$ ▷ Point ADD
 - 6: **Return**(Q).
-

Algorithm 3.4 Right-to-Left binary method for point multiplication

Input: $k = (k_{t-1}, \dots, k_1, k_0)$, and $P \in E(\mathbb{F}_{2^m})$

Output: : $kP \in E(\mathbb{F}_{2^m})$

```
1:  $Q \leftarrow \infty$ 
2: for  $i = 0$  to  $t - 1$  do
3:   if  $k_i = 1$  then
4:      $Q \leftarrow Q + P$  ▷ Point ADD
5:    $P \leftarrow 2P$  ▷ Point DBL
6: Return( $Q$ ).
```

3.3 Cryptographic System Design

The ECDSA cryptosystem, as previously mentioned in chapter 2, relies on public key pairs (Q, d) . Thus, a random number generator block is needed to generate the private key d . The implemented EC PM design is then used to generate the public key Q by applying the multiplication on the randomly generated private key and a point on the EC, as described in Algorithm 2.1. This process is also used in the EC-ElGamal protocol to generate the keys to be exchanged between the communicating parties, and used in the encryption scheme. The random number generator block also generates the nonce required in the signature generation process, described by Algorithm 2.2. This algorithm, as well as Algorithm 2.3 describing the signature verification process, rely on hashing. The hash function chosen for implementation is the "SHA256" function. This function was chosen as it is one of the four fixed-length SHA-3 algorithms approved by the NIST secure hash standard [16].

Another block needed is a mapping table generator. This block takes in the EC parameters associated with the binary curve chosen, and generates a mapping table that would subsequently be used for the encryption scheme. This process is illustrated in Figure 3.5. The message encryption block takes as input the receiver's public key Q_B , the message (made up of a set of characters "m"), and the sender's key-pair (d_A, Q_A) and produces the encrypted message. To perform the encryption, the mapping P_m of the characters m are retrieved one by one and encrypted individually. The decryption process relies on the same mapping look-up table as described in Algorithm 2.5. The same original message is also signed by the signature generation block using the key-pair (d_A, Q_A) . At the end of the encryption and signature generation process, the resulting encrypted message and message signature are then transferred to the receiver to be decrypted then verified.

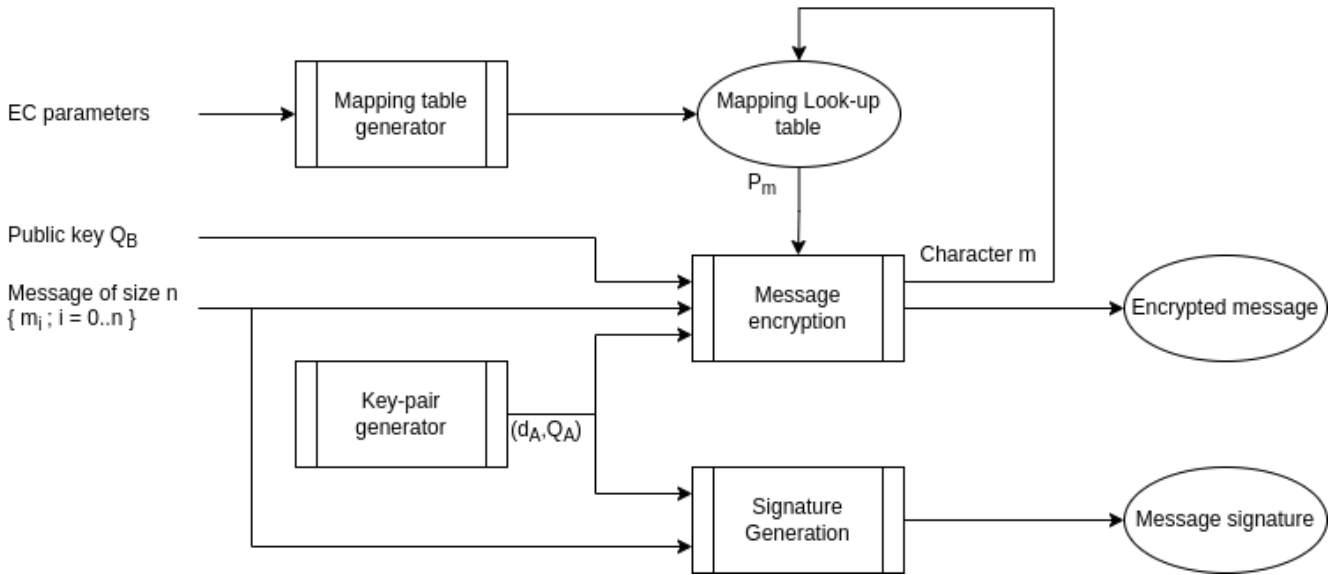


Figure 3.5: Block diagram of the joint EC cryptographic system.

3.4 Conclusion

In this chapter, we have covered the cryptographic system architecture and its elements. We have focused on the Radix- 2^w multiplication design and the components necessary for its implementation as it is the backbone of our project. We have also introduced the binary method for EC PM as we will be using it for cost comparison with the Radix- 2^w multiplication methods. Finally, we have shown the joint system design. In the following chapter, we will discuss the implementation details of the previously reviewed designs and the evaluation of our solution.

Chapter 4

EC Signature/Encryption System Implementation and Results

4.1 Introduction

The ECDSA implementation flow is two fold: writing the programs necessary and debugging on a private computer, then, running and debugging on the Board. This was a more convenient method in terms of testing the different code blocks. Since these blocks operate on long numbers (keys of bit-lengths 164, 233, etc.), it is necessary to test for large samples of randomly generated integers. This makes it significantly faster to run these tests on a laptop. CLion is an Integrated Development Environment (IDE) for C/C++ languages that was used for the development of the software solution for ECDSA and EC-ElGamal encryption, which were to be later fitted for the ZedBoard. CLion IDE was chosen for its support of development for various platforms, including embedded systems and cross-compilation¹, making it suitable for our application. However, to use it as such, it must be configured to work with the ZedBoard by setting up the appropriate toolchain² and build configurations. The lack of a direct built-in support for ZedBoard on Clion made Xilinx's³ software development kit a more suitable choice for running and debugging the software on the board.

In this chapter, we will start by introducing the ZedBoard, touch on its relevant functionalities and briefly explain the overview of the full system in light of the used board. Next, we delve into the details of the implementation. We will introduce the library we used for multiple precision arithmetic⁴ as the large sizes of the keys necessitate. Then, we will describe a global overview of the implemented cryptographic system highlighting relevant details proper to our implementation. Following that, we will delve into the Radix- 2^w EC multiplications; where we touch on some details concerning our implementation of the different layers of arithmetic preceding the multiplication. Subsequently, we will describe the methods we employed for testing the Radix- 2^w method and discuss the obtained results. Afterwards, we will describe the specifics of the implementation

¹Creating executable code for a platform other than the one on which the compiler is running.

²A set of programs, to build on, that are used for specialized (related) software development.

³A leading provider of programmable logic devices and associated technologies, as well as development tools, software libraries, and intellectual property cores to facilitate the design and implementation of digital systems using their products.

⁴Calculations performed on numbers that can have a limitless number of digits, constrained only by the memory capacity of the computer.

of the protocols and the user interface. Next, we will show an illustrative example on message encryption and signing. Finally, we will review the modifications made to adapt the cryptographic system to the ZedBoard.

4.2 Hardware Environment

4.2.1 Zynq Evaluation and Development Board

The ZedBoard is a FPGA development board based on the Xilinx Zynq-7000 All Programmable (AP) System On Chip (SoC). It is comprised of two main parts: a Processing System (PS) formed around a dual-core Advanced Reduced Instruction Set Computer Machine (ARM) Cortex-A9 processor, and Programmable Logic (PL) consisting of FPGA fabric. The ARM Cortex-A9 is an application grade processor, capable of running full operating systems such as Linux, while the programmable logic is based on Xilinx 7-series FPGA architecture. Thus, allowing development of both hardware and software components on the same device. It also features several memory and storage options, including Double Data Rate3 (DDR3) Synchronous Dynamic Random-Access Memory (SDRAM) for running software applications, flash memory for booting the system, and a micro-SD card slot for expandable storage, The ZedBoard also offers a wide range of peripherals for connectivity and I/O operations. The interface between these different elements within the Zynq architecture is based on the Advanced eXtensible Interface (AXI) which provides high bandwidth and low latency connections.

To connect the board to a host PC (Personal Computer), the following two ports are utilized:

- **USB-JTAG port:** It allows for high-speed data transfer between the ZedBoard and the host computer, enabling efficient programming and debugging operations. This Universal Serial Bus (USB) port follows the Joint Test Action Group (JTAG) protocol. This protocol provides commands and signals to communicate with and control the internal components of the ZedBoard.
- **USB-UART port:** It is primarily used for establishing a serial communication link between the ZedBoard and a host computer. It enables the exchange of data in a serial format, allowing for various applications such as console output, data transfer, and control between the ZedBoard and the host. This USB port follows the Universal Asynchronous Receiver / Transmitter (UART) protocol, which is a common asynchronous serial communication protocol. It provides features such as configurable baud rate, data bits, stop bits, and parity for flexible communication settings.

Figure 4.1 illustrates the placement of these two ports, as well as the layout of the remaining elements of the ZedBoard.

