University of Palermo

PhD joint program:
University of Catania - University of Messina
XXXV cycle

Doctoral Thesis

Geometric methods in coding theory and cryptography

Author:
Giuseppe Filippone

Supervisor:
Prof. Domenico Tegolo

Co-Supervisor:
Prof. Giovanni Falcone

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in

Mathematics and Computational Sciences

May 29, 2023

Declaration of Authorship

I, Giuseppe Filippone, declare that this thesis titled, "Geometric methods in coding theory and cryptography" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Giuseppe Filippone

Date: May 29, 2023

Contents

Declaration of Authorship i

Contents ii

List of Figures v

List of Tables vi

List of Symbols vii

Introduction 3

I Mathematical background 6

1 Cryptography 7

2 Algebraic curves 12
- 2.1 Projective space . . . 12
- 2.2 Homogeneous rational functions . . . 14
- 2.3 Algebraic variety . . . 16
- 2.4 Divisor of algebraic curves . . . 18

3 Elliptic curves 20
- 3.1 Elliptic curves in Weierstrass form . . . 20
  - 3.1.1 Group law . . . 22
  - 3.1.2 Elliptic curves in Weierstrass form over the field of complex numbers . . . 24
  - 3.1.3 Elliptic curves in Weierstrass form over the field of rational numbers . . . 25
  - 3.1.4 Elliptic curves in Weierstrass form over finite fields . . . 26
- 3.2 Montgomery curves . . . 27
- 3.3 Edwards curves . . . 27
  - 3.3.1 Group law . . . 28
  - 3.3.2 Equivalence between Edwards curves and Elliptic curves in Weierstrass form . . . 31
  - 3.3.3 Twisted Edwards curves . . . 34

4 Hyperelliptic curves 36
- 4.1 Group law . . . 37
- 4.2 Meaning of the Cantor-Koblitz algorithm . . . 39
- 4.3 Doubling of divisors without Cantor-Koblitz algorithm . . . 43

5 Elliptic curves cryptography **45**
5.1 Diffie-Hellman and ElGamal 46

6 Code-based cryptography **48**
6.1 Coding theory 48
6.2 McEliece cryptosystem 50
6.3 Algebraic-geometric Goppa codes 51

II Curves over $\mathbb{Q}_p$ **53**

7 Local fields **54**
7.1 The p-adic numbers 55

8 Elliptic curves over local rings **57**
8.1 The case $\mathbb{K} = \mathbb{Q}_p$ 58
8.2 Group law over $\mathbb{Q}_p$ 60
8.3 The Log function 61
8.4 ECDLP exploiting the map $\text{Log}_{\overline{\mathcal{W}}}$ 62
8.4.1 Refinement of the ECDLP over $\mathbb{Q}_p$ 63
8.5 ECDLP from $\mathbb{Q}$ to $\mathbb{Q}_p$ 65

9 Hyperelliptic curves over $\mathbb{Q}_p$ **68**
9.1 "Weaknesses" of Cantor-Koblitz algorithm 70

10 Edwards curves over $\mathbb{Q}_p$ **73**
10.1 The map Exp over local fields 73
10.2 The map Exp over $\mathbb{Q}_p$ 75

III Coding theory **77**

11 Goppa codes for Edwards curves **78**
11.1 The Riemann-Roch space $\mathcal{L}(D)$ 78
11.1.1 Computational cost 81
11.2 AG Goppa codes 82
11.2.1 Computational cost of constructing a Goppa code over Edwards curves 83
11.3 A small example 83
11.4 A small example of a McEliece cryptosystems 84
11.5 Implementation 85

12 Goppa codes for hyperelliptic curves **86**
12.1 The Riemann-Roch space $\mathcal{L}(D)$ 86
12.2 AG Goppa codes 91
12.3 A small example 92

13 HL-codes and their decoding algorithm **94**
13.1 Reed-Muller codes 94
13.2 HL-codes 99
13.3 Decoding 100

Conclusions **108**

IV Appendices and index 110

A Pseudocodes 111

A.1 ECDLP using the Log function 111
A.2 AG Goppa codes for Edwards curves 115
A.3 AG Goppa codes for hyperelliptic curves 117
A.4 HL-codes 119

B Implementations and benchmarks 129

B.1 ECDLP using the Log function 129
B.2 AG Goppa codes for Edwards curves 134
B.3 AG Goppa codes for hyperelliptic curves 135
B.4 HL-codes 137

Bibliography 147

Index 153

List of Figures

3.1 Sum of two points in an elliptic curve 24
3.2 Sum of two points on an Edwards curve 29

4.1 Sum of two divisors in a hyperelliptic curve of genus 2 39
4.2 Representation of divisors by means of the pair (parabola, straight line) 40
4.3 The only cubic (in light blue) passing through the points belonging to the supports of the two divisors $D_1 = P_1 + P_2 - 2 \cdot \Omega$ (in green) and $D_2 = Q_1 + Q_2 - 2 \cdot \Omega$ (in red) 42
4.4 The only cubic (in light blue) passing through the four points P_1, P_2, Q_1, and Q_2, where P_1 and P_2 are in green, while Q_1 and Q_2 are in red 42
4.5 Tangent cubics with Taylor at points P and Q 44
4.6 Tangent cubics at points P and Q 44

11.1 Edwards real curve with $d = -8$. The points P, R, and R' have the following coordinates (a, b), $(a, -b)$, and $(-a, -b)$, respectively 80

B.1 Benchmark for $p = 3$ and $p = 37$ 130
B.2 Benchmark for $p = 3$, $p = 5$ and $p = 17$ 130
B.3 Benchmark of the ECDLP for the elliptic curve given by the equation $y^2 = x^3 - x + \frac{1}{4}$ over $\mathbb{Z}/p^k\mathbb{Z}$ 130
B.4 Encryption times for an HL-code of parameter $m = 10$ 138
B.5 Decryption times for an HL-code of parameter $m = 10$ 138
B.6 WALL-time for an HL-code of parameter $m = 10$ 138
B.7 Encryption times for an HL-code of parameter $m = 12$ 138
B.8 Decryption times for an HL-code of parameter $m = 12$ 138
B.9 WALL-time for an HL-code of parameter $m = 12$ 138

List of Tables

8.1 ECDLP as successive approximations modulo 3^k 65

B.1 Hardware 129
B.2 Mean time for the protocol 139

List of Symbols

$\mathbb{K}$	is a generic field
$\mathbb{C}$	is the field of complex numbers
$\mathbb{R}$	is the field of real numbers
$\mathbb{Q}$	is the field of rational numbers
$\mathbb{Z}$	is the field of integer numbers
$\mathbb{N}$	is the field of natural numbers
$\mathbb{Q}_p$	is the field of p-adic numbers
$\mathbb{Z}_p$	is the field of p-adic integers
$\mathrm{GF}(p)$	is the Galois finite field of integers modulo p
$\mathbb{Z}/p^k\mathbb{Z}$	is the ring of integers modulo p^k
$\mathrm{char}(\mathbb{K})$	is the characteristic of the field $\mathbb{K}$
$\mathrm{card}(\mathbb{K})$	is the cardinality (or the order) of the field $\mathbb{K}$
$a \pmod p$	is the modulo p operation of the integer a
$\mathbf{x}$	is a vector x of n entries
A	is a $m \times n$ matrix A
A_n	is a $n \times n$ matrix A
$\mathbf{P}$	is the projective space
$\mathbf{P}^n(\mathbb{K})$	is the projective space $\mathbf{P}$ over a vector space $\mathbb{K}^n$
${}^h f$	is the homogenization of the function f
$\mathbf{P}^1(\mathbb{K})$	is the projective line
$\mathbf{P}^2(\mathbb{K})$	is the projective plane
$[Z : X : Y]$	are the homogeneous coordinates of a point belonging to a projective curve
$\mathrm{Exp}(z)$	is the exponential map that returns the point P from $\wp(z)$ and $\wp'(z)$
$\mathrm{Exp}_{\mathcal{C}}(z)$	is the exponential map with respect to the curve $\mathcal{C}$
$\mathrm{Mod}(P)$	is the modulo map that returns the point P which coordinates are reduced modulo p
$\mathrm{Mod}_{\mathcal{C}}(P)$	is the modulo map with respect to the curve $\mathcal{C}$
$\mathrm{Mod}_h(P)$	is the modulo map that returns the point P which coordinates are reduced modulo p^h
$\mathrm{Log}(z)$	is the inverse function of $\mathrm{Exp}(z)$
$\mathrm{Log}_{\mathcal{C}}(z)$	is the inverse function of $\mathrm{Exp}_{\mathcal{C}}(z)$
λP	is the scalar multiplication by $\lambda \in \mathbb{K}$ of each coordinate of the (affine or projective) point P, that is, $\lambda P = [\lambda x_0 : \lambda x_1 : \ldots : \lambda x_n]$ or $\lambda P = (\lambda x, \lambda y)$

$\mathrm{E}(m,k)$	is an encryption algorithm for any message $m \in \mathbb{D}^n$ and key $k \in \mathbb{D}^l$
$\mathrm{D}(c,k)$	is a decryption algorithm for any encrypted message $c \in \mathbb{D}^n$ and key $k \in \mathbb{D}^l$
$\mathbb{D}$	is an arbitrary domain used in both the two maps E and D
$\mathcal{C}(\mathbb{K})$	is an arbitrary algebraic curve $\mathcal{C}$ over a field $\mathbb{K}$
$\mathcal{C}_k$	is $\mathcal{C}\left(\mathbb{Z}/p^k\mathbb{Z}\right)$
$\mathcal{W}(\mathbb{K})$	is an arbitrary generalized elliptic curve in Weierstrass form $\mathcal{W}$ over a field $\mathbb{K}$
$\overline{\mathcal{W}}(\mathbb{K})$	is an arbitrary short elliptic curve in Weierstrass form $\overline{\mathcal{W}}$ over a field $\mathbb{K}$
$\mathcal{E}(\mathbb{K})$	is an arbitrary Edwards curve $\mathcal{E}$ over a field $\mathbb{K}$
$\mathcal{H}(\mathbb{K})$	is an arbitrary Hyperelliptic curve $\mathcal{H}$ over a field $\mathbb{K}$
$\mathcal{M}(\mathbb{K})$	is an arbitrary Montgomery curve $\mathcal{M}$ over a field $\mathbb{K}$
$\mathrm{Div}(\mathcal{C})$	is the group of divisor of an arbitrary algebraic curve $\mathcal{C}$
$\mathrm{Div}^0(\mathcal{C})$	is the group of zero degree divisor of an arbitrary algebraic curve $\mathcal{C}$
$\mathrm{Princ}(\mathcal{C})$	is the principal divisor class of an arbitrary algebraic curve $\mathcal{C}$
$\mathcal{J}(\mathcal{C})$	is the Jacobian $\mathrm{Div}^0(\mathcal{C})/\mathrm{Princ}(\mathcal{C})$ of an arbitrary algebraic curve $\mathcal{C}$ over a field $\mathbb{K}$
$\mathcal{J}_k(\mathcal{C})$	is the Jacobian of an arbitrary algebraic curve $\mathcal{C}$ over $\mathbb{Z}/p^k\mathbb{Z}$, for some p
$\mathrm{P}(A)$	is the probability that an event A occurs
$n \cdot P$	is the formal multiplication between a scalar $n \in \mathbb{Z}$ and a point P in a divisor
$P - Q$	is the formal subtraction between two points in a divisor
$P + Q$	is the formal addition between two points in a divisor
$P \oplus Q$	is the group addition between the two divisors $P - O$ and $Q - O$ belonging to the Jacobian of an algebraic curve, where O is the identity element
$\ominus P$	is the opposite element of P according to the group operation of the Jacobian of an algebraic curve
$P \ominus Q$	is the group subtraction between the two divisors $P - O$ and $Q - O$ belonging to the Jacobian of an algebraic curve, where O is the identity element. This is the abbreviation of $P \oplus (\ominus Q)$
$n \otimes P$	is the group addition of $P - O$ with itself $n \in \mathbb{Z}$ times, that is, $\bigoplus_{i=1}^{n} P$ if $n > 0$, while for $n < 0$ one takes $\bigoplus_{i=1}^{-n} \ominus P$
$D_1 \oplus D_2$	is the group addition between two principal divisors of the Jacobian of a hyperelliptic curve
$\ominus D$	is the opposite element of the divisor D according to the group operation of the Jacobian of a hyperelliptic curve
$D_1 \ominus D_2$	is the group subtraction between two principal divisors of the Jacobian of a hyperelliptic curve, that is, $D_1 \oplus (\ominus D_2)$
$n \otimes D$	is the group addition of the divisor D with itself $n \in \mathbb{Z}$ times, that is, $\bigoplus_{i=1}^{n} D$ if $n > 0$, where for $n < 0$ one takes $\bigoplus_{i=1}^{-n} \ominus D$
$\mathcal{L}(D)$	is the Riemann-Roch space of a divisor D belonging to the Jacobian of a curve