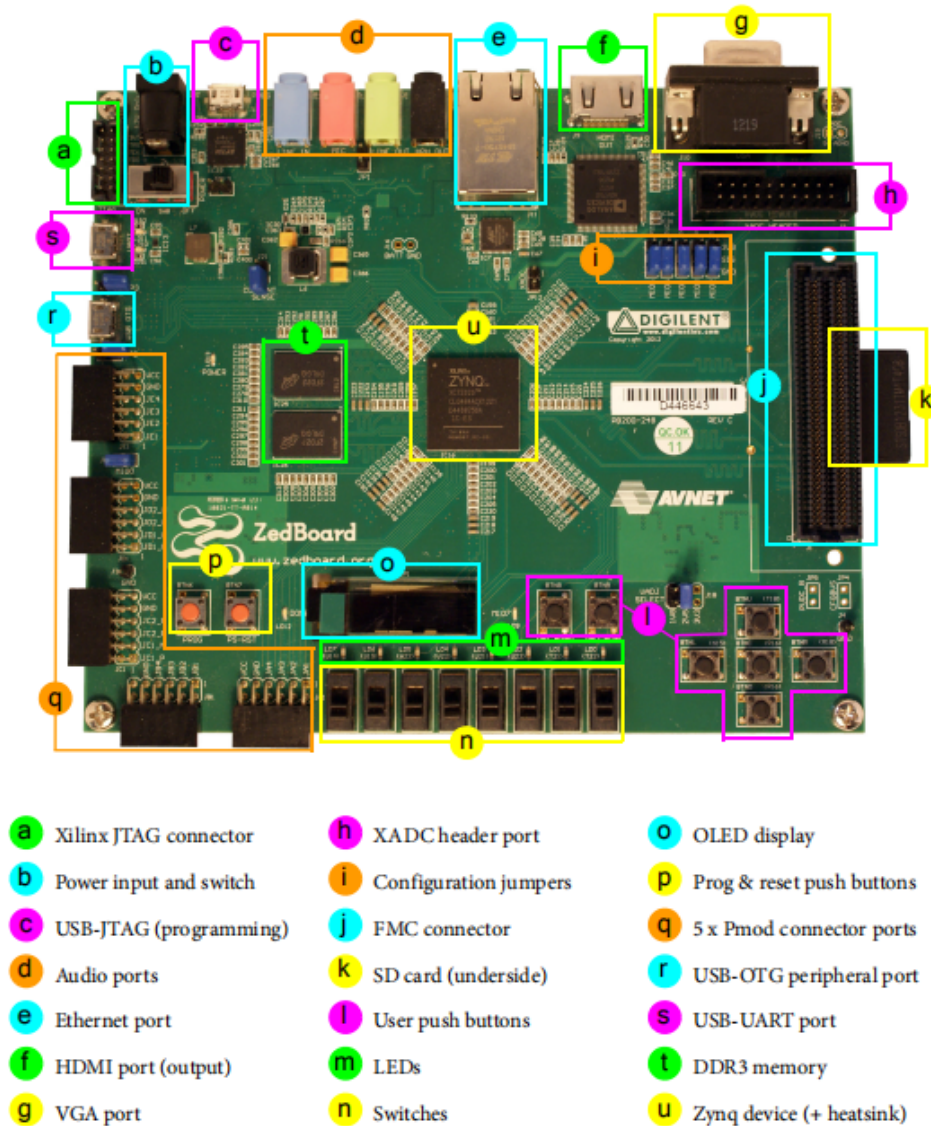


Figure 4.1: *ZedBoard layout*[5].

To design and develop the hardware and software components of the ZedBoard, separate IDEs were used. These being Vivado and SDK IDEs, which are both tools developed by Xilinx and are commonly used for FPGA development, including the ZedBoard. Vivado is used for FPGA design and implementation, while SDK is used for embedded software development targeting the processors within the ZedBoard. Together, they enable the development of complex systems that combine both hardware and software components. The typical workflow involves designing the hardware system in Vivado; then, in SDK, creating and configuring software projects to communicate with the custom hardware implemented in Vivado.

4.2.2 Development Tools

4.2.2.1 Vivado

Vivado is an IDE used for designing, implementing, and programming Xilinx FPGAs and SoCs. It provides a graphical interface for creating FPGA designs, performing synthesis, placement, routing, and generating programming files. Vivado supports both hardware description lan-

guages (HDLs) VHDL and Verilog, as well as high-level synthesis (HLS) languages like C/C++. It also includes Intellectual Property (IP)⁵ cores and a wide range of debugging and verification tools.

For this implementation, Vivado was used to customize the processing system. This was achieved through the following steps:

- **Launching Vivado:** Opening the Vivado IDE and creating a project.
- **Creating the block design:** Once the project is open, "Create Block Design" can be selected under the "IP Integrator" section of the "Flow Navigator" pane.
- **Selecting the PS IP:** In the IP Integrator tool, the "Zynq Processing System" IP is selected from the list of IPs and added to the block design.
- **Modifying the selected IP's Configuration:** By double-clicking on the added "Zynq Processing System" IP block, its configuration dialog is opened, where it can be customized. In the "PS-PL Configuration" tab, the AXI master interface can be unchecked. This is done because, as mentioned, the PL portion is not integrated in this implementation. Additionally, the SD card can be selected in the "Peripheral I/O Pins" tab to allow for its detection and use.
- **Generating the bitstream:** This consists of selecting the "Generate Bitstream" in the "Flow Navigator" pane. Bitstream generation combines the synthesized and implemented design into a binary bitstream file that can be programmed onto the FPGA. Vivado automatically runs synthesis and implementation as part of the bitstream generation process. Synthesis converts the design into a gate-level representation suitable for implementation on the FPGA. Implementation involves mapping the synthesized design onto the target FPGA, including placement and routing.

The generated bitstream file can then be saved and exported to a software development IDE, that being Xilinx SDK in this implementation.

4.2.2.2 Xilinx SDK

SDK (Software Development Kit), also known as Xilinx SDK, is also an IDE that is part of the Xilinx Vivado Design Suite. It is specifically designed for embedded software developments targeting Xilinx FPGAs and SoCs. It provides a set of tools, libraries, and compilers to develop and debug embedded software that interacts with the FPGA fabric and peripherals on the board. SDK supports programming in languages such as C, C++, and Assembly. It also offers code editing, project management, debugging, and software performance analysis features. The applications developed on SDK run on Xilinx processing systems and can be designed to interact with other programmable hardware blocks or firmware.

4.3 Software System Implementation

4.3.1 Global System Overview

In light of the functionalities of the zedboard and its PS system, the use of serial communication and the SD card was employed to turn the ECDSA/EC-ElGamal encryption system design into

⁵IPs are pre-designed functions that can directly be included in FPGA designs

a functional application. It is illustrated in Figure 4.2.

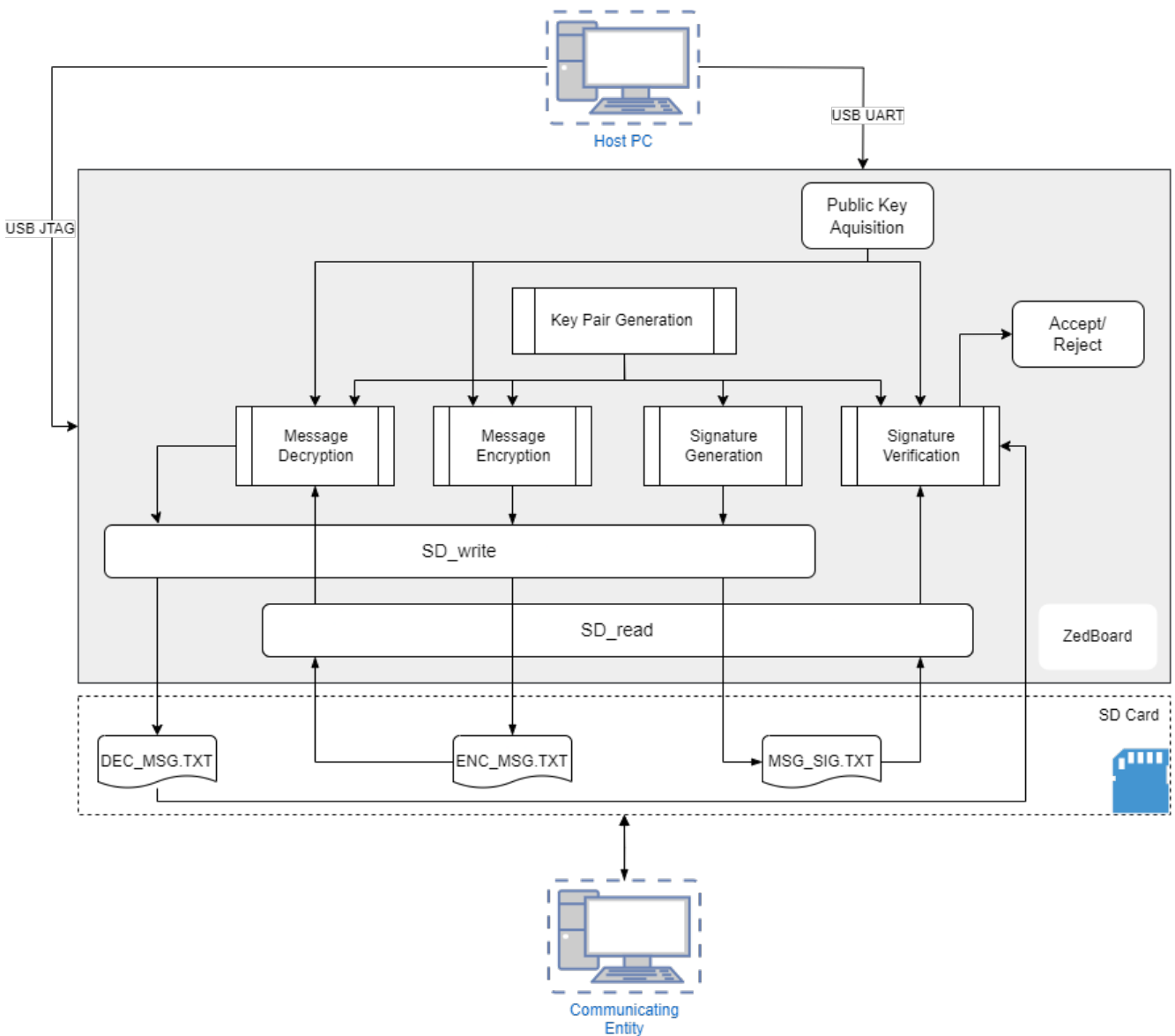


Figure 4.2: *Global system layout.*

The flow of operation consists first of the public key acquisition; where said public keys of the communicating entities are exchanged through serial communication with the board. These public keys are used for the cryptographic protocols described in chapter 3 (ECDSA and EC-ElGamal). The outputs of these protocols are exchanged between said entities to run the processes contained within the system. Typically, this is achieved through a secured public channel. In our implementation, this channel is emulated through an SD card which transports said outputs between the board and the communicating entity; for example, a PC. The SD card contains text files that hold the outputs. For ECDSA, the file used to hold the signature is "*MSG_SIG.TXT*". In signature generation, the generated signature is written into this file. While in signature verification, it is read to retrieve the signature which is then verified. As for the EC-ElGamal encryption protocol, two files are assigned: "*DEC_MSG.TXT*", and "*ENC_MSG.TXT*". When encrypting a message, the resulting ciphertext is written into "*ENC_MSG.TXT*". This file is then read in the decryption process to retrieve the ciphertext and decrypt it. Once decrypted, the

message is then stored into "*DEC_MSG.TXT*". This file is also used in the signature verification; where it would be read to retrieve the message which is then used as input for this process.