Structure of thesis

The structure of this thesis is as follows:

- In Introduction, we provide a brief overview of cryptography and its importance;
- In chapter 1, we present the mathematical foundations of different types of cryptography and the concept of security;
- In chapter 2, we introduce the topics of curves and algebraic varieties;
- In chapter 3, our focus is on elliptic curves, including their group law, and we present some key results regarding their properties based on their ground field;
- In chapter 4, we delve into the topic of hyperelliptic curves, including their group law, and specifically focus on the Cantor-Koblitz algorithm used to sum two divisors belonging to the Jacobian of a hyperelliptic curve;
- In chapter 5, we provide a brief introduction to cryptography using elliptic curves, with a particular focus on the importance of selecting "good" elliptic curves that ensure an acceptable level of security. Specifically, we discuss concerns related to the properties of elliptic curves, such as the need to avoid "anomalous" curves;
- In chapter 6, we introduce the fundamental concepts of coding theory and discuss its relevance within the context of cryptography, i.e. the code-based cryptography;
- In chapter 7, we provide an introduction to local fields, their structure, and their properties;
- In chapter 8, we examine the structure of the Jacobians of elliptic curves over local fields. In particular, we focus on the field of p-adic numbers. Moreover, we provide an algorithm to compute the inverse of the exponential map defined on elliptic curves in Weierstrass form in order to compute the elliptic curve discrete logarithm;
- In chapter 9, we extend the results of chapter 8 to hyperelliptic curves;
- In chapter 10, we analyze the Jacobian structure of Edwards curves over the field of p-adic numbers.
- In chapter 11, we give a basis of the Riemann-Roch space for Edwards curves over finite fields, and we use the latter basis to compute a generating matrix for an algebraic-geometric Goppa code for such curves;
- In chapter 12, we give a basis of the Riemann-Roch space for hyperelliptic curves over finite fields using the Mumford representation of the divisors of their Jacobian, and we use the latter basis to computer a generating matrix for an algebraic-geometric Goppa code for such curves;

- In chapter 13, we describe HL-codes as sub-classes of Reed-Muller codes (RM codes), and we extend the original decoding algorithm for RM codes to HL-codes;
- In Conclusions, we resume the results of this thesis.

At the end of this thesis, readers will find a list of symbols used throughout the document, as well as a comprehensive index to facilitate navigation and understanding of the topics discussed.

Introduction

Cryptography, or the study of techniques for secure communication in the presence of third parties, has a long and fascinating history that spans thousands of years. Its origins can be traced back to ancient civilizations, where it was used for military and diplomatic purposes. In this chapter, we provide a brief overview of the history of cryptography, highlighting some of the key developments and milestones that have shaped the field.

One of the earliest recorded uses of cryptography was the scytale, a rod used by the ancient Greeks to send secret messages by wrapping a strip of parchment around it and writing on it. The message could only be read by someone who had a rod of the same diameter as the one used to write the message. This technique was famously used by the Spartan military to communicate battlefield orders.

In the 20th century, the development of sophisticated mechanical devices such as the Enigma machine allowed for the secure transmission of messages. The Enigma machine, introduced by the Germans in the 1920s, used a series of rotating disks to encode messages, which were then transmitted via radio or telegraph. The machine was later used by the Germans during World War II to encode military communications.

With the advent of computers in the 20th century, the field of cryptography underwent a significant transformation. For instance, in 1978, Rivest, Shamir, and Adleman introduced the RSA algorithm [65], which is widely used today for secure communication. The RSA algorithm is based on the difficulty of factoring large composite numbers, and it is used for a variety of applications, including online communication and secure storage of sensitive data.

In the 1990s, the National Institute of Standards and Technology (NIST) launched a competition to select a new standard for symmetric-key encryption, which is a type of encryption that uses the same key for both encryption and decryption. In 2001, NIST selected the Advanced Encryption Standard (AES) as the new standard [26]. The AES is a widely used algorithm that is considered to be secure against attacks by classical computers.

With the growing concern about the potential vulnerabilities of current cryptographic techniques in the face of quantum computers, there has been a renewed interest in developing the so-called post-quantum cryptography [10]. Quantum computers, which use quantum-mechanical phenomena to perform calculations, are believed to be able to break many of the encryption schemes that are currently used. Unlike the classical ones, quantum computers exploit the qubits (quantum bits) to overcome the limits of classical technology based on the binary system. The qubits, in fact, carry considerably more information than the classic 64-bit binary registers. The greater the number of qubits employed, the greater the threat to modern cryptographic algorithms. Specifically, while classical computers use bits that can only assume two values (0 or 1), quantum computers take advantage of quantum superposition, allowing them to perform calculations simultaneously on multiple quantum states. This

enables quantum computers to solve some computational problems in much shorter times compared to classical computers, especially for problems that involve factorizing large numbers, exhaustive searching or computing the inverse of a function (such as with the Grover's algorithm). For instance, the well-known Shor's algorithm [71] for integer factorization (in quantum computers) is an enormous threat for the RSA algorithm. Fortunately, at the present time, there are no quantum computers with enough qubits to undermine neither RSA nor modern cryptography. However, in order to now-days security, researchers are working to develop cryptographic techniques that are resistant to attacks by quantum computers.

As already mentioned, the history of cryptography is closely tied to the development of various encryption techniques. In the early days of cryptography, mono-alphabetic ciphers were commonly used. These ciphers used a fixed substitution alphabet, where each letter in the plaintext (the message to encrypt) was replaced with a corresponding letter in the ciphertext (the encrypted message). One of the most famous mono-alphabetic ciphers is the Caesar cipher, which was used by Julius Caesar to send secret messages to his generals.

As the use of mono-alphabetic ciphers became more widespread, it became increasingly easy for attackers to break them using various statistical analysis techniques, which consist of checking how often letters appear in an encrypted message. More precisely, statistical analyses of encrypted messages have shown a correspondence between the most frequent letters in ciphered texts using mono-alphabetic ciphers and the most frequent letters in the language in which the unencrypted message was written (for instance, in English the two most frequent letters are "a" and "e", while the least frequent letters are "x", "j", "q" and "z"). Thus, it follows that mono-alphabetic ciphers are absolutely insecure because one can easily decipher them. In response, poly-alphabetic ciphers were developed, which used multiple substitution alphabets in an attempt to thwart statistical attacks. One of the most famous poly-alphabetic ciphers is the Vigenère cipher, which was developed in the 16th century and remained unbroken for over 300 years.

Despite their improved security, poly-alphabetic ciphers were eventually broken using sophisticated cryptanalytic techniques. In 1917, Gilbert Vernam, an engineer at the American Telephone and Telegraph Company (AT&T), developed the Vernam cipher, which is considered to be the first practical example of a perfect secrecy cipher. The fundamental concept behind the Vernam cipher is the one-time pad (OTP), which is a random secret key that is used for both encrypting and decrypting a message. The security of the Vernam cipher is based on the fact that the key is at least as long as the message and is used only once. More precisely, this cipher is immune to the ciphertext-only attack (or COA), that is, a ciphertext attack where a malicious user tries to retrieve the unencrypted message using only the encrypted one. More precisely, due to the randomness of the OTP, it is impossible for an attacker to use statistical analysis or other methods to decipher the message without access to the key.

The development of the Vernam cipher marked an important milestone in the history of cryptography, as it demonstrated that perfect secrecy (meaning that the ciphertext reveals no information about the plaintext message, even if the attacker has infinite computational power) was theoretically possible. Today, the Vernam cipher is still considered to be one of the most secure encryption techniques, although it is rarely used in practice due to the difficulty of securely transmitting, storing and generating

such random secret key. Specifically, it is impossible to use the original Vernam cipher since it requires a truly random OTP, which is not feasible to generate. Nonetheless, the concept of the Vernam cipher has been adapted and repurposed with the use of pseudo-random number generators (PRNGs). These generators produce a pseudo-random key at least as long as the plaintext, starting from a seed value. While this approach emulates the Vernam cipher's idea, unlike a truly random OTP, it is susceptible to several issues concerning the vulnerability of PRNGs in how they generate their sequence of "random" numbers.

Overall, the field of cryptography has played a crucial role in the evolution of secure communications, blockchains, smart contracts, crypto-currencies, and will continue to be an important area of research in the future. As technology and mathematics continue to advance, we can expect to see new developments and innovations in the field of cryptography.

In general, cryptography exploits mathematical problems that are inherently difficult to solve for anyone without the private information, that is, for any user not authenticated. Among these mathematical problems, we find the discrete logarithm problem (such as with cryptography using elliptic curves), the factorization of integers (such as with RSA), the decoding of a linear code (such as with the McEliece cryptosystem), finding the inverse of a function (such as with 3DES, AES, and other symmetric ciphers).

In this thesis, we will mainly deal with cryptographic algorithms based on algebraic curves, a possible attack that aims to speed up the current algorithms in solving the discrete logarithm problem for algebraic curves, and code-based cryptography using HL-codes.

The results obtained in chapter 10 have been published in [31]. Moreover, the results obtained in chapter 8, chapter 11, chapter 12, and chapter 13 have been collected in four articles submitted for possible publication. We address the reader to the corresponding preprints in [29, 32, 33, 37].