The computer version of this system implementation was initially realised. In the next section we will discuss the basic representations of numbers and points that are the foundation of our implementation and how they were achieved.

4.3.2 Scalars and Points Representation

Our implementation requires programming to be done using either C or C++. This is due to the application being made for the ZedBoard processing system. Therefore, our implementation was restricted to having native data types that do not exceed 64 bits. Since the key sizes used are significantly large, we faced issues storing and operating on scalars of this size (i.e., 164 bits, 233 bits, etc.). Initially, the size of the keys used was fixed to 233 bits allowing the use of arrays to store their values. This array method required the implementation of partial operations. Carries, the result of the operation, and conditions on some bits are all taken into account with this method. The implementation of such operations required additional functions to locate specific bits, intermediate variables for carries and counters, as well as the heavy use of loops. There were two major disadvantages to this approach: The inflexibility of key lengths, and the complexity of these basic operations (addition, multiplication, subtraction, inversion). With the increased difficulty to implement and build on these function, a library that would provide finite field operations on large scalars seemed a good alternative. However, these libraries often had the issue of limited sized scalars. Another solution was using a library that could handle operations on large scalars without approximating the resulting values. One such library is the GNU Multiple Precision (GMP) library . Details concerning all the functionalities GMP provides can be found on their website [17].

Since this application requires the use of signed integers to represent keys, we will be focusing on the integer category of functions called "mpz". This category allows for a long list of arithmetic and logic operations to be performed. A variable of type *mpz_t* is used to store the value of an integer through a string and a specified number base (i.e, string = "857B68A0f", base = 16 would interpret the string as a hexadecimal number and store it in the variable). This use of strings made the use of C++ as a programming language more advantageous for this implementation.

The used GMP library does not automatically run on the ZedBoard processor. However, the cross-compilation of this library was possible. This library was cross-compiled on a UNIX host using instructions from [18]. It is note worthy that the configuration of this library requires setting up the "*CFLAGS*" (compiler flags) correctly for the specific processor. For our case the following flags were used in the command for the cross-compilation:

```
./configure CC=arm-none-eabi-gcc CFLAGS="-nostartfiles --specs=nosys.specs  
-mtune=cortex-a9 -mfpu=vfpv3 -mfloat-abi=hard -mcpu=cortex-a9"  
--host=arm-none-eabi --disable-assembly
```

The *CC=arm-none-eabi-gcc* targets the ARM architecture and is used for standalone applications. The flags specified in the configuration correspond to the ZedBoard processor. Figure 4.3 shows the result of this process.

```

Version:          GNU MP 6.2.1
Host type:       arm-none-eabi
ABI:            32
Install prefix: /usr/local
Compiler:       arm-none-eabi-gcc
Static libraries: yes
Shared libraries: no

```

Figure 4.3: Result of the cross-compilation of GNU MP library for ARM Cortex-A9 processor.

As for the EC point representation, to start off, an EC must be selected from the NIST-recommended binary curves in order to run the implemented cryptographic system. Each of the curves is defined by its binary field's primitive polynomial, the coefficients a and b of the curve equation (eq.(1.2)), and the generator point of the curve G . All parameters for each curve were retrieved from [6]. Figure 4.4 shows an example of the parameters provided for the B-283 EC where $f(z)$ is the underlying binary field's primitive polynomial, a and b are the coefficients, and finally G_x and G_y are the Affine coordinates for the generator point.

```

f(z):  z283 + z12 + z7 + z5 + 1
h:     2
n:     7770675568902916283677847627294075626569625924376904889\
      109196526770044277787378692871
      (=0x3fffffff ffffffff ffffffff ffffffff ffffef90 399660fc
        938a9016 5b042a7c efadb307)
tr:    2863663306391796106224371145726066910599667
      (= (2m+1) - h · n = 0x 20df8cd33e06d8eadfd349f7ab0620a499f3)
a:     1
      (=0x00000000 00000000 00000000 00000000 00000000 00000000
        00000000 00000000 00000001)

Polynomial basis:
b:     0x27b680a c8b8596d a5a4af8a 19a0303f ca97fd76 45309fa2
      a581485a f6263e31 3b79a2f5
Gx:  0x5f93925 8db7dd90 e1934f8c 70b0dfec 2eed25b8 557eac9c
      80e2e198 f8cdbecd 0x86b12053
Gy:  0x3676854 fe24141c b98fe6d4 b20d02b4 516ff702 350eddb0
      826779c8 13f0df45 be8112f4

```

Figure 4.4: B-283 NIST-recommended binary elliptic curve parameters [6].

Overall, two classes were implemented: *EC_point*, and *KeyPair*, in order to simplify the programming:

1. ***EC_point***: this class was created to represent points on EC, and subsequently, facilitate the implementation of EC operations. Points on ECs are assumed to have DL-coordinates X, Y , and Z which are represented as *mpz_t* type attributes, with *mpz_t* being the multiple precision integer type from the GMP library. When an *EC_point* is instantiated, it takes the coordinates of the point at infinity by default. However, if the coordinates are provided for the constructor as strings, they are used instead. The methods for this class allow the values X, Y, Z to be modified, printed, compared, etc.

2. **KeyPair**: this class was created to facilitate the implementation of ECDSA and ElGamal encryption protocols as they both rely on Public-key cryptography. Every instance of this class has two attributes: a public key Q of the class EC_point , and a private key d of type mpz_t . When a *KeyPair* key-pair is generated, a randomly generated scalar is assigned to the private key d . As shown in Algorithm 2.1, the result of the multiplication of d with the EC generator G is assigned to the public key Q .

4.4 Radix 2^w Elliptic Curve Multiplication Implementation

The first layer of the implementation involves writing the functions for the binary field operation Algorithms 1.1, 1.2, and 1.3. Due to the nature of the data type mpz_t being a pointer, both operands and result variables are entered as parameters. Therefore, These functions performing binary field operations are void type. The second and third layers of the EC multiplication implementation build on the binary field operations functions, and their implementation follows.

4.4.1 Elliptic Curve Addition and Doubling Implementation

The DBL operation was implemented to operate on EC_point types (EC points respresented in DL coordinate system) as shown in Algorithm 3.1. On the other hand, the ADD operation, which was coded according to Algorithm 3.2, uses mixed Affine-DL coordinates. This means that its operands, $point_1$ and $point_2$, are represented in DL-coordinates and Affine coordinates respectively. In order for the ADD operation to function properly, $point_2$ must be in Affine coordinate representation. Therefore, whenever a point $S(X, Y, Z)$ is to be added to another point $A(X_a, Y_a, Z_a)$, for example, a conversion to $S_{Affine}(x_s, y_s, 1)$ is performed on S using the DL-Affine conversion function.

4.4.2 Elliptic Curve Point Multiplication Implementation

4.4.2.1 Binary Method for Point Multiplication Implementation

First, the function implementing the naive binary EC PM operation was written. The purpose of this implementation is to, both, examine the functionality of the implemented ADD and DBL, as well as provide later comparison to the Radix- 2^w method for multiplication. The binary method can be performed from both Left-to-Right (Algorithm 3.3) and Right-to-Left (Algorithm 3.4). The disadvantage of the Right-to-Left method is the necessity of performing coordinate conversion after every point DBL operation. This is due to the coordinates of the output point. Although the DBL operation takes a point in DL-coordinates as input, the ADD operation must have at least one point in Affine coordinates. Thus, requiring the extra cost for conversion. In order to avoid this extra cost, the binary Left-to-Right method shown in Algorithm 3.3 was chosen, as ADD operations involve adding a fixed point P to sum, thus, not necessitating coordinate conversion.

4.4.2.2 Radix- 2^w Method for Point Multiplication Implementation

Since the result of EC PM can not be judged correct through simple observation, some testing programs must be designed and implemented. In order to simplify the debugging, the writing of the Radix- 2^w EC PM was done in two steps. The Radix- 2^w recoding was tackled first as it is the

foundation of the Radix- 2^w EC PM. Having a functional recoding, the program for the EC PM was written second.

As the Radix- 2^w EC PM is performed on-the-fly, the processes for recoding and multiplication are fundamentally joint. Therefore, a simplification of the EC PM Algorithm 1.5 that focuses on the recoding part was programmed as shown in Figure 4.5. The simplified multiplication was achieved through equating points on the EC to simple scalars. Thus, the multiplied point P in the algorithm was replaced by a scalar $p = "1"$, and the accumulator R , storing the product, was equated to $r = 1 \times k$. Therefore, the value of the input scalar k should equal the value of output scalar r . The program for the simplified multiplication was written as described for the Radix- 2^w EC PM design in chapter 3, section 1.7.2, with the above mentioned modifications. It follows that eliminating the use of points entails the dismissal of the coordinate conversion process. The precomputed set of points is represented with an array containing the odd set, shown in eq.(1.31), in the simplified multiplication. The reason being that the precomputation set is made up of the products of point P with each value of the odd set, and by simplifying P to $p = 1$, the precomputation set is simplified to the odd set.