Part I

Mathematical background

1 Cryptography

In this chapter, we provide an introduction to the field of cryptography, including key definitions and modern standards.

Cryptography involves the use of encryption algorithms to convert plaintext, or unencrypted information, into ciphertext, or encrypted information and vice versa.

Cryptography algorithms can be divided into two main categories: symmetric-key and asymmetric-key encryption. The former uses the same key, known as shared key, to encrypt the plaintext and decrypt the ciphertext, while the latter uses one key, known as public-key, to encrypt the plaintext and another key, known as private-key, to decrypt the ciphertext.

Remark 1.1. *It is not strictly necessary for the key used in encryption to be as long as the plaintext, as secure key expansion protocols can be implemented, depending on the cryptographic protocol being used.*

In order to distinguish between symmetric-key and asymmetric-key encryption algorithms, it is first necessary to understand the difference between encryption and decryption methods.

As outlined in definition 1.1, an encryption method is a function that maps a pair (plaintext, key) to the corresponding ciphertext. Conversely, as defined in definition 1.2, a decryption method maps a pair (ciphertext, key) to the corresponding plaintext. It is noteworthy that these mappings may not be invertible everywhere. However, this property allows for the definition of secure methods, as outlined in definition 1.3 and definition 1.4.

Definition 1.1 (Encryption method)**.** *Let* $\mathrm{E}(m,k)$ *be an encryption method defined as follows:*

$$\begin{aligned} \mathrm{E} : \mathbb{D}^n \times \mathbb{D}^l &\longrightarrow \mathbb{D}^n \\ (m,k) &\longmapsto c \end{aligned} \tag{1.1}$$

where $l \leq n \in \mathbb{N}$*,* $\mathbb{D}$ *is a suitable domain such as a finite field,* k *is a key,* m *is a plaintext, and* c *is the corresponding ciphertext.*

Definition 1.2 (Decryption method)**.** *Let* $\mathrm{D}(c,k)$ *be a decryption method defined as follows:*

$$\begin{aligned} \mathrm{D} : \mathbb{D}^n \times \mathbb{D}^l &\longrightarrow \mathbb{D}^n \\ (c,k) &\longmapsto m \end{aligned} \tag{1.2}$$

where $l \leq n \in \mathbb{N}$*,* $\mathbb{D}$ *is a suitable domain such as a finite field,* k *is a key,* c *is a ciphertext, and* m *is the corresponding plaintext.*

In order to ensure the safety of encrypting and decrypting, it is crucial that both the encryption and decryption methods are secure. Specifically, this means that the probability that the encryption of two distinct plaintexts results in the same ciphertext, or the decryption of two distinct ciphertexts results in the same plaintext,

is negligible. In other words, the encryption and decryption methods should be invertible almost everywhere within their domain.

Definition 1.3 (Secure encryption method)**.** *Let* $\mathrm{E}(m, k)$ *be an encryption method, then the method E is secure if, for any pair of plaintexts* (m_1, m_2) *such that* $m_1 \neq m_2$*, the probability* $\mathrm{P}(\mathrm{E}(m_1, k) = \mathrm{E}(m_2, k))$ *is negligible, where* k *is a cryptographically secure key.*

Definition 1.4 (Secure decryption method)**.** *Let* $\mathrm{D}(c, k)$ *be a decryption method, then the method D is secure if, for any pair of ciphertexts* (c_1, c_2) *such that* $c_1 \neq c_2$*, the probability* $\mathrm{P}(\mathrm{D}(c_1, k) = \mathrm{D}(c_2, k))$ *is negligible, where* k *is a cryptographically secure key.*

Remark 1.2. *It is not straightforward to define a cryptographically secure key, as the strength of a key is dependent on various parameters such as the size of the key, typically measured in terms of number of bits, or the randomness of the key, which is influenced by the method used to generate the key. Typically, a key is considered secure if it passes various security tests.*

As previously said, a symmetric-key encryption algorithm, defined in definition 1.5, is a cryptographic algorithm that uses the same secret key for both encryption and decryption of data. Examples of symmetric-key encryption algorithms include AES, DES, and Blowfish.

On the other hand, an asymmetric-key encryption algorithm, also known as a public-key encryption algorithm, defined in definition 1.6, uses two different keys: a public key and a private key. The public key is used for encrypting the data, while the private key is used for decrypting the data. This means that the encryption process can be performed by anyone who has access to the public key, but the decryption process can only be performed by the owner of the private key. Examples of asymmetric-key encryption algorithms include RSA, Elliptic Curve Cryptography (ECC), and Diffie-Hellman key exchange.

In addition to the key usage differences, symmetric-key encryption algorithms are typically faster and more efficient than asymmetric-key encryption algorithms. However, they are less secure due to the requirement of sharing the secret key among all parties involved in the encryption and decryption process. Asymmetric-key encryption algorithms, on the other hand, are considered more secure as the private key is not shared with anyone and can only be used by the owner of the key.

Definition 1.5 (Symmetric-key encryption algorithm)**.** *Let* $S(\mathrm{E}, \mathrm{D}, k)$ *be a symmetric-key encryption algorithm, also known as shared-key encryption algorithm, over a domain* $\mathbb{D}$*, where* E *is an encryption method,* D *is a decryption method, and* $k \in \mathbb{D}^l$ *is the shared-key. For any plaintext* $m \in \mathbb{D}^n$*, where* $n \geq l$*, it holds that* $\mathrm{D}(\mathrm{E}(m, k), k) = m$*.*

Definition 1.6 (Asymmetric-key encryption algorithm)**.** *Let* $A(\mathrm{E}, \mathrm{D}, k_1, k_2)$ *be an asymmetric-key encryption algorithm, also known as public-key encryption algorithm, over a domain* $\mathbb{D}$*, where* E *is an encryption method,* D *is a decryption method,* $k_1 \in \mathbb{D}^{l_1}$ *is a public-key, and* $k_2 \in \mathbb{D}^{l_2}$ *is a private-key, where* $l_1, l_2 \in \mathbb{N}$ *are not necessarily distinct. For any plaintext* $m \in \mathbb{D}^n$*, where* $n \geq l_1$ *and* $n \geq l_2$*, it holds that* $\mathrm{D}(\mathrm{E}(m, k_1), k_2) = m$*.*

Remark 1.3. *In symmetric-key encryption, there are two main types of ciphers: stream ciphers and block ciphers. Stream ciphers encrypt the plaintext one bit or byte*

at a time. They generate a stream of keystream bits, which are combined with the plaintext bits in a bit-by-bit or byte-by-byte exclusive-OR operation. The keystream is typically generated by a pseudo-random number generator, which is initialized with a secret key. On the contrary, block ciphers divide the plaintext into fixed-size blocks and encrypt them in sequence, with the encryption of each block being dependent on the previous blocks. They apply the same transformation to each block of plaintext, using a secret key. The transformation is typically a combination of permutations and substitutions, which is repeated several times (rounds) in order to increase the security. The main difference between a stream cipher and a block cipher is that the latter divides the plaintext into fixed-size blocks and encrypts them independently, while the former encrypts the plaintext one bit or byte at a time. Stream ciphers are typically faster and more efficient than block ciphers, but they are less secure since they can be vulnerable to certain attacks such as plaintext attacks. On the other hand, block ciphers are more secure and resistant to such attacks, but they are slower and less efficient.

Although one can define a security level, it is acknowledged that any cryptographic algorithm is breakable, meaning that there is always at least one attack that is capable of retrieving the encryption key, plaintext, or any other secret parameter of the algorithm. For this reason, it is important to define when a cryptographic algorithm is considered secure and under what types of attacks it is vulnerable.

Definition 1.7 (Cryptographic attack)**.** *A cryptographic attack is a method that attempts to violate the security of a cryptographic system by identifying one or more weaknesses in one or more of its components. In this case, one says that a cryptographic attack breaks a cryptographic algorithm.*

Cryptographic attacks can take many forms, each focusing on different aspects of a cryptographic algorithm in an attempt to break it. For example, some attacks exploit the choice of algorithm parameters to optimize the process of breaking the algorithm, while others seek flaws in the cryptographic protocol in order to decode messages without knowledge of the key. The simplest form of attack, which makes no assumptions about the data, is known as a brute force attack.

Definition 1.8 (Brute force attack)**.** *A brute force attack, also known as exhaustive search, is a type of cryptographic attack in which an attacker systematically tries all possible combinations of inputs in order to find the correct one, such as attempting every possible key to decrypt a ciphertext.*

A typical example of a brute force attack is the exhaustive search for the encryption key. For instance, consider a cryptographic algorithm that utilizes an 80-bit key. The brute force attack requires an exhaustive key-search by trying all possible 2^{80} keys. However, in practice, the number of keys to search can be much lower than the total number of possible keys, making it possible for modern computers to quickly perform a brute force attack on certain cryptographic algorithms. Furthermore, the brute force attack can be made more efficient by using techniques such as pre-computation and table lookups, which can significantly reduce the number of keys that needed to be searched. Additionally, it is important to consider the possibility of parallelization and the use of specialized hardware to speed up the attack. Therefore, when evaluating the security of a cryptographic algorithm, it is essential to take into account not only the number of possible keys but also the potential computational resources available to an attacker.

Definition 1.9 (Power of a computer)**.** *The power of a computer, in the context of cryptography, refers to the number of floating point operations per second (FLOPS) it can perform.*

Remark 1.4. *Another unit of measurement commonly used in cryptography is the million instructions per second (MIPS), which measures the number of instructions per second executed by a microprocessor. However, to use this unit of measurement, it is necessary to measure the number of machine instructions (i.e., instructions executed by the microprocessor) that the cryptographic attack performs. This can be used to compare the computational resources required for different attacks on the same cryptographic algorithm.*

A secure cryptographic algorithm, as defined in definition 1.10, is one that is able to withstand the most efficient known attack, as determined by the current most powerful computer in the world. The security strength of the algorithm is measured in terms of the number of steps required to break the algorithm, with a higher number indicating a stronger level of security. The security level of the algorithm should be significantly greater than the computational power of the most powerful computer in order to consider the algorithm secure.

Definition 1.10 (Secure cryptographic algorithm)**.** *Let C_A be a cryptographic algorithm, and let $\mathcal{O}(f(x))$ be the cost of decrypting a ciphertext generated by C_A by an authorized user. Let B be the most efficient cryptographic attack that can currently break C_A. The algorithm C_A is considered to be secure (with respect to B) if the cost of B is $\mathcal{O}\left(e^{f(x)}\right)$.*

Remark 1.5. *It should be noted that the security of a cryptographic algorithm, as described in definition 1.10, does not take into account the increasing computational power of modern computers. As computing power grows, the parameters of a secure cryptographic algorithm, such as key length, must also increase to maintain a desired level of security. Specifically, let K be the power (in FLOPS) of the current most powerful computer. Hence, the algorithm C_A in definition 1.10 is secure (with respect to B) if the cost of B is $\mathcal{O}\left(e^{f(x)}\right)$, and its security strength (or security level) with respect to B is at least t bits, meaning that the number of steps required to break C_A with B is equal to 2^t, with $2^t \gg K$.*

Remark 1.6. *An equivalent way to evaluate the security of the cryptographic algorithm C_A with respect to a specific attack B is to consider the number of instructions required to execute the attack. Specifically, if the number of instructions n required to execute the attack B is significantly greater than the number of MIPS K of the current most powerful computer, then the algorithm C_A can be considered secure against that attack.*

In definition 1.10, we state that a cryptographic algorithm is considered secure if it is considered to be “hard” to break using the most efficient attack currently known. The number 2^t must be significantly greater than the number K. For instance, the top security level accepted today is at least 128 bits (or at least 80 for standard secret messages), while the most powerful computer has $K \approx 415 \times 10^{15} \approx 2^{52}$ FLOPS. This means that it would take approximately $2^{128} / 2^{52} = 2^{76}$ seconds, or over 2.4 quadrillion years (i.e., 10^{15} years), to break a cryptographic algorithm (with a security level of 128 bits) using the most efficient attack currently known. Specifically, in this case, we considered the brute force attack since the number of keys to be checked is given by the ratio of the security level and the power of the most powerful computer

in the world. It is important to note that any other attack that utilizes a subset of keys would reduce the search time even further.

In order to conclude this chapter, it is worth noting that a fundamental feature of cryptographic algorithms is that they must be made publicly available. This ensures the portability and widespread use of these algorithms in modern technologies. Additionally, it guarantees that there are no hidden vulnerabilities or "backdoors" that would allow the creators of the algorithm to decrypt messages without the use of the encryption key.

Definition 1.11 (Backdoor)**.** *In cryptography, a backdoor is a mechanism or technique that enables the creators of an algorithm to gain unauthorized access to the cryptographic system or algorithm, thereby compromising the confidentiality and integrity of the system. Such a mechanism can provide the ability to bypass authentication procedures or gain access to encrypted data without the proper decryption key.*

Backdoors can be deliberately inserted by the creators of the cryptographic system for the purpose of gaining access to sensitive information, or they can be the result of a vulnerability or flaw in the design of the system. Backdoors pose a significant security risk as they undermine the integrity and confidentiality of the cryptographic system, and they can be exploited by unauthorized actors to gain access to sensitive information.

Although backdoors are often used for legitimate purposes, such as performing recovery procedures that would otherwise be impossible or very difficult, their existence can compromise the security of a cryptographic protocol. Even the National Institute of Standards and Technology (NIST), a leading institute that also standardizes various cryptographic protocols, has been shown [4, 7, 11, 13] to have backdoors within its protocols.

Remark 1.7. *It is worth mentioning that the field of study that deals with the security of various cryptographic protocols is called cryptanalysis, and its goal is to break cryptographic systems by identifying any weaknesses.*

Specifically, cryptanalysis is the study of methods for obtaining encrypted information without access to the secret key. Additionally, it analyzes a cryptographic system or algorithm to identify weaknesses or vulnerabilities that could be exploited to gain unauthorized access to encrypted information. Cryptanalysis can take several forms, including breaking a ciphertext to obtain the plaintext, breaking a digital signature to forge a new one, or discovering the key used to encrypt or decrypt the information. Finally, cryptanalytic techniques can include, but are not limited to, brute-force attacks, differential cryptanalysis, linear cryptanalysis, and side-channel attacks.

2 Algebraic curves

Elliptic curves are a specific type of algebraic curve defined over a field that are commonly utilized in cryptographic contexts. In order to understand them, it is necessary to first familiarize oneself with some fundamental concepts. For a general introduction, readers are directed to a classic reference, e.g. [43].

2.1 Projective space

In this section, we provide definitions of projective space in order to facilitate the explanation of elliptic curves in the following chapter.

Projective space is a fundamental concept in algebraic geometry. It is a way of extending the notion of Euclidean space to include "points at infinity", which cannot be represented in the traditional Cartesian coordinate system.

The concept of projective space originated in the field of projective geometry, which was developed in the late 19th century as a way to study geometric properties that remain invariant under projection. One of the main figures in the development of projective geometry was the French mathematician Jean-Victor Poncelet, who in his work "Traité des propriétés projectives des figures" (Treatise on the Projective Properties of Figures), published in 1822, laid the foundation for the study of projective geometry.

Poncelet's work was motivated by the need to understand geometric properties that are not affected by changes in position or orientation. He observed that many geometric figures, such as lines and conic sections, possess properties that remain unchanged under projection. This led him to introduce the concept of a "projective invariant", which is a property of a figure that remains unchanged under projection.

Poncelet's work was further developed by other mathematicians such as Augustin-Louis Cauchy, who introduced the concept of "projective transformation" which is a transformation that preserves the incidence relations between points and lines. This led to the development of the group of projective transformations, known as the projective linear group. Additionally, these transformations can be represented by matrices and the group law is the matrix multiplication.

The concept of projective space was later formalized by the German mathematician David Hilbert, who in his work "Foundations of Geometry" (1899) defined projective space, as stated in definition 2.1, as the set of equivalence classes of non-zero vectors in a vector space over a field, where two vectors are considered equivalent if they are proportional. This is known as the "homogeneous coordinates" representation of a point. For example, in a two-dimensional projective space, a point (x, y) can also be represented as (rx, ry) for any non-zero scalar r.

In the context of projective geometry, a point in projective space is represented by a homogeneous vector, i.e. a vector whose entries are elements of a field and are not all

zero. The dimension of a projective space refers to the number of coordinates needed to specify a point in that space. For instance, a projective space of dimension n over a field $\mathbb{K}$ is denoted by $\mathbf{P}^n(\mathbb{K})$. In other words, it is a set of all the one-dimensional subspaces of a vector space of dimension $n+1$ over $\mathbb{K}$.

Definition 2.1 (Projective space)**.** *If $V = \mathbb{K}^{n+1}$ is a vector space over a field $\mathbb{K}$, then the projective space $\mathbf{P}^n(\mathbb{K})$ is the set of equivalence classes of $V \setminus \{\mathbf{0}\}$ under the equivalence relation $(\sim)$ defined by $x \sim y$, for $x, y \in V$, if there is a non-zero element $\lambda \in \mathbb{K}^*$ such that $x = \lambda y$.*

Definition 2.2 (Homogeneous coordinates)**.** *Let $[P]$ be an element of $\mathbf{P}^n(\mathbb{K})$, one says that P is a projective point of projective (or homogeneous) coordinates $[x_0 : x_1 : \ldots : x_n]$, and one writes $P = [x_0 : x_1 : \ldots : x_n]$ if the vector $[x_0, x_1, \ldots, x_n]$ is a representative of the class $[P]$. One simply writes $P \in \mathbf{P}^n(\mathbb{K})$ to mean the representative of the class $[P]$.*

Remark 2.1. *As pointed out previously and according to definition 2.1, every point in the projective space is equivalent to a scalar multiplicative of its coordinates, that is, if $P \in \mathbf{P}^n(\mathbb{K})$ is such that $P = [x_0 : x_1 : \ldots : x_n]$, then $P \equiv [\lambda x_0 : \lambda x_1 : \ldots : \lambda x_n] = \lambda P$, where $\lambda \in \mathbb{K}^*$.*

Thus, one defines a point in dehomogenized coordinates by simply dividing its coordinates by a scalar that makes the first non-zero coordinate equal to 1.

Definition 2.3 (Dehomogenized coordinates)**.** *Let P be an element of $\mathbf{P}^n(\mathbb{K})$, and let $0 \leq i < n$ be the first index such that $x_i \neq 0$, that is, $P = [0 : \ldots : 0 : x_i : \ldots : x_n]$, with $x_i \neq 0$. One can dehomogenize P by dividing each coordinate of P for x_i, that is, $P = \left[0 : \ldots : 0 : \frac{x_i}{x_i} : \ldots : \frac{x_n}{x_i}\right]$.*

Hence, by representing each element of $\mathbf{P}^n(\mathbb{K})$ in dehomogenized coordinates, one has an unique representative for each equivalence classes.

In the following, we present two types of points: proper and improper points. Specifically, the latter distinction is given by projective geometry. On one hand, proper points, also known as finite points, are the points that can be represented by a set of coordinates in the traditional Cartesian coordinate system. These points are located within the projective space and have a finite distance from the origin. On the other hand, improper points, also known as points at infinity, are points that cannot be represented by a set of coordinates in the traditional Cartesian coordinate system. These points are located outside the projective space and have an infinite distance from the origin. They are often considered as an extension of the projective space. Therefore, the projective space can be considered as an extension of the affine space, incorporating the concept of improper points.

The distinction between proper and improper points arises from the idea that geometric figures can be transformed by projection and, as previously established, the properties of a figure that remain unchanged under projection are called projective invariants. The concept of a point at infinity allows us to extend the projective space and consider a point at infinity as a point of the space, making it important for the study of projective geometry.

The reason why they are called in this way is because proper points are considered to be “proper” points in the traditional sense, while improper points are considered to be “improper” or “at infinity” as they cannot be represented by coordinates in the traditional sense.

In summary, proper points are finite points within the projective space while improper points are the points at infinity, which are located outside the projective space and are considered to extend the projective space. This distinction is important for the study of projective geometry as it allows for the consideration of points at infinity as proper points in the space.

Specifically, these two types of points are distinguished, conventionally, according to the value of the first coordinate x_0 of a point, P.

Definition 2.4 (Proper and improper points)**.** *Let P be an element of $\mathbf{P}^n(\mathbb{K})$. If $P = [x_0 : x_1 : \ldots : x_n]$ with $x_0 \neq 0$, then P is a proper point, otherwise, if $x_0 = 0$, then P is an improper point (also known as point at infinity).*

Remark 2.2. *Note that referring to the elements of $\mathbf{P}^n(\mathbb{K})$ with $x_0 = 0$ as "improper points" is simply a convention. In general, elements of $\mathbf{P}^n(\mathbb{K})$ with $x_i = 0$, where $0 \leq i \in \mathbb{Z}$, can also be referred to as improper points. The choice of coordinate only affects the reference to the (projective) line at infinity. In the case presented, the line at infinity is defined by the equation $x_0 = 0$, otherwise, in other references it could be defined by the equation $x_i = 0$. Both choices are equivalent, and it is possible to move from one reference system to the other and vice versa through appropriate transformations.*

In order to conclude this section, we present two well-known special instances of projective space: the projective line and the projective plane. On one hand, a projective line over a field $\mathbb{K}$, denoted by $\mathbf{P}^1(\mathbb{K})$, is defined as the set of all one-dimensional subspaces of a vector space of dimension 2 over $\mathbb{K}$. This is known as a "Riemann sphere" representation of the complex numbers. On the other hand, a projective plane over a field $\mathbb{K}$, denoted by $\mathbf{P}^2(\mathbb{K})$, is defined as the set of all one-dimensional subspaces of a vector space of dimension 3 over $\mathbb{K}$.