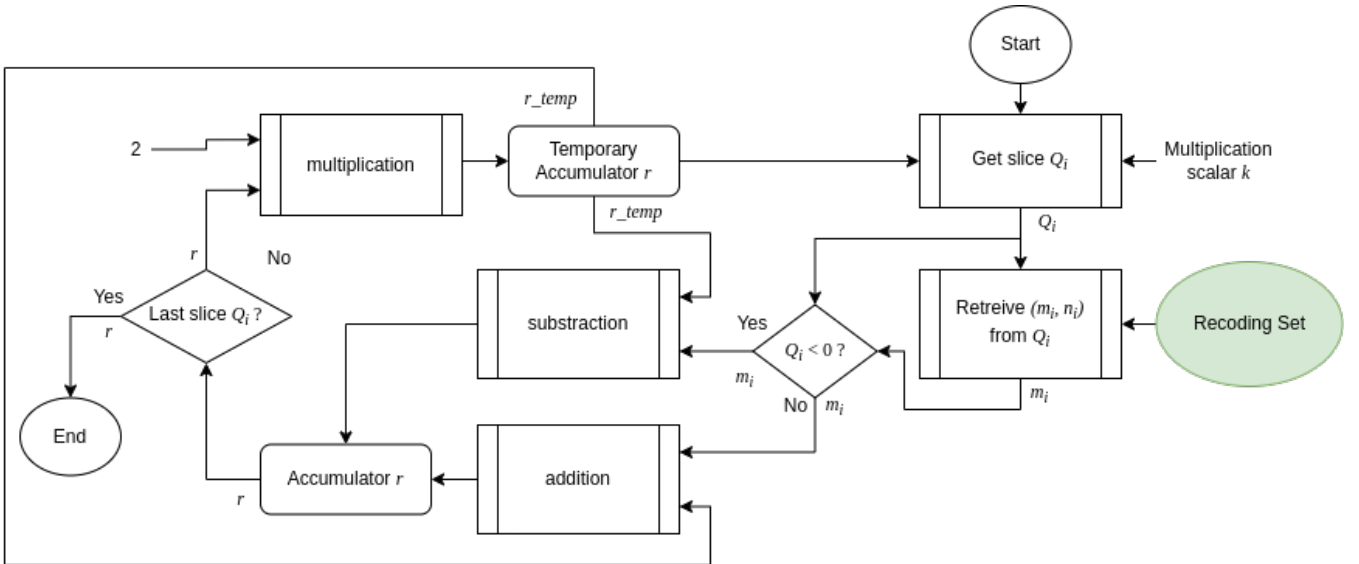


Figure 4.5: Block diagram of the simplified Radix- 2^w EC PM.

To ensure functionality of the simplified multiplication program, prior to the realisation of the Radix- 2^w method for EC PM, the code must be tested for a large number of scalars k . This is due to k having a sizable bit-length, therefore, numerous possible values. Given the impossibility of testing for every value of k , a small randomly generated subset is tested. The testing was performed through the bench test whose flowchart is shown in Figure 4.6. The function `mpz_urandomb()` belongs to the GMP library. It is a Random Number Generator (RNG) that takes 3 parameters: `rop`, `state`, and `n`, to generate a random number. The variable `rop` is the randomly generated scalar of type `mpz_t`, k in our case. The `state` variable defines the type of random number generation used depending on its initialization. The different types of initialization are detailed in the GMP library documentation in [19]. For our purposes, we used the default initialization, which provides a compromise between randomness and speed. The value of the variable `n` specifies the range that the randomly generated number falls into: 0 to 2^n-1 , inclusive. The randomly generated k is processed by the simplified multiplication block called "Simplified Radix- 2^w multiplication" to produce r . k and r are compared to ensure r equals $1 \times k$ for the above explained reason. The

longer this bench test runs for without error, the less likely there is an issue with the recoding. Both bench tests ran for approximately 24 hours each without error.

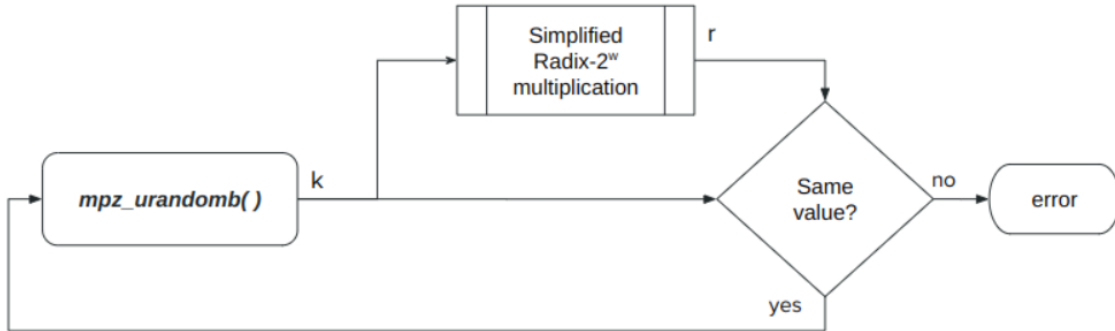


Figure 4.6: *Radix-w recoding test bench.*

Unlike the simplified multiplication, the Radix- 2^w EC PM requires a precomputation set of points. This set is stored in a global array of *EC_point* type variables. This allows the EC PM program to have access to the precomputed points' values. The precomputation process block shown previously in Figure 3.2 was programmed for the calculation of the precomputation set, shown in eq.(1.36), as described in eq.(1.37). It takes the point to be multiplied, the EC parameters, and the size of the window w as inputs. It suffices to call this function at the start of the EC PM to have all precomputed values available in the global array. Since every precomputed point $m \times P$ corresponds to a single odd value m , this odd value is used to retrieve the required point.

Given a fixed w , the recoding set is fixed as shown in eq.(1.32). It entails that each pair of the values m and n correspond to a specific slice value Q . Since Q is directly obtained from the scalar k , it is viable as index to obtain the appropriate values for m and n from a look-up table. As previously discussed in chapter 3, the values of w corresponding to NIST-recommended binary curves B-163 and B-233 is equal to 5, while that corresponding to B-283, B-409, B-571 is equal to 6. Therefore, two look-up-tables were made for each value of w . The tables are written as arrays where each table is made up of two arrays storing the sets of values for m and n . The arrays are ordered such that each m, n pair can be retrieved using the same index. An example for $w = 3$ (as the tables for $w = 5$, or $w = 6$, are large) is shown in Table 4.1, where each element in the m and n arrays are ordered such that they can be accessed by the corresponding value of the slice Q .

Q	m	n
0 0 0 0	0	0
0 0 0 1	1	0
0 0 1 0	1	0
0 0 1 1	1	1
0 1 0 0	1	1
0 1 0 1	3	0
0 1 1 0	3	0
0 1 1 1	1	2
1 0 0 0	1	2
1 0 0 1	3	0
1 0 1 0	3	0
1 0 1 1	1	1
1 1 0 0	1	1
1 1 0 1	1	0
1 1 1 0	1	0
1 1 1 1	1	2

Table 4.1: Radix-2³ recoding set look up table.

With both the precomputation and recoding set available, the Radix-2^w EC PM program was written according to Algorithm 1.5 as shown in Figure 4.7. The EC PM block is followed by the DL-Affine conversion block. The DL-Affine coordinate conversion is explained in chapter 3, section 3.2.2 and the function performing it was written accordingly.

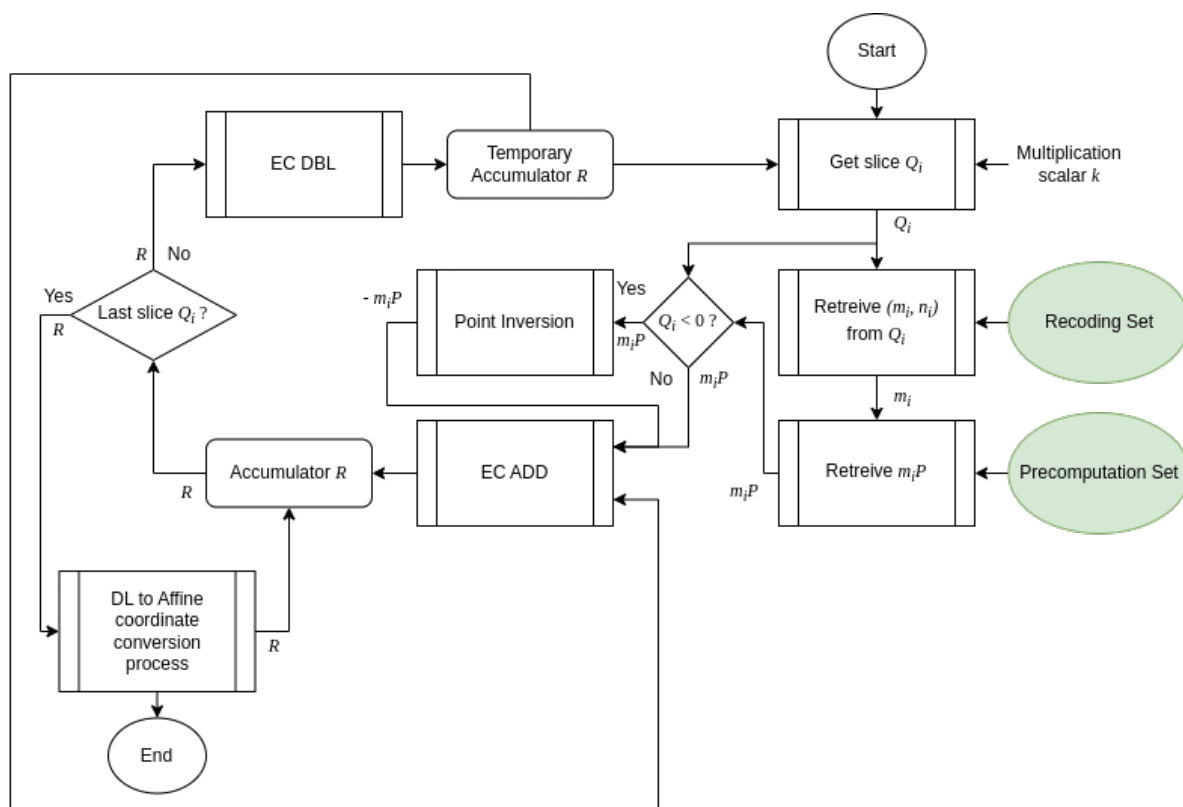


Figure 4.7: Block diagram of the Radix-2^w EC PM.

4.4.2.3 Radix- 2^w Method for Double Point Multiplication Implementation

Unlike the Radix- 2^w EC PM, the simultaneous method for Radix- 2^w EC DPM does not require the use of the recoding set. Therefore, m and n values are not relevant. Since w is fixed to two, the w window size limits the slice values to $2^3 = 8$ possible values: 000, 001, 010, 011, 100, 101, 110, 111. Evaluating the values of these slices according to eq.(1.27) yields the values: 0, 1, 1, 2, -2, -1, -1, 0. An array d containing the values 0, 1, 1, 2, 2, 1, 1, 0 is used to represent the absolute value of the slices given their binary value. The sign bit (right-most bit) is accounted for independently. Since the EC DPM involves two scalars u and v , a slice pair (du, dv) (corresponding to the scalars u and v respectively) is considered instead of a single slice Q .