For the latter, a bijective correspondence is defined between each point $P \in \mathbf{P}^2(\mathbb{K})$ such that $x_0 \neq 0$ and a point on the affine plane. In particular, let $P \in \mathbf{P}^2(\mathbb{K})$ be a point whose homogeneous coordinates are $[x_0 : x_1 : x_2] = [Z : X : Y]$. If $Z \neq 0$, then the bijective function $P \mapsto \overline{P} = \left(\frac{X}{Z}, \frac{Y}{Z}\right) = (x, y)$ maps the projective point P onto the affine point $\overline{P}$. In this case, it follows that $P = [Z : X : Y] \equiv (x, y)$. Similarly, if $Z = 0$ and $X \neq 0$, then one maps the improper points uniquely onto a line, specifically $[0 : X : Y] \equiv \left[0 : \frac{X}{X} : \frac{Y}{X}\right]$. Finally, if $Z = 0$ and $X = 0$, then the point $[0 : 0 : Y] \equiv \left[0 : 0 : \frac{Y}{Y}\right] = \Omega$ is an additional point commonly referred as a further point at infinity.

2.2 Homogeneous rational functions

Homogeneous polynomials and rational homogeneous functions are mathematical concepts that are widely used in various fields of mathematics, such as algebraic geometry, number theory, and cryptography. On one hand, a homogeneous polynomial is a polynomial in which the degree of each variable is the same, and the coefficients are not dependent on the variables. Additionally, homogeneous polynomials can be used to define algebraic varieties in projective space, which are important objects in algebraic geometry. On the other hand, rational homogeneous functions are a type of homogeneous polynomials in which the coefficients are rational numbers. They are often used in the study of projective varieties, for instance, to parameterize algebraic

curves, and they can be defined as a ratio of two homogeneous polynomials with the same degree.

In general, homogeneous polynomials and rational homogeneous functions are important tools in algebraic geometry and other related fields, providing a way to study geometric objects and algebraic equations in a more general and abstract setting.

Definition 2.5 (Homogeneous polynomial)**.** *Let $f \in \mathbb{K}[x_0, x_1, \ldots, x_n]$ be a polynomial in $n+1$ variables. If all non-zero terms of f have a degree equal to $0 \leq d \in \mathbb{N}$, then f is referred to as a homogeneous polynomial of degree d.*

Definition 2.6 (Homogeneous rational function)**.** *If $f, g \in \mathbb{K}[x_0, x_1, \ldots, x_n]$ are two homogeneous polynomials of degree d, then $F = \frac{f}{g}$ is referred to as a homogeneous rational function of degree d.*

Homogeneous polynomials and rational homogeneous functions possess several important properties that are widely studied and utilized in various branches of mathematics.

One of the main properties of homogeneous polynomials is that they are closed under scalar multiplication. This means that if a homogeneous polynomial is multiplied by a scalar, the resulting polynomial will still be homogeneous. This property is a direct result of the fact that the degree of each variable in a homogeneous polynomial is the same, making it easy to maintain this property after scalar multiplication.

Another important property of homogeneous polynomials is that they can be represented by homogeneous coordinates. This allows for a more compact and elegant representation of geometric objects, such as points and lines in projective space.

Rational homogeneous functions also possess several important properties. In particular, let P be an element in $\mathbf{P}^n(\mathbb{K})$, if f is a homogeneous polynomial of degree d, then $f(\lambda P) = \lambda^d f(P)$, where $\lambda \in \mathbb{K}^*$. Furthermore, if $F = \frac{f}{g}$ is a homogeneous rational function of degree d, then it follows that

$$F(\lambda P) = \frac{f(\lambda P)}{g(\lambda P)} = \frac{\lambda^d f(P)}{\lambda^d g(P)} = F(P).$$

Thus, the evaluation at $P \in \mathbf{P}^n(\mathbb{K})$ of a homogeneous rational function is the same for every $\lambda P = Q \equiv P$.

Definition 2.7 (Homogenization)**.** *Let $f \in \mathbb{K}[x_1, x_2, \ldots, x_n]$ be a non-homogeneous polynomial of degree d. One can homogenize f, i.e., convert it into a homogeneous polynomial, by introducing a new variable x_0 and multiplying each non-zero term of degree k_i of f times ${x_0}^{d-k_i}$. The polynomial ${}^h f$ is referred to as the homogenization of the polynomial f, and it is a homogeneous polynomial of degree d.*

Definition 2.8 (Dehomogenization)**.** *Let ${}^h f \in \mathbb{K}[x_0, x_1, \ldots, x_n]$ be a homogeneous polynomial of degree d. One can dehomogenize ${}^h f$, i.e., convert it back into a non-homogeneous polynomial, by setting $x_0 = 1$, resulting in $f(x_1, \ldots, x_n)$, which is referred to as the dehomogenization of ${}^h f$.*

Another important property of rational homogeneous functions is that they can be decomposed into their zeros and poles. This decomposition is known as the factorization of the function, and it is achieved by expressing the function as the ratio of two polynomials, where the zeros of the function correspond to the zeros of the numerator polynomial and the poles of the function correspond to the zeros of the denominator

polynomial. The decomposition of the function in zeros and poles is a fundamental tool to study their properties.

Definition 2.9 (Zeros and poles)**.** *Let $F = \frac{f}{g}$ be a homogeneous rational function of degree d. A point P is a zero of F (and, as a result, a zero of f) if the polynomial f vanishes at P, while a point Q is a pole of F (and, as a result, a zero of g) if the polynomial g vanishes at Q.*

One of the fundamental theorems in algebraic geometry is Bezout's theorem for rational homogeneous functions, which states that the number of zeros of a homogeneous rational function is equal to the number of its poles. This theorem is derived from the factorization of the function into its zeros and poles and holds for any algebraic closed field and for any homogeneous rational function defined in a projective space.

Theorem 2.1 (Bezout's theorem)**.** *Let $F = \frac{f}{g}$ be a homogeneous rational function of degree d over an algebraically closed field $\overline{\mathbb{K}}$. The number of zeros of F is equal to the number of poles of F over the algebraic closure of the defining field of F.*

2.3 Algebraic variety

Algebraic varieties are one of the central objects of study in algebraic geometry. They are geometric objects that are defined by the common solutions of a system of polynomial equations. The study of algebraic varieties dates back to the 19th century with the work of mathematicians such as Augustin-Louis Cauchy, Jean-Victor Poncelet, and Carl Friedrich Gauss. However, it was not until the work of David Hilbert and his school of algebraic geometry in the late 19th and early 20th century that the field began to take shape.

The definition of an algebraic variety is quite general and encompasses a wide range of geometric objects, including affine and projective varieties, as well as complex varieties. As stated in definition 2.11, an affine variety is a common solution set of a system of polynomial equations in affine space, while a projective variety, as stated in definition 2.12, is a common solution set of a system of homogeneous polynomial equations in projective space. Complex varieties, on the other hand, are the common solution sets of a system of polynomial equations in complex space.

One of the main results in algebraic geometry is the classification of algebraic varieties. This classification is based on the dimension of the variety and its degree. The dimension of an algebraic variety is the maximum number of linearly independent vectors that can be found in the variety. The degree of an algebraic variety is the maximum number of points of intersection of the variety with a general linear space of the same dimension.

One of the most important results in the study of algebraic varieties is the Nullstellensatz, or "zero-set theorem". This theorem, first proven by David Hilbert, states that the radical ideal generated by the polynomials that define a variety is equal to the ideal generated by the polynomials that vanish on the variety. This result has important consequences in the study of algebraic geometry, including the ability to study the algebraic properties of a variety, such as its dimension, by studying the ideal generated by the polynomials that define it.

Algebraic varieties also play an important role in many other branches of mathematics, such as number theory, cryptography, and coding theory. For example, the theory of elliptic curves, which are a special type of algebraic varieties, is used in the study of

cryptography and coding theory. Algebraic geometry also has important applications in computer science, such as in the study of computational complexity and in the design of algorithms for solving polynomial equations.

Definition 2.10 (Algebraic variety)**.** *Let $\overline{\mathbb{K}}$ be an algebraically closed field, one says that $\mathcal{V}$ is an algebraic variety if $\mathcal{V}$ is the set of solutions (the zero set) of a system of polynomial equations over $\overline{\mathbb{K}}$.*

Definition 2.11 (Affine variety)**.** *Let $\overline{\mathbb{K}}$ be an algebraically closed field, and let n be a natural number, one says that $\mathbf{A}^n$ is an affine vector space of dimension n over $\overline{\mathbb{K}}$. Let S be a set of non-homogeneous polynomials $f \in \overline{\mathbb{K}}[x_1, \ldots, x_n]$, one defines*

$$Z(S) = \{\mathbf{x} \in \mathbf{A}^n \colon f(\mathbf{x}) = 0 \text{ for all } f \in S\}.$$

Hence, if $\mathcal{V}$ is a nonempty subset of $\mathbf{A}^n$ such that $\mathcal{V} = Z(S)$ for some S, then $\mathcal{V}$ is a nonempty affine algebraic set, also known as affine variety.

Definition 2.12 (Projective variety)**.** *Let $\overline{\mathbb{K}}$ be an algebraically closed field, and let n be a natural number, one says that $\mathbf{P}^n$ is a projective vector space of dimension n over $\overline{\mathbb{K}}$. Let S be a set of homogeneous polynomials $f \in \overline{\mathbb{K}}[x_0, x_1, \ldots, x_n]$ of degree d, one defines $Z(S) = \{\mathbf{x} \in \mathbf{P}^n \colon f(\mathbf{x}) = 0 \text{ for all } f \in S\}$. Hence, if $\mathcal{V}$ is a nonempty subset of $\mathbf{P}^n$ such that $\mathcal{V} = Z(S)$ for some S, then $\mathcal{V}$ is a nonempty projective algebraic set, also known as projective variety.*

An interesting property of algebraic varieties is irreducibility, meaning that they cannot be expressed as a union of two or more proper algebraic subvarieties. Additionally, they are closely related to the concept of prime ideals in algebraic geometry. In particular, the ideal of an irreducible variety is a prime ideal. Furthermore, irreducible varieties are also important in the study of singularities, i.e., points where the algebraic variety is not locally "smooth". Overall, understanding the concept of irreducibility is crucial for understanding the structure of algebraic varieties and for applying algebraic techniques to geometric objects.

Definition 2.13 (Irreducible variety)**.** *If $\mathcal{V}$ is an (affine or projective) algebraic variety, then $\mathcal{V}$ is considered to be irreducible if it cannot be expressed as the union of two or more proper algebraic subvarieties.*

Algebraic curves are a specific subset of algebraic varieties, defined as the common solution set of a single polynomial equation in two variables. The study of algebraic curves is a fundamental aspect of algebraic geometry. An important result in the study of algebraic curves is the Riemann-Roch theorem, which relates the dimension and degree of a curve (or a variety) to its genus and the number of its singularities. This theorem is a fundamental tool in the study of algebraic varieties, and has important applications in the study of algebraic geometry and other branches of mathematics.

Definition 2.14 (Affine algebraic curve)**.** *Let $\mathbb{K}$ be a field. An affine algebraic plane curve, or simply affine algebraic curve, is defined as the zero set over $\mathbb{K}$ in the affine plane of a non-homogeneous polynomial in two variables, typically denoted as x and y.*

Definition 2.15 (Projective algebraic curve)**.** *Let $\mathbb{K}$ be a field. A projective algebraic plane curve, or simply projective algebraic curve, is defined as the zero set over $\mathbb{K}$ in the projective plane of a homogeneous polynomial in three variables, typically denoted as X, Y, and Z.*

Remark 2.3. *It is possible to convert an affine algebraic curve into a projective algebraic curve by a process known as completion. This is achieved by homogenizing the defining polynomial of the affine curve. Conversely, a projective algebraic curve can be converted into an affine algebraic curve by a process known as restriction, which is achieved by setting $Z = 1$. Since these operations are inverses of each other, it is common to refer to algebraic curves without specifying whether they are affine or projective.*

2.4 Divisor of algebraic curves

In algebraic geometry, the concept of divisors on algebraic curves $\mathcal{C}$ is used to classify the zeros and poles of rational functions on the curve. As outlined in definition 2.17, a divisor on an algebraic curve is defined as a formal sum of points on the curve, with each point having an associated coefficient that represents the multiplicity of that point.

Definition 2.16 (Free group)**.** *The free group over a set $S = \{x_i\}$, with $i \in I$, where I is a set of indices, is the group G whose objects are (formal) products, that is, they are obtained as simply the superposition of elements of S,*

$$g = \prod_{j \in J} x_j{}^{\epsilon_j}, \quad J \subseteq I, \epsilon_j = \pm 1, \tag{2.1}$$

by extending to a finite set of indices J, such that there is only the rule $x \cdot x^{-1} = x^{-1} \cdot x = 1$. In addition, if one imposes that the elements of G have to commute, then the group G is said to be a free abelian group, where the operation is $(+)$ and 0 is the neutral element.

Definition 2.17 (Divisor of an algebraic curve)**.** *Let $\mathcal{C}$ be an algebraic curve over a field $\mathbb{K}$. A divisor of $\mathcal{C}$ is an element of the free abelian group $\mathrm{Div}(\mathcal{C})$ over the set $\mathcal{C}(\mathbb{K})$ of $\mathbb{K}$-rational points in $\mathcal{C}$, that is, it is a formal sum $D = \sum_{P \in \mathcal{C}(\mathbb{K})} n_P \cdot P$, where $n_P \in \mathbb{Z}$ is the multiplicity of P in D, and only a finite number of n_P is non-zero.*

As there are many points belonging to $\mathcal{C}$, it is convenient to represent the set of points belonging to a divisor D, that is, the set of points such that $n_P \neq 0$ in D.

Definition 2.18 (Support of a divisor)**.** *Let $\mathcal{C}$ be an algebraic curve over a field $\mathbb{K}$, and let D be a divisor in $\mathrm{Div}(\mathcal{C})$. The support of the divisor D is the set of points belonging to D such that $n_P \neq 0$. Specifically, if $\mathrm{supp}(D) = \{P_1, \ldots, P_k\}$, then $D = \sum_{i=1}^{k} n_{P_i} \cdot P_i$, with $n_{P_i} \neq 0$.*

One of the main properties of divisors is that they have an associated degree, as stated in definition 2.19, which is defined as the sum of the coefficients of the points in the divisor. The degree of a divisor is a useful invariant that can be used to classify divisors into different types, such as effective divisors (as stated in definition 2.20), which have non-negative coefficients.

Definition 2.19 (Degree of a divisor)**.** *Let D be a divisor of an algebraic curve $\mathcal{C}$, that is, $\sum_{P \in \mathrm{supp}(D)} n_P \cdot P = D \in \mathrm{Div}(\mathcal{C})$. Hence, one defines the degree of the divisor D as $\deg(D) = \delta(D) = \sum_{P \in \mathrm{supp}(D)} n_P \in \mathbb{Z}$.*

Definition 2.20 (Effective divisor)**.** *Let $\mathcal{C}$ be an algebraic curve. The divisor*

$$D = \sum_{P \in \mathrm{supp}(D)} n_P \cdot P$$

is considered to be an effective divisor of $\mathcal{C}$ if for any $P \in \text{supp}(D)$, then it holds that $n_P \geq 0$.

An important class of divisors for an algebraic curve is the class of zero degree divisors.

Definition 2.21 (Zero degree divisor). *Let $\mathcal{C}$ be an algebraic curve. The divisor D is considered to be a zero degree divisor of $\mathcal{C}$ if $\delta(D) = 0$.*

Theorem 2.2 (Zero degree divisors of an algebraic curve). *The set of all the zero degree divisors of an algebraic curve $\mathcal{C}$, also known as $\text{Div}^0(\mathcal{C})$, constitutes an additive subgroup of the free abelian group $\text{Div}(\mathcal{C})$.*

Definition 2.22 (Principal divisor of a homogeneous rational function). *Let $F = \frac{f}{g}$ be a homogeneous rational function of degree d. A principal divisor of F is the zero degree divisor $\text{div}(F) = \sum_{i=1}^{n} P_i - \sum_{i=1}^{n} Q_i$, where P_i and Q_i are the zeros and the poles of F, respectively.*

Definition 2.23 (Principal divisors group). *Let $F = \frac{f}{g}$ be a homogeneous rational function of degree d. The group of principal divisors of F, denoted as $\text{Princ}(F)$, forms an equivalence class under the equivalence relation $(\sim)$ such that, for any $D_1, D_2 \in \text{Princ}(F)$, one has that $D_1 \sim D_2$ if $D_1 = D_2 + \text{div}(G)$ for a suitable homogeneous rational function G.*

Given the concepts of divisors and principal divisors of an algebraic curve, one has all the necessary components to define the group associated with an algebraic curve $\mathcal{C}$.

Definition 2.24 (Jacobian of an algebraic curve). *Let $\mathcal{C}$ be an algebraic curve over a field $\mathbb{K}$. The Jacobian $\mathcal{J}(\mathcal{C})$ of $\mathcal{C}$ is defined as the group of zero degree divisors of $\mathcal{C}$, modulo the group of principal divisors of $\mathcal{C}$, and it is expressed as:*

$$\mathcal{J}(\mathcal{C}) = \text{Div}^0(\mathcal{C})/\text{Princ}(\mathcal{C}). \tag{2.2}$$

3 Elliptic curves

In this chapter, we introduce the concept of elliptic curves, based on the definitions previously discussed. We also provide a brief overview to several types of elliptic curves, including those in Weierstrass form [15, 21, 44, 47, 73], Montgomery form [61], Edwards form [8, 9, 28, 52], and the more general hyperelliptic curves [18, 49].

In the following, in order to discuss elliptic curves, it is necessary to first define and distinguish between certain concepts related to algebraic geometry such as regular functions, algebraic groups, and abelian varieties.

Definition 3.1 (Regular function)**.** *A regular function is a morphism from an algebraic variety to the affine line.*

Definition 3.2 (Algebraic group)**.** *An algebraic group G is both a group and an algebraic variety, with group operations (multiplication and inversion) given by regular functions.*

Definition 3.3 (Abelian variety)**.** *An abelian variety $\mathcal{V}$ is a projective algebraic variety that is also an algebraic group.*

Definition 3.4 (Smooth algebraic curve)**.** *An algebraic curve $\mathcal{C}$ is considered to be smooth, or non-singular, if it is a one-dimensional, irreducible, abelian variety over an algebraically closed field.*

Remark 3.1. *Note that a smooth algebraic curve is a projective algebraic curve that is characterized by its property of being infinitely differentiable, that is, it does not have any singularity points.*

Definition 3.5 (Elliptic curve)**.** *An elliptic curve $\mathcal{C}$ over a field $\mathbb{K}$ is a smooth, projective algebraic curve of genus $g = 1$. The curve is equipped with a group law, defined by the algebraic structure, and a distinguished point O, called the identity element, which serves as the neutral element of the group operation. The points on the curve, along with the identity element, form an abelian group under this operation, referred to as the group law of the elliptic curve.*

Remark 3.2. *It is noteworthy that the elements D of the Jacobian $\mathcal{J}(\mathcal{C})$ of an elliptic curve $\mathcal{C}$ over a field $\mathbb{K}$ have the following canonical forms: $D = O - O$ (the identity element of the group, represented simply as $D = O$) or $D = P - O$ (or simply $D = P$), where $P \in \mathcal{C}(\mathbb{K})$.*

3.1 Elliptic curves in Weierstrass form

Elliptic curves in Weierstrass form were first studied by K.T.W. Weierstrass and are the most commonly used in elliptic curve cryptography. Their properties depend on the field $\mathbb{K}$ over which they are defined. For more information on elliptic curves, we address the reader to classic book, e.g. [73].

Definition 3.6 (Generalized elliptic curve in Weierstrass form)**.** *An elliptic curve in generalized Weierstrass form, denoted as $\mathcal{W}$, over a field $\mathbb{K}$ is defined by the equation*

$$y^2 + (a_1x + a_3)y = x^3 + a_2x^2 + a_4x + a_6, \tag{3.1}$$

where $a_1, a_2, a_3, a_4, a_6 \in \mathbb{K}$. The curve $\mathcal{W}$ has at least a $\mathbb{K}$-rational point, i.e., the point at infinity, commonly denoted as $\Omega = [0 : 0 : 1]$, which serves as the identity element of the group defined by the algebraic structure of $\mathcal{W}$.

In the following, we provide the formulas to compute the discriminant and the j-invariant of elliptic curves. These parameters are important in determining certain interesting properties of these curves. Given the elliptic curve in generalized Weierstrass form $\mathcal{W}$, the parameters $b_2, b_4, b_6, b_8, c_4 \in \mathbb{K}$ are defined as

$$\begin{aligned} b_2 &= {a_1}^2 + 4a_2, & b_4 &= 2a_4 + a_1a_3, \\ b_6 &= {a_3}^2 + 4a_6, & b_8 &= {a_1}^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2{a_3}^2 - {a_4}^2, \\ c_4 &= {b_2}^2 - 24b_4. \end{aligned}$$

The discriminant of the curve, denoted as $\Delta_{\mathcal{W}}$, and the j-invariant, denoted as $j_{\mathcal{W}}$, are then computed as

$$\Delta_{\mathcal{W}} = -{b_2}^2b_8 - 8{b_4}^3 - 27{b_6}^2 + 9b_2b_4b_6, \qquad j_{\mathcal{W}} = \frac{{c_4}^3}{\Delta_{\mathcal{W}}}.$$

As previously claimed, the j-invariant of an elliptic curve is an important parameter. For instance, it is important to determine whether two curves are isomorphic, as well to classify elliptic curves into different families. Specifically, two elliptic curves $\mathcal{C}_1$ and $\mathcal{C}_2$ are isomorphic over an algebraically closed field $\overline{\mathbb{K}}$ if their j-invariant is the same.