Similarly to EC PM, a precomputation program was written and must be called before the start of the EC DPM computation. The result of the precomputation program is an array where the precomputed set shown in eq.(1.40) is stored. The combinations of the slice-pair values and their sign-bit indicate the order and operands that the ADD and DBL operations take as shown in Algorithm 1.6. Thus, a switch statement, taking the slice pair as parameter, was used to distinguish between cases. Figure 4.9 shows the implementation of the Radix- 2^w EC DPM. To implement the Radix- 2^w EC DPM, similarly to the EC PM, the simplified Radix- 2^w double multiplication program was written first, followed by the normal one. Given two scalars u and v and two points P and Q , the expected result is $u \times P + v \times Q$. By setting these points to unit scalar "1", the output is therefore $u + v$. In the context of Figure 4.9, the change from EC points to scalars translates to the removal of the point precomputation and the coordinate conversion blocks, as well as the replacement of the EC operations by scalar operations of equivalent function. To accommodate the simplification in the switch statement taking the slice pair (du, dv) , the EC ADD is turned into scalar addition, and EC DBL into scalar squaring. Thus, the simplification was implemented and subsequently tested, prior to the implementation of the normal Radix- 2^w EC DPM. Figure 4.8 shows the bench test done for the simplified multiplication.

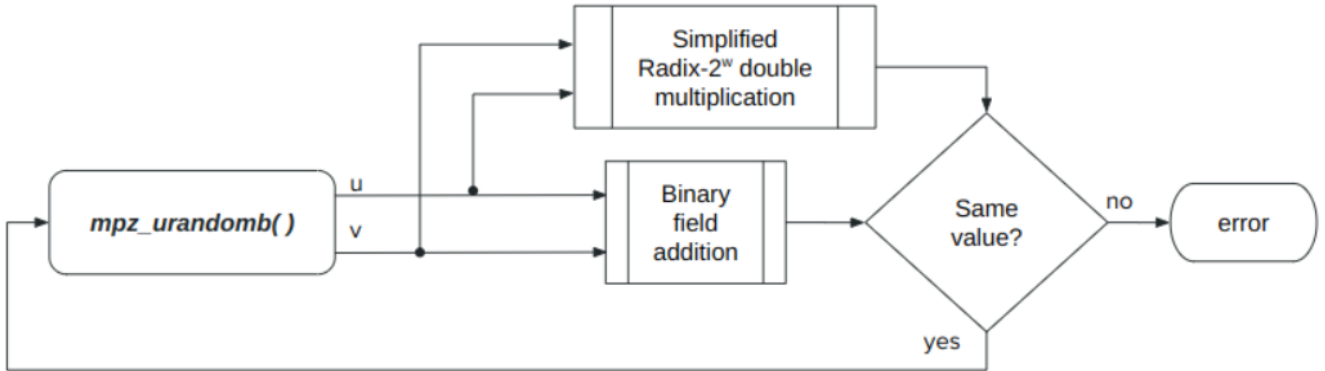


Figure 4.8: Radix- 2^w DPM recoding test bench.

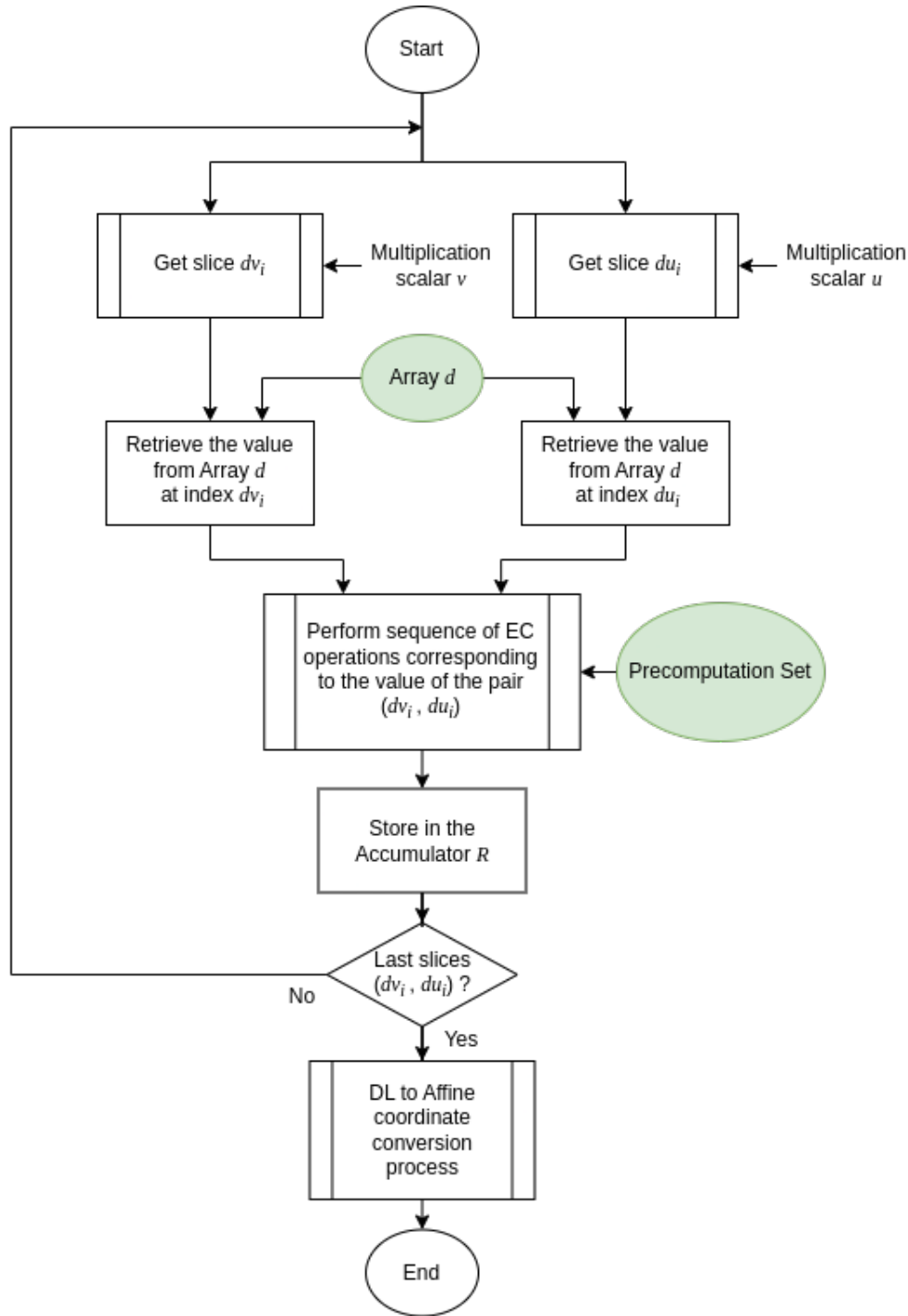


Figure 4.9: Block diagram of the Radix- 2^w EC DPM.

4.5 Elliptic Curve Multiplication Blocks Tests

4.5.1 Functionality Validation

As previously mentioned, given the nature of the EC multiplication functions, the results of these computations can not be validated by direct observation. Instead, they must be compared to the output of an already working function. These comparisons must also be performed for a large number of samples due to the size of the keys (the operand of the multiplication). It

is impossible to test for all key values, but a subsection large enough requires an automated process for testing. Therefore, finding a ready EC multiplication implementation for testing these blocks was necessary. We found a limited number of open-source libraries in different programming languages built for performing multiple precision EC arithmetic. For example, "SageMath", which is a versatile, open-source, mathematical tool. However, to the best of our knowledge, its use of binary field is limited to polynomial form. Despite writing conversion functions for field elements to and from polynomial form, it proved difficult to operate on them, and even more so, to automate the process.

The website "GF(2^m) elliptic curve calculator"[20] provides an online calculator for performing operations on binary fields and ECs (including PM). Our implementation of EC arithmetic was tested using this website. In order to test the EC PM and DPM implementations, an automated process was necessary due to the large number of samples required. A python testing program was written using *Selenium*, which is an open source tool used for browser automation. The testing process (shown in Figure 4.10) consists of two parts:

1. Generating a number of sample scalars k_i and their corresponding points $P_i(Y_i, X_i)$, where P_i is the result of the Radix 2^w EC PM of the generator point G and k_i . Both k_i and the coordinates of P_i are stored in a "result.txt" file in the order $k_1, Y_1, X_1, k_2, Y_2, X_2 \dots$ shown in Figure 4.10.
2. The "result.txt" file is used next by the testing program, described in Figure 4.11, to verify that the results of the EC PM match. This is performed by, first, retrieving k_i from the result file. Then, the multiplication $k_i \times G$ is computed on the website, the resulting coordinates are retrieved and stored in variables X_{check}, Y_{check} . These values are then compared to the ones retrieved from "result.txt". An error is produced in any case of mismatch. If the end of the file is reached without errors, an "Ok" message is delivered.

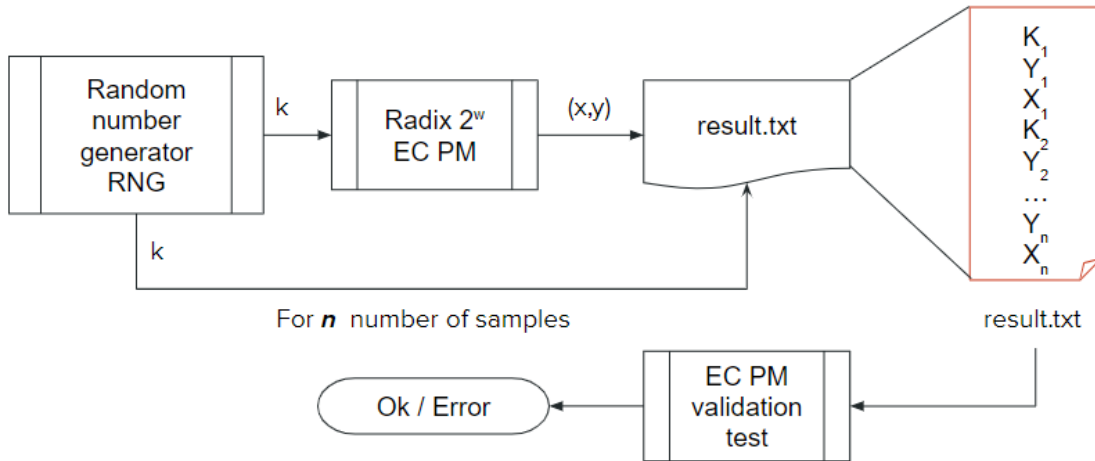


Figure 4.10: Radix- 2^w EC PM implementation validation flowchart.