Remark 3.3. *Note that the discriminant is a measure of the singularities of an elliptic curve, and is equal to zero if and only if the curve is singular (or non-smooth).*

When dealing with elliptic curves over a finite field, one must also consider the effect of reduction modulo a prime number. Specifically, in the case in which $\mathbb{K}$ is a finite field of characteristic, denoted as $\mathrm{char}(\mathbb{K})$, equal to a prime number p, and cardinality $\mathrm{card}(\mathbb{K}) = p^t$, where $t \in \mathbb{N}$, one can refer to a good or bad reduction of $\mathcal{W}$ modulo p.

Theorem 3.1 (Good reduction at p)**.** *Let $\mathcal{W}$ be an elliptic curve in Weierstrass form over a field $\mathbb{K}$. The curve $\mathcal{W}$ has a good reduction modulo a prime $p > 0$ if $\Delta_{\mathcal{W}} \not\equiv 0 \pmod p$ (that is, if the reduced curve is non-singular).*

Conversely, if $\Delta_{\mathcal{W}} \equiv 0 \pmod p$, then $\mathcal{W}$ has a bad reduction modulo p. Furthermore, one may reduce $\mathcal{W}$ to one of its normal forms by eliminating its terms xy, y, and x^2 through simple transformations of x and y. Specifically, if the characteristic of the field $\mathbb{K}$ is different from 2, then one can reduce the generalized Weierstrass form through appropriate substitutions of the variables x and y, in order to eliminate the terms xy and y. Additionally, if $\mathrm{char}(\mathbb{K})$ is also different from 3, then one can perform another substitution in order to eliminate the term x^2.

Theorem 3.2 (Short elliptic curve in Weierstrass form)**.** *Let $\mathcal{W}$ be an elliptic curve in generalized Weierstrass form over a field $\mathbb{K}$ of characteristic different from 2 and from 3. It is possible to reduce $\mathcal{W}$ to the short Weierstrass form $\overline{\mathcal{W}}$, defined by the*

equation $y^2 = x^3 + ax + b$*, by applying the following transformation map:* $(x, y) \mapsto \left(f(x) - \frac{h^2(x)}{4}, y - \frac{h(x)}{2}\right)$*, where* $f(x) = x^3 + a_2x^2 + a_4x + a_6$*, and* $h(x) = a_1x + a_3$*.*

Remark 3.4. *The curve* $\overline{\mathcal{W}}$ *is non-singular if* $\Delta_{\overline{\mathcal{W}}} = -16(4a^3 + 27b^2) \neq 0$ *over* $\mathbb{K}$*.*

In the following, we will discuss the reduced forms of elliptic curves in generalized Weierstrass form $\mathcal{W}$ over a field $\mathbb{K}$ when the characteristic of the field is equal to 2 or 3. These cases require specific reductions depending on the j-invariant of the elliptic curve. Specifically, if $\mathrm{char}(\mathbb{K}) = 2$, then it is possible to reduce the generalized Weierstrass form to one of the following forms (cf. [44, 73]):

$$\begin{aligned} y^2 + a_3'y &= x^3 + a_4'x + a_6' && \text{if } j_{\mathcal{W}} = 0, \\ y^2 + a_1''xy &= x^3 + a_2''x^2 + a_6'' && \text{if } j_{\mathcal{W}} \neq 0. \end{aligned}$$

The first form is also referred as a supersingular (see definition 3.7) elliptic curve in Weierstrass form over $\mathbb{F}_2$, while the second form is a non-supersingular elliptic curve in Weierstrass form. On the other hand, if $\mathrm{char}(\mathbb{K}) = 3$, then it is possible to reduce the generalized Weierstrass form to one of the following forms (cf. [44, 73]):

$$\begin{aligned} y^2 &= x^3 + a_4'x + a_6' && \text{if } j_{\mathcal{W}} = 0, \\ y^2 &= x^3 + a_2''x^2 + a_6'' && \text{if } j_{\mathcal{W}} \neq 0. \end{aligned}$$

Definition 3.7 (Supersingular elliptic curve)**.** *Let* $\mathcal{W}$ *be an elliptic curve in Weierstrass form over* $\mathbb{K}$ *such that* $\mathrm{char}(\mathbb{K}) = p$*, where* p *is a prime number. The curve* $\mathcal{W}$ *is considered to be supersingular if there are no elements in* $\mathcal{J}(\mathcal{W})$ *that have order (see definition 3.8) equal to* $\mathrm{card}(\mathbb{K})$*.*

Given that it is always possible to reduce an elliptic curve in generalized Weierstrass form to one of its reduced forms, depending on the characteristic of the field and the j-invariant of the curve, it follows that it is always possible to consider the reduced form directly.

3.1.1 Group law

The group law of an elliptic curve in Weierstrass form $\mathcal{W}$, known as the chord-and-tangent rule, makes $\mathcal{W}$ an abelian group, with the point at infinity Ω as the identity element, since it is commutative, associative and has an inverse for each point.

As illustrated in fig. 3.1, let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two points belonging to $\mathcal{W}(\mathbb{K})$. The chord-and-tangent rule states that:

$$\begin{aligned} \ominus P_1 &= (x_1, -y_1 - (a_1x_1 + a_3)), \\ P_1 \ominus P_2 &= P_1 \oplus (\ominus P_2), \\ P_1 \oplus P_2 &= \begin{cases} \Omega & \text{if } P_1 = \ominus P_2 \\ (x_3, y_3) & \text{if } P_1 \neq \ominus P_2 \end{cases}, \end{aligned} \tag{3.2}$$

where

$$\begin{aligned} x_3 &= \lambda^2 + a_1\lambda - a_2 - x_1 - x_2, \\ y_3 &= (x_1 - x_3)\lambda - y_1 - a_1x_3 - a_3, \end{aligned} \tag{3.3}$$

and λ is the slope of the line between P_1 and P_2, defined as

$$\lambda = \begin{cases} \dfrac{y_1 - y_2}{x_1 - x_2} & \text{if } P_1 \neq P_2, \\ \dfrac{3{x_1}^2 + 2a_2x_1 - a_1y_1 + a_4}{2y_1 + a_1x_1 + a_3} & \text{if } P_1 = P_2. \end{cases} \tag{3.4}$$

Specifically, in terms of divisors, when using the chord rule, one identifies $P_1 \oplus P_2$ through the two zero divisors $\operatorname{div}(r_{1,2})$ and $\operatorname{div}(l_3)$, where $r_{1,2}$ is the line passing through P_1 and P_2, and intersecting the curve $\mathcal{W}$ at a third point P_3, and l_3 is the line passing through P_3 and orthogonal to the x-axis (and thus also passing through the point Q, which is symmetric to P_3 with respect to the x-axis). In particular, on the projective plane, one has that

$$\begin{aligned} \operatorname{div}(r_{1,2}) &= P_1 + P_2 + P_3 - 3\Omega, \\ \operatorname{div}(l_3) &= P_3 + Q + \Omega - 3\Omega, \\ \operatorname{div}\left(\frac{r_{1,2}}{l_3}\right) &= (P_1 + P_2) - (Q + \Omega). \end{aligned} \tag{3.5}$$

Consequently, one has that $P_1 + P_2 \equiv Q + \Omega$, from which it follows that $(P_1 - \Omega) + (P_2 - \Omega) \equiv (Q - \Omega)$. Thus, in order to sum the two divisors $P_1 - \Omega$ and $P_2 - \Omega$ using the above rule, one has the following equality: $P_1 \oplus P_2 = Q = \ominus P_3$. Note that, in the case of the tangent rule, one has that $P_1 = P_2$, that is, $r_{1,2}$ is the tangent line at $P_1 = P_2$, and one writes $2 \otimes P_1$ instead of $P_1 \oplus P_1$.

In general, one writes $n \otimes P$ to represent the addition of the point P (or its opposite) with itself $|n|$ times, according to the group law of $\mathcal{W}$„ that is, $n \otimes P = \bigoplus_{i=1}^{n} P$ if $n > 0$, or $n \otimes P = \bigoplus_{i=1}^{-n} (\ominus P)$ if $n < 0$.

Definition 3.8 (Order of a point)**.** *Let $\mathcal{W}$ be an elliptic curve in Weierstrass form over a field $\mathbb{K}$, and let $P - \Omega$ be a divisor belonging to $\mathcal{J}(\mathcal{W})$. If there exists a minimum and finite $0 < n \in \mathbb{N}$ such that $n \otimes P = \Omega$, then n is the order of the divisor $(P - \Omega) \in \mathcal{J}(\mathcal{W})$ (or, equivalently, as stated in remark 3.2, the order of the point P), and one writes $\operatorname{ord}(P - \Omega) = \operatorname{ord}(P) = n$. However, if such n does not exist, then $P - \Omega$ has order equal to ∞.*

Remark 3.5. *Note that the only element in $\mathcal{J}(\mathcal{W})$ of order equal to 1 is the identity element $\Omega - \Omega$.*

Definition 3.9 (Order of an elliptic curve)**.** *Let $\mathcal{W}$ be an elliptic curve in Weierstrass form over a field $\mathbb{K}$. The order of $\mathcal{W}$ is equal to the number of $\mathbb{K}$-rational points belonging to $\mathcal{W}(\mathbb{K})$, including the point at infinity Ω.*

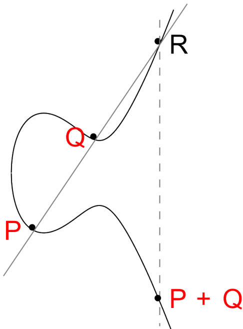

FIGURE 3.1: Sum of two points in an elliptic curve

3.1.2 Elliptic curves in Weierstrass form over the field of complex numbers

In this section, we will discuss some properties of elliptic curves when the field of consideration is the field of complex numbers. There exists a strong connection between elliptic curves in short Weierstrass form and the doubly periodic (with respect to a lattice Λ generated by $2\omega_1, 2\omega_2 \in \mathbb{C}$) Weierstrass $\wp$-function and its derivative $\wp'$, whose half-periods are ω_1 and ω_2. More precisely, the Weierstrass $\wp$-function is related to the elliptic integral, that is, one has that:

$$z = \int_{\wp(z)}^{\infty} \frac{1}{\sqrt{4t^2 - g_2 t - g_3}} dt. \tag{3.6}$$

Given the two functions $\wp$ and $\wp'$, one has the following equality:

$$\left(\frac{1}{2}\wp'(z)\right)^2 = \wp^3(z) - \frac{g_2}{4}\wp(z) - \frac{g_3}{4},$$

where $g_2, g_3 \in \mathbb{C}$ are constants known as elliptic invariants.

One notes that, with

$$x = \wp(z), \quad y = \frac{1}{2}\wp'(z),$$
$$a = -\frac{g_2}{4}, \quad b = -\frac{g_3}{4},$$

the point (x, y) fulfills the equation $y^2 = x^3 + ax + b$, which is of an elliptic curve in short Weierstrass form. In fact, the Weierstrass $\wp$-function and its derivative determine all the points of an elliptic curve in short Weierstrass form $\overline{\mathcal{W}}$ through the following map:

$$\begin{aligned} \mathrm{Exp}\colon \mathbb{C} &\longrightarrow \overline{\mathcal{W}}(\mathbb{C}) \\ z &\longmapsto \left[1 : \wp(z) : \frac{1}{2}\wp'(z)\right], \end{aligned}$$

which describes how to embed the complex torus $T = \mathbb{C}/\Lambda$ into the complex projective plane. Specifically, the map Exp defines a group isomorphism between the complex torus T and the group defined by the chord-and-tangent rule for $\overline{\mathcal{W}}$. Additionally, the map Exp defines an exponential map, that is, $\mathrm{Exp}(z_1 + z_2) = \mathrm{Exp}(z_1) \oplus \mathrm{Exp}(z_2)$, where $(\oplus)$ is the sum by using the chord-and-tangent rule.