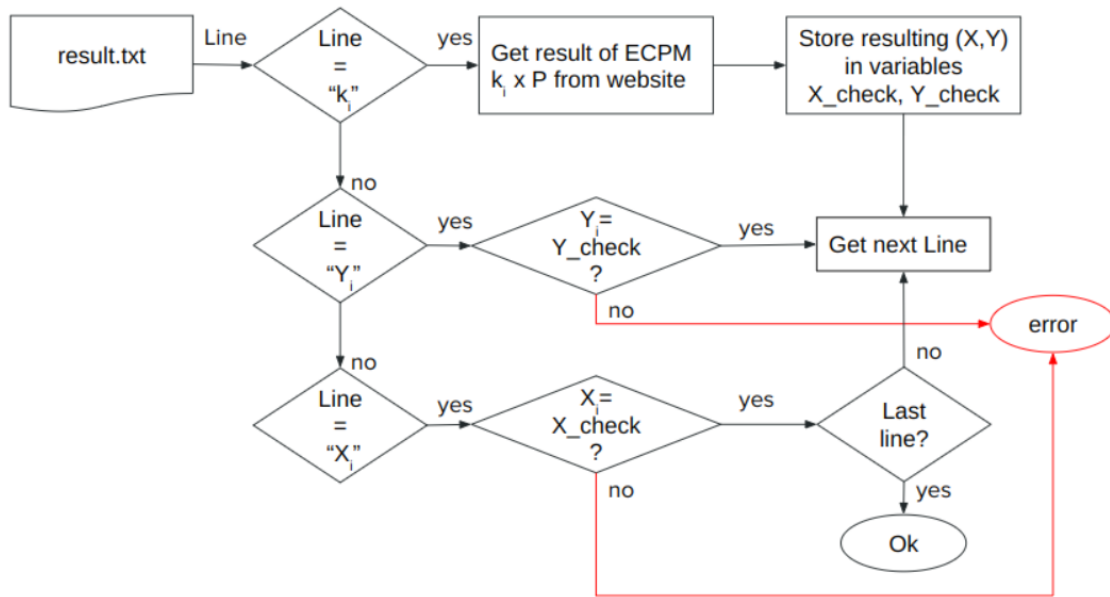


Figure 4.11: EC PM validation test flowchart.

The same procedure is followed for verifying the Radix- 2^w EC DPM implementation. The only difference being the contents of the "result.txt" file. While EC PM requires a single scalar k and a single point P (in our case the generator G), EC DPM requires two scalars u and v as well as two points P and Q . Both EC PM and EC DPM output a single point. Therefore, in the case of EC DPM both scalars are stored followed by the resulting point.

4.5.2 Elliptic Curve Point Multiplication Comparison

For comparison purposes, the average cost of binary and Radix- 2^w EC multiplication methods in terms of ADD operations was computed for a 10,000 randomly generated scalar samples, for each NIST-recommended binary curve. Figure 4.12 showcases how this comparison between the binary and the Radix- 2^w EC PM methods was conducted.

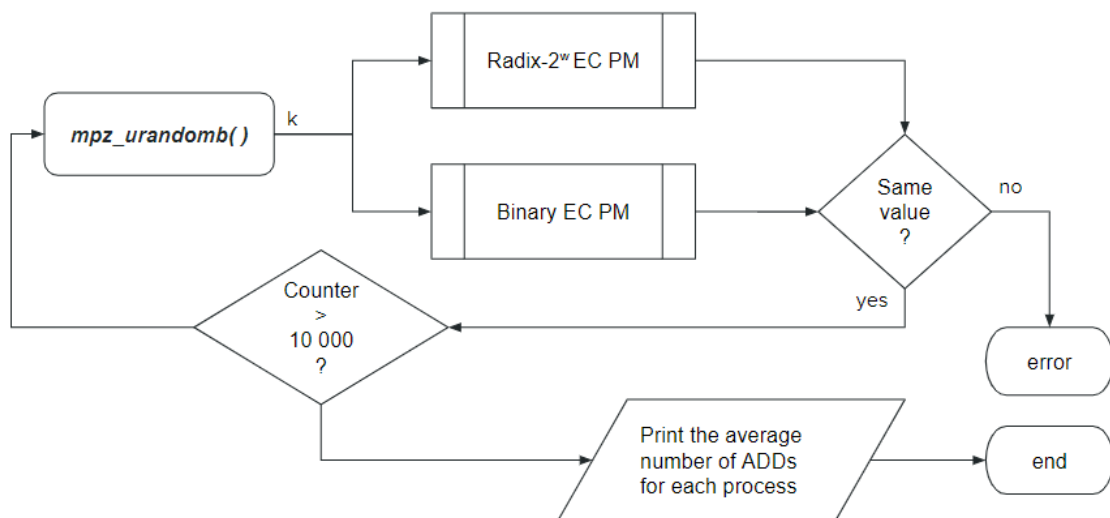


Figure 4.12: Diagram for comparison of binary and Radix 2-w methods for EC PM in terms of cost in ADDs.

Table 4.2 shows the results of the comparison carried out between the binary and the Radix- 2^w EC PM methods. The results show that the average cost of ADDs increases with the increase of the scalar size associated with each curve. This increase is natural as longer scalars are split into more slices, therefore, requiring more EC ADDs and DBLs overall. In addition to the number of ADDs performed within the multiplication process, the precomputations also take up a number of 2^{w-2} ADDs. In the case of the B-233 curve for example, the window size $w = 5$. Therefore, the required ADDs for precomputation is equal to 8. Summing the ADD cost for the precomputation and multiplication processes, the total number of additions is equal to 53.482, which is consistent with the estimations in the paper [4]. The table shows that the Radix- 2^w EC PM utilizes less than half the number of ADDs utilized within the binary EC PM. The reduction is due to the difference in density for the number representation with 0.19 for the Radix- 2^w representation and 0.5 for the binary method.

EC Multiplication Algorithm	Cost Reduction per NIST-recommended binary curve				
	B-163	B-233	B-283	B-409	B-571
Binary PM	81.143	116.485	142.311	204.703	283.395
Radix- 2^w PM	31.967	45.482	46.962	67.711	94.243
Radix- 2^w PM Total	39.967	53.482	54.962	75.711	102.243

Table 4.2: Average cost of EC PM for each multiplication method in terms of EC ADDs for each NIST-recommended binary curve.

The improvement percentage of the Radix- 2^w EC PM over the binary method was calculated using formula (4.1), and is shown in Table 4.4.

$$\text{Improvement Percentage} = 1 - \frac{\text{Total number of ADDs for (compared) PM method}}{\text{Number of ADDs for (compared to) PM method}} \quad (4.1)$$

Similarly, a comparison between Radix- 2^w EC DPM and an addition of two Radix- 2^w PMs was performed for 10,000 EC DPM operations. The results arranged in Table 4.4 show an improvement percentage averaging 47.509% in the cost of Radix- 2^w EC DPM in comparison to the binary EC PM+PM. The improvement percentage of the Radix- 2^w EC DPM over the binary method was also calculated using formula (4.1).

EC Multiplication Algorithm	Cost Reduction per NIST-recommended binary curve				
	B-163	B-233	B-283	B-409	B-571
Radix- 2^w DPM	82	117	144	207	288
Radix- 2^w DPM Total	88	123	150	213	294
Radix- 2^w PM+PM	64.887	91.918	95.02	136.372	189.521
Radix- 2^w PM+PM Total	80.887	107.918	111.02	152.372	205.521
Binary PM+PM	163.907	234.01	283.92	409.983	571.773

Table 4.3: Average cost of EC DPM for each multiplication method in terms of EC ADDs for each NIST-recommended binary curve.

However, when compared with the addition of two EC PMs, the Radix- 2^w EC DPM does not show improvement in terms of ADD cost. For example, for a scalar of bit-length 233, PM method

uses approximately 92 ADDs, while EC DPM uses 117. Taking the precomputation cost: PM requires twice the cost of a single point precomputation, making the overall ADDs about 108; EC DPM requires 6 ADDs for precomputation making the total cost of ADDs 123. The improvement percentage of the Radix- 2^w EC DPM over the addition of two Radix- 2^w EC PM for each EC was calculated according to formula (4.1), and is shown in Table 4.4. On average, Radix- 2^w EC DPM shows a decrease in cost efficiency of 28.144%. The reason the simultaneous Radix- 2^w EC DPM costs more ADDs in this case is that the window size for this algorithm was fixed to $w = 2$. In the paper [9], the simultaneous algorithm we implemented for Radix- 2^w EC DPM was not most optimized for cost. Other algorithms for Radix- 2^w EC DPM can be found in the same paper with further cost details.

Compared EC Multiplication Algorithms	Cost Reduction Percentage per NIST recommended EC				
	B-163	B-233	B-283	B-409	B-571
Radix- 2^w PM / Binary PM	50.745%	54.087%	61.379%	63.014%	63.922%
Radix- 2^w DPM / Binary PM+PM	46.311%	47.438%	47.168%	48.047%	48.581%
Radix- 2^w DPM / Radix- 2^w PM+PM	-8.794%	-13.975%	-35.110%	-39.789%	-43.051%

Table 4.4: Comparative percentage for cost of operations.

4.6 Elliptic Curve based Cryptographic Protocol Implementation

4.6.1 Public-Key Implementation

The key generation process is performed by instantiating a *KeyPair* object. The private and public keys are generated as attributes as described in Algorithm 2.1. The private key is generated using the RNG *mpz_urandomb()* (previously discussed in section 4.3.2). However, this function produces the same list of pseudo random integer values at every run. This is due to RNG algorithms using recursive methods starting at a base value. This base value depends on another integer value called a *seed*. Having a fixed value for the *seed* would result in the same list of generated values. This is not an issue for generating a testing set of scalars for arithmetic operations. However, in the case of generating a private key, this sort of predictability poses a security problem. This issue was resolved (on Linux) using *getrandom()* to randomly generate the *seed* value for *mpz_urandomb()*, subsequently improving the randomness of the multiple precision random values. The public key is generated through performing an EC PM of the private key and the generator G using the Radix- 2^w method from our implementation.

4.6.2 ElGamal Elliptic Curve Encryption Implementation

Using Elliptic Curves for Encryption and Decryption requires a unified mapping table. The purpose of this table is to assign each character, belonging to the set of characters used in the exchanged messages, to a point on the EC. Table 4.5 shows the mapping used in our implementation. The character set used includes letters A,B,...,Z and digits 0,1,...,9. In this method, each

character is encrypted independently of the rest of the message. The encryption depends solely on the keys and the character. The Algorithm 2.4 is used to encrypt each character individually.

ASCII CODE	SYMBOL	P_m
48	"0"	$1 \times G$
49	"1"	$2 \times G$
50	"2"	$3 \times G$
...
57	"9"	$10 \times G$
65	"A"	$11 \times G$
66	"B"	$12 \times G$
67	"C"	$13 \times G$
...
90	"Z"	$36 \times G$

Table 4.5: *Lookup table for encryption in ECC.*

In order to encrypt an array of characters (a string *message*), a string parsing loop was utilized. The end result of this process would be a sequence of points belonging to the EC. The encryption function stores the coordinates of each point, in sequence, into a file *"ENC_MSG.TXT"* that it creates. To reverse this process, the decryption function must first find the *"ENC_MSG.TXT"* file. Using a loop, it retrieves one point P_c at a time. Each point is utilized to obtain a character m_i , as described in Algorithm 2.5. The resulting sequence of characters are appended to a file *"DEC_MSG.TXT"* that is created by this function. At the end of this process, the original message should be found in that text file.

4.6.3 Elliptic Curve Digital Signature Implementation

The signature generation program was written as a *method*⁶ for *KeyPair*. This means that signature generation can only be performed given a key pair. With a *message* string input, the implementation of Algorithm 2.2 produces a text file *"MSG_SIG.TXT"* containing the signature (r,s). Required domain parameters are available through the *ec_parameters.h* file, as previously mentioned. The hashing function *SHA256* used was acquired from [21]. On the other hand, the signature verification program was written as a simple function. As specified in Algorithm 2.3, this function requires the public key of the other communicating party in addition to domain parameters. The public keys for both parties must be exchanged prior to using the ECDSA. The signature verification function expects two files: *"DEC_MSG.TXT"*, and *"MSG_SIG.TXT"*. These files would contain the message itself and its signature, respectively.

4.7 Joint Elliptic Curve Cryptographic Protocols Implementation

The programs for Public-Key, ECDSA, and ElGamal protocols were joined in the main function to achieve a TUI with access to both protocols individually and jointly as showcased in Figure 4.13.

⁶A procedure (function) associated with a class

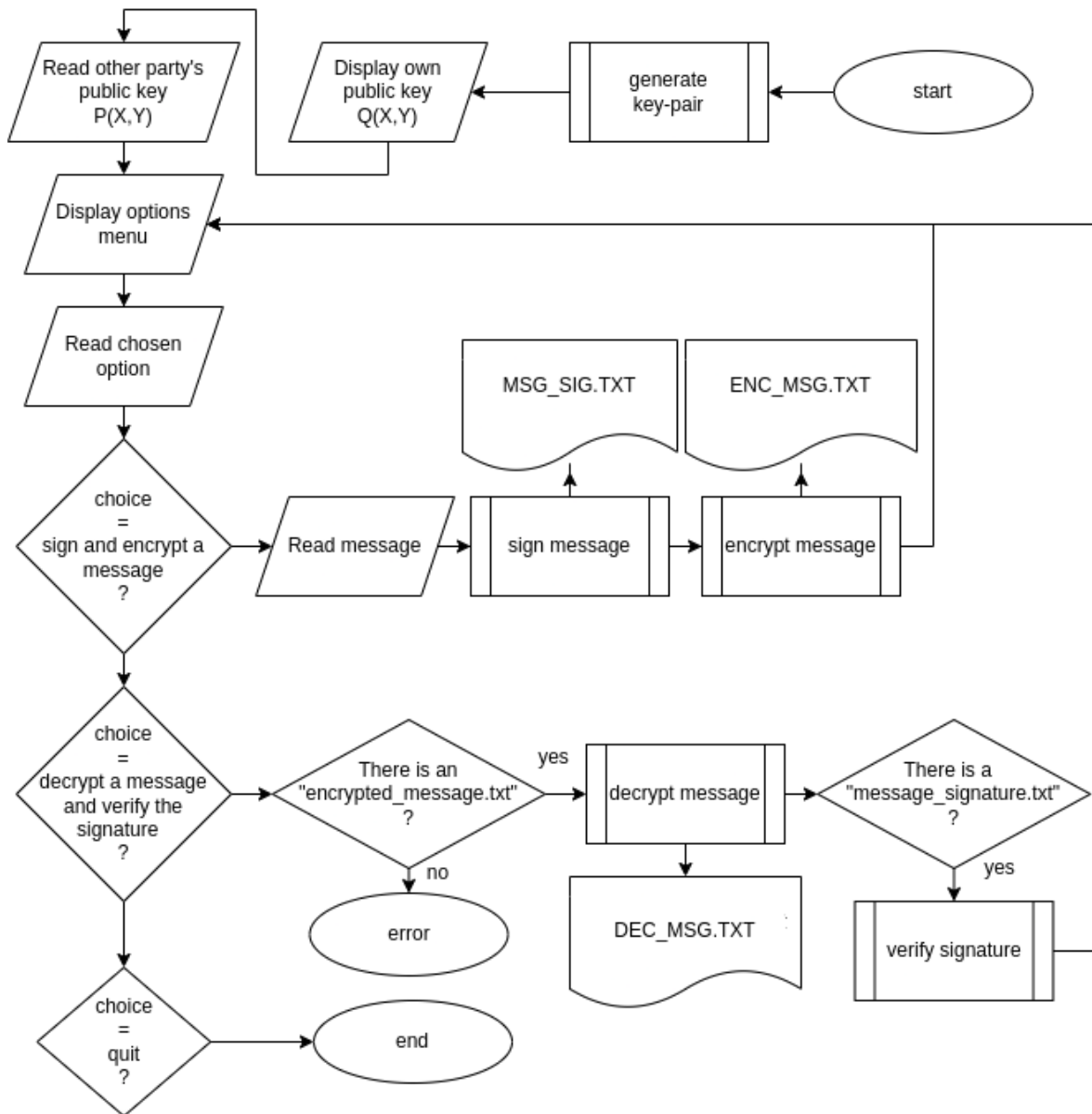


Figure 4.13: Flowchart showing the user interface implemented to access the cryptographic protocols.

Running the software implementation starts by a public key exchange. First, a key-pair is generated, then the corresponding public key is displayed. Next, the user is required to enter the other communicating party’s public key coordinates. Following the key exchange, a menu is displayed where the user is given a choice to sign and encrypt a message, or decrypt a message and verify its signature.

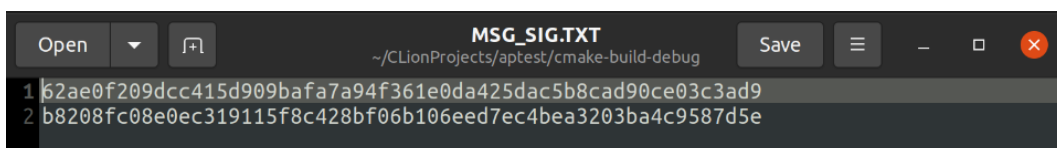
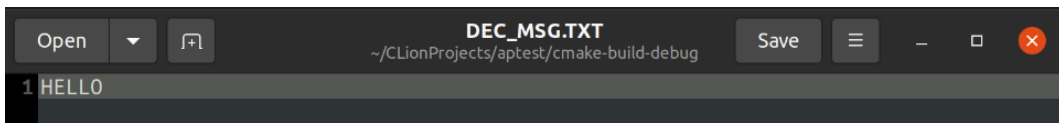


Figure 4.14: Produced signature of a message "MSG_SIG.TXT".



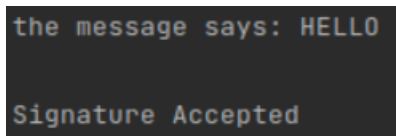
```
1 df64a10b152d535f8b2e865c2b679b777a100ff533a0658964bd46fed5
2 882fc10e460f663293d196be7833ee4848d7b78c0d194aeb7f82c969b2
3 183696716ea305905d37b34e4655b164c3abc09700192b9a28acbf8721
4 9be9a69a37e6fcc18b06cb6f6b2255df37a6ab96a1d8ac10481f79a418
5 855667ad3c460e3b9f777e2d72206829cf1cac107a6299d4e100d7aa0d
6 11d6fa1dd63d9c3769cc6851fae6269e0627b972e6f26bbbb10c43a90a8
7 855667ad3c460e3b9f777e2d72206829cf1cac107a6299d4e100d7aa0d
8 11d6fa1dd63d9c3769cc6851fae6269e0627b972e6f26bbbb10c43a90a8
9 1dcd97aa8471afbdaa75ef10b0d5f14a80a1428c66a7c60c2edfa73887d
10 c5276f21d32570edf730e1e4fcf6d178f871e92e408b85b23c25431072
```

Figure 4.15: *Produced encryption of a message "ENC_MSG.TXT".*



```
1 HELLO
```

Figure 4.16: *Produced decryption of an encrypted message "DEC_MSG.TXT".*



```
the message says: HELLO
Signature Accepted
```

Figure 4.17: *Result of the example decryption and signature verification from the TUI view of software implementation.*

Given a message "HELLO", the resulting signature is shown in Figure 4.14. This is accompanied by the encryption of the message shown in Figure 4.15. Since "HELLO" contains 5 letters, its encryption is expected to be represented by 5 EC points. This sums up to a total of 10 scalars, since each point has two coordinates (X,Y). The decrypted message file is shown in Figure 4.16 where "HELLO" was successfully retrieved. From the TUI, the decrypted message and the verification are printed as shown in Figure 4.17.

4.8 System Implementation on the ZedBoard

In order to run the ECDSA/EC-ElGamal encryption system software on the ZedBoard, a number of adjustments must first be made. In addition to serial communication, which is used for the initial key exchange, the use of this software requires reading from, and writing to, text files. In order to make the transportation of an encrypted message or signature file from the board to a different machine, using an SD card, possible.

4.8.1 File Manipulation on the ZedBoard

To manipulate files on the SD card of the board, the approach taken was to use the Xilinx File System (XilFFS) library. This library is provided by Xilinx for their embedded platforms. It is designed to provide file system support for external storage devices, such as SD cards, in embedded applications. XilFFS is based on the generic FATFS⁷ open-source library. It stands for

⁷A lightweight and portable file system implementation for small embedded systems.

File Allocation Table (FAT) File System. It supports various file system formats, including FAT12, FAT16, and FAT32, which are commonly used in SD cards and other removable storage devices. XilFFS provides a simplified application programming interface for accessing files and directories on external storage devices connected to the ZedBoard; performing common file operations such as opening, reading, writing, and closing files, as well as navigating and manipulating directories. To use the XilFFS library, it needs to be selected in the Board Support Package settings of the application project in SDK.

A few more steps are necessary and customary no matter the file system operation implemented. This includes declaring a '*FATFS*' object. This object serves as the workspace or context that allows the code to interact with the file system on the storage device, being the SD card. It is used to manage and keep track of the overall file-system, including information such as its type, free space, and various file-system parameters. A '*FIL*' object is initialized. '*FIL*' is a structure that represents an individual file within the file-system. It is used to handle access and operations specific to the file, such as opening, closing, reading, and writing. It can be thought of as a handle or reference to a specific file in the file-system. The next step is to mount the file system on the logical drive using the '*f_mount*' function. This serves to establish a connection between the file-system and the storage device on which it is located, allowing subsequent file system operations to be performed on the mounted device.

The two main operations needed in this implementation are:

1. **Reading a text file:**

SD_read is a custom function written using the FATFS library, designed to read the contents of a text file located on the ZedBoard's SD card. It takes two parameters: *FileName*, which is the name of the file to be read, and *buffer_size*, which specifies the size of the buffer in which the data read is stored. The *FileName* parameter is converted to a C-style string using the *c_str* function. This is because the *f_open* function, used to open the file as the name suggests, expects a *constchar** argument for the file name. A character array, with the specified *buffer_size*, is also declared and used to store the data read from the specified file. The latter is then opened in read mode, and a loop is executed to read the file data using the *f_read* function. This is done by repeatedly reading data from the file, using the *f_read* function, storing it in the character array, and appending it to the string type buffer. Once the reading is complete, the file is closed using the *f_close* function, and the string type buffer containing the data from the file is returned.

2. **Writing on a text file:**

SD_write is a custom function written using the FATFS library, designed to write on a text file located on the ZedBoard's SD card. It takes two parameters: *FileName*, and *constchar* data* representing the content to be written to the file. The function opens the file specified by the *FileName* parameter using *f_open(&fil, File_name, FA_CREATE_ALWAYS | FA_WRITE | FA_READ)*. The *FA_CREATE_ALWAYS* flag ensures that if the file exists, it will be overwritten. If the file doesn't exist, a new file will be created. The *FA_WRITE* and *FA_READ* flags indicate that the file should be opened for both writing and reading. The data provided by the *data* parameter is then written to the file using *f_write*. Finally, the function closes the file and displays a message indicating that the file has been written.

The integration of these two functions into the ECDSA/EC-ElGamal encryption system implementation is depicted in Figure 4.2.

General Conclusion

This project focused on the implementation of ECDSA using the Radix- 2^w method for EC PM and DPM on an ARM processor, in the context of NIST recommended binary elliptic curves. The GMP library integer data type was used to achieve multiple precision computations in our implementation of the binary field functions. The EC ADD and DBL operations were built upon our binary field functions with the EC points represented in Projective DL-coordinates. Subsequently, a binary EC PM function was written for comparison purposes, followed by the Radix- 2^w EC PM and EC DPM functions which were later utilized in our cryptographic system. Finally, the Public-key, ECDSA and EC-ElGamal protocols were implemented as independent software programs which were joint for the TUI lastly.

In order to test the functionality of the Radix- 2^w EC PM and EC DPM, automated testing was needed due to the large number of samples required. The calculator website "GF(2^m) elliptic curve calculator"[20] was used in the testing to validate the computations. Following this, the cost of the binary PM, Radix- 2^w EC PM and DPM were compared in terms of ADDs. The average number of ADDs used for each multiplication was computed for a sample of 10,000 randomly generated integers for each NIST recommended binary field elliptic curve. We found that the Radix- 2^w EC PM method is more efficient than the binary method, averaging 58.63% improvement in the cost of ADDs for the ECs B-163, B-233, B-283, B-409, and B-571. The number of ADD operations incurred during the Radix- 2^w EC PM process is consistent with estimations in the paper [4]. As for the Radix- 2^w EC DPM, it was compared to the cost of the ADD of two EC PM multiplications. When compared to the ADD of two binary EC PM multiplications, it proved to be 58.63% more cost efficient than the binary (averaging for the same sequence of ECs). However, when compared to the the ADD of two Radix- 2^w EC PM multiplications, the EC DPM was found to be 28.14% less cost efficient. Meaning, the Radix- 2^w EC DPM costs more than the sum of two Radix- 2^w EC PMs. These results are due to the Radix- 2^w EC DPM method having a fixed $w = 2$ that is not most optimized for cost, which is consistent of the findings in paper [9]. The results of said testing show that both the Radix- 2^w EC PM and DPM methods demonstrate the expected computational efficiency compared to the binary method. The elliptic curve joint cryptographic protocol was then implemented for both PCs and the zedboard. The signatures and encrypted messages are successfully produced and processed by both types of machines: using the TUI for PCs, and serial communication for the zedboard. The resulting files containing the signature and the encrypted message are transferred between communicating devices manually, through the SD card, and placed appropriately for reading and writing.

This project successfully achieved its primary objective, however, it would benefit from a number of improvements. First, the binary field squaring was implemented as a simple multiplication in our squaring function. This can be improved by implementing existing optimization algorithms for this function. Second, this system exclusively runs on the NIST recommended binary elliptic curves. The implementation of prime field operations along with the inclusion of the parameters

of their corresponding curves would enhance the versatility and applicability of the system. Third, the use of a multiple precision library to ensure the integrity of the large values of the scalars leaves the memory usage ambiguous. Hence, it could pose a problem in memory constrained devices. The issue could be improved by using a different method for achieving the desired precision in the computations (i.e., array representation for large scalars). In addition, unlike the signature file containing exactly 2 values independently of the message, the encrypted message file's size is not fixed thus may posing problems concerning memory and speed with memory constraint devices. Lastly, since our implementation deals with serial communication and text files, it would benefit from a more efficient method for data transfer. This could be performed through the integration of TCP-IP protocols for example. These future improvement suggestions would enable broader and more efficient applications for the system.

Bibliography

- [1] I. Costan, “Elliptic curves.” [Online]. Available: <https://blog.costan.ro/post/2019-09-25-elliptic-curves/>
- [2] M. Muhlberghuber, “Comparing ecdsa hardware implementations based on binary and prime fields,” Inffeldgasse, Graz, Austria, June 2011.
- [3] D. Hankerson, A. J. Menezes, and S. Vanstone., *Guide to elliptic curve cryptography*. Springer-Verlag New York, Inc, 2004.
- [4] A. K. Oudjida and A. Liacha, “Radix-2w arithmetic for scalar multiplication in elliptic curve cryptography,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 5, pp. 1979–1989, 2021.
- [5] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *ZedBoard System Architecture*. Strathclyde Academic Media, 2014, p. 135.
- [6] L. Chen, D. Moody, K. Randall, A. Regenscheid, and A. Robinson, “Recommendations for discrete logarithm-based cryptography: Elliptic curve domain parameters,” *NIST Special Publication, NIST SP 800-186*, 2023. [Online]. Available: <https://doi.org/10.6028/NIST.SP.800-186>
- [7] V. Dimitrov and K. Järvinen, “Another look at inversions over binary fields,” in *2013 IEEE 21st Symposium on Computer Arithmetic*, 2013, pp. 211–218.
- [8] S. Bartolini, I. Branovic, R. Giorgi, and E. Martinelli, “Effects of instruction-set extensions on an embedded processor: A case study on elliptic-curve cryptography over $gf(2^m)$,” *IEEE Transactions on computers*, vol. VOL.57, no. NO.5, 2008.
- [9] A. K. Oudjida, A. Liacha, F. Nait-Abdesselam, and A. Khouas, “Simple and efficient algorithms for double point multiplication in elliptic curve cryptography,” *2023 forthcoming*, 2021.
- [10] “Digital signature standard (dss),” National Institute of Standards and Technology (NIST), Tech. Rep. Federal Information Processing Standard (FIPS) 186-5, 2013.
- [11] E. Barker, “Recommendation for key management: Part 1 – general,” National Institute of Standards and Technology (NIST), Special Publication 800-57 Part 1 Revision 5, May 2020.
- [12] S. Levy, “Performance and security of ecdsa,” 2015. [Online]. Available: <https://api.semanticscholar.org/CorpusID:12722775>
- [13] J. Adikari, V. Dimitrov, and P. Mishra, “Fast multiple point multiplication on elliptic curves over prime and binary fields using the double-base number system.” *IACR Cryptology ePrint Archive*, vol. 2008, p. 145, 01 2008.

- [14] R. Azarderakhsh and K. Karabina, “A new double point multiplication algorithm and its application to binary elliptic curves with endomorphisms,” *Computers, IEEE Transactions on*, vol. 63, pp. 2614–2619, 10 2014.
- [15] J. C. López-Hernández and R. Dahab, “Improved algorithms for elliptic curve arithmetic in $gf(2^n)$,” in *ACM Symposium on Applied Computing*, 1998.
- [16] Q. H. Dang, “Secure hash standard (shs),” *Federal Inf. Process. Stds.(NIST FIPS)*, 2012. [Online]. Available: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>
- [17] “Introduction to gmp (gnu mp 6.2.1).” [Online]. Available: <https://gmplib.org/manual/Introduction-to-GMP>
- [18] “Build options (gnu mp 6.2.1).” [Online]. Available: <https://gmplib.org/manual/Build-Options>
- [19] “Random state initialization (gnu mp 6.2.1).” [Online]. Available: <https://gmplib.org/manual/Random-State-Initialization>
- [20] L. Leinweber, “Gf(2m) elliptic curve calculator.” [Online]. Available: http://www.leinweb.com/case/gfeccalc.html#storage_ops
- [21] System-Glitch, “System-glitch/sha256: A c++ sha256 implementation.” [Online]. Available: <https://github.com/System-Glitch/SHA256>