Note that if $\gamma \in \Lambda$, then $\lim_{z\to\gamma} \mathrm{Exp}(z)$ is equal to $\Omega = [0:0:1]$, that is, the point at infinity of $\overline{\mathcal{W}}$.

The Weierstrass $\wp$-function (and its derivative) can be expressed [1, 3] through a Laurent series in a neighborhood of zero. Specifically, one has:

$$\begin{aligned} \wp(z) &= \frac{1}{z^2} + \sum_{k=2}^{\infty} c_k z^{2k-2}, \\ \wp'(z) &= -\frac{2}{z^3} + \sum_{k=2}^{\infty} (2k-2) c_k z^{2k-3}, \end{aligned} \tag{3.7}$$

where $c_2 = \frac{g_2}{20}$, $c_3 = \frac{g_3}{28}$ and $c_k = \frac{3}{(2k+1)(k-3)} \sum_{m=2}^{k-2} c_m c_{k-m}$.

Remark 3.6. *Note that while the above results were presented over $\mathbb{C}$, they can also be generalized over any suitable field $\mathbb{K}$ by taking into account a neighborhood of zero in order to ensure the convergence for the series expansion of $\wp(z)$ and $\wp'(z)$. In this case, the elements g_2, g_3, and z would belong to the field $\mathbb{K}$.*

Remark 3.7. *The map* Exp *is also known as the Weierstrass uniformizing map, and can also be expressed as follows:*

$$\begin{aligned} \mathrm{Exp}\colon \mathbb{C} &\longrightarrow \mathbf{P}^2(\mathbb{C}) \\ z &\longmapsto \left[1 : \wp(z) : \frac{1}{2}\wp'(z)\right]. \end{aligned} \tag{3.8}$$

3.1.3 Elliptic curves in Weierstrass form over the field of rational numbers

An elliptic curve defined over the field of rational numbers has some particular features. For instance, the Mordell-Weil theorem states that $\mathcal{W}(\mathbb{Q})$ is a finitely generated abelian group and, in particular, is a finite direct sum of r copies of $\mathbb{Z}$ and finite cyclic groups. Specifically, one can write $\mathcal{W}(\mathbb{Q})$ as $\mathcal{W}_{\mathrm{tors}}(\mathbb{Q}) \oplus \mathbb{Z}^r$, where $r \in \mathbb{N}$ is known as the (geometric) rank of $\mathcal{W}$, and $\mathcal{W}(\mathbb{Q})_{\mathrm{tors}}$ represents the torsion subgroup (see definition 3.10) of $\mathcal{W}(\mathbb{Q})$, that is, the set of $\mathbb{Q}$-rational points with finite order.

Thus, $\mathcal{W}(\mathbb{Q})$ is finitely generated by a point P of finite order and a set of r points $\{T_i\}$ of infinite order, meaning that if $Q \in \mathcal{W}(\mathbb{Q})$, then Q can be expressed as $Q = (k \otimes P) \oplus (\sum_{i=1}^{r} a_i \otimes T_i)$, where a_i are k are integer numbers different from ∞, P is a generator for $\mathcal{W}(\mathbb{Q})_{\mathrm{tors}}$, and $\{T_i\}$ is a set of r linearly independent points of infinite order belonging to the curve. Note that, over the field $\mathbb{Q}$, the points with infinite order have rational coordinates.

Definition 3.10 (Torsion subgroup of $\mathcal{W}$)**.** *Let $\mathcal{W}$ be an elliptic curve in Weierstrass form over a field $\mathbb{K}$. The torsion subgroup of $\mathcal{W}$, denoted as $\mathcal{W}(\mathbb{K})_{\mathrm{tors}}$, is defined as follows:*

$$\mathcal{W}(\mathbb{K})_{\mathrm{tors}} = \{Q \in \mathcal{W}(\mathbb{K})\colon \exists\ 0 < n \in \mathbb{N} \setminus \{\infty\},\ n \otimes Q = \Omega\}, \tag{3.9}$$

where $(\otimes)$ is the scalar multiplication defined on the group law.

It is well-known that the torsion subgroup over $\mathbb{Q}$ is isomorphic to either of the following 15 groups: $\mathbb{Z}/N\mathbb{Z}$, for $N = 1, \ldots, 10$ or $N = 12$, or $\mathbb{Z}/2\mathbb{Z} \times \mathbb{Z}/N\mathbb{Z}$ for $N = 1, 2, 3, 4$.

Remark 3.8. *One can define the set of points of a specific order $n \in \mathbb{N}$, known as the n-torsion subgroup. Specifically, one writes*

$$\mathcal{W}(\mathbb{K})[n] = \{Q \in \mathcal{W}(\mathbb{K}) \colon n \otimes Q = \Omega\}. \tag{3.10}$$

For instance, if $\mathcal{W}$ is a supersingular elliptic curve, then one has that $\mathcal{W}(\mathbb{K})[\mathrm{card}(\mathbb{K})] = \emptyset$.

Remark 3.9. *The rank of an elliptic curve $\mathcal{W}$ over $\mathbb{Q}$ can be challenging to compute. This is because it involves finding the size r of the smallest torsion-free generating set of points $\{T_1, T_2, \ldots, T_r\}$ such that, for $i = 1, \ldots, r$, $T_i \in \mathcal{W}(\mathbb{Q})$ and $T_i \neq \Omega$ [2, 73].*

Although the rank is challenging to compute, one simply computes the torsion subgroup $\mathcal{W}(\mathbb{Q})_{\mathrm{tors}}$ by using the Nagell–Lutz theorem.

Theorem 3.3 (Nagell–Lutz theorem)**.** *Let $\mathcal{W}$ be an elliptic curve in Weierstrass form over $\mathbb{Q}$. The point $(x, y) = P \in \mathcal{W}(\mathbb{Q})$ has finite order, i.e., $P \in \mathcal{W}(\mathbb{Q})_{\mathrm{tors}}$, if and only if either $x, y \in \mathbb{Z}$, or $\exists\, m, n \in \mathbb{Z}$ such that $\mathrm{ord}(P) = 2$ and $(x, y) = (\frac{m}{4}, \frac{n}{8})$.*

3.1.4 Elliptic curves in Weierstrass form over finite fields

In this part, we present some properties of elliptic curves in Weierstrass form $\mathcal{W}$ over a finite field $\mathbb{K}$ of characteristic $\mathrm{char}(\mathbb{K}) = p$ and $\mathrm{card}(\mathbb{K}) = p^t$, where $t \in \mathbb{N}$ and p is a prime number. For further information, we address the reader to [73].

Elliptic curves over finite fields have an elegant and simple structure to study. One of the most challenging tasks when studying elliptic curves over a general field $\mathbb{K}$ is counting the number of their rational points. However, through Hasse's theorem, one knows that there is a specific bound for the order of the group $\mathcal{J}(\mathcal{W})$.

Theorem 3.4 (Hasse's theorem)**.** *Let $\mathcal{W}$ be an elliptic curve in Weierstrass form over a finite field $\mathbb{K}$ of characteristic $\mathrm{char}(\mathbb{K}) = p$ and $\mathrm{card}(\mathbb{K}) = q = p^t$, where $t \in \mathbb{N}$ and p is a prime number. Hasse's bound for $\mathcal{W}$ is*

$$|N - (q - 1)| \leq 2\sqrt{q}, \tag{3.11}$$

where $N \in \mathbb{N}$ is the order of $\mathcal{J}(\mathcal{W})$.

Another interesting fact is that $\mathcal{J}(\mathbb{K})$ forms a finite abelian group, which is always cyclic or the product of two cyclic groups. This result, along with Hasse's bound, has enabled the development of efficient algorithms for counting the number of points on these curves such as the Baby-step giant-step, Schoof's algorithm, and Schoof–Elkies–Atkin's algorithm (an improvement of Schoof's algorithm). For instance, Schoof's algorithm uses the division polynomials [16] of an elliptic curve to count the elements belonging to the Jacobian. Additionally, Cantor in [19] generalized the concept of division polynomials to hyperelliptic curves.

The order of the Jacobian of an elliptic curve is a significant security parameter in the context of cryptography, as the higher this number, the greater the difficulty of breaking cryptographic protocols that use elliptic curves (see chapter 5 for further details).

3.2 Montgomery curves

Montgomery curves $\mathcal{M}$ are particular forms of elliptic curves, different from those in Weierstrass form $\mathcal{W}$. They were introduced by P. Montgomery in 1987 in the work [61], and are widely used in various cryptographic applications.

Definition 3.11 (Montgomery curve)**.** *A Montgomery curve $\mathcal{M}$ is an elliptic curve over a field $\mathbb{K}$, with $\mathrm{char}(\mathbb{K}) \neq 2$, defined by the equation*

$$By^2 = x^3 + Ax^2 + x, \tag{3.12}$$

where $A, B \in \mathbb{K}$ and $B(A^2 - 4) \neq 0$.

The use of Montgomery curves in cryptographic methods is due to their ability to optimize the addition and doubling operations of the group law in comparison to those of elliptic curves in Weierstrass form. Specifically, one can rewrite the affine coordinates of a point $P = (x, y)$ as (partial) projective coordinates $P = (X, Z)$ without keeping track of the Y coordinate. This is possible because the Y coordinate can be easily determined from X and Z by solving a square root in the field $\mathbb{K}$. While the classical group law that uses affine coordinates can always be employed, operating with the projective coordinates X and Z leads to a significant reduction in the number of computations. In fact, the group law of these curves is currently one of the most efficient among all other elliptic curve forms.

A prominent example of their use is the end-to-end encryption adopted by WhatsApp, which utilizes the well-known Montgomery curve known as Curve25519.

3.3 Edwards curves

The Edwards curves $\mathcal{E}$ are a family of non-smooth curves introduced by H. Edwards in 2007 [28]. Later, D.J. Bernstein and T. Lange generalized them [8, 9] and provided an efficient implementation in order to apply these curves for cryptographic applications.

Like the Montgomery curves, the Edwards curves have several advantages compared to the classic elliptic curves in Weierstrass form. Specifically, they allow for the optimization of the addition and doubling operations of the group law, leading to a significant reduction in computational complexity. Furthermore, one of the main advantages of Edwards curves is their high level of security. They have been shown to be resistant to certain types of side-channel attacks, making them a popular choice for secure communication and encryption. Due to their high level of security and efficiency, Edwards curves have been adopted in several cryptographic systems. For instance, they are used in the Ed25519 signature system, the X25519 Diffie-Hellman key exchange, and the EdDSA. In particular, they are also used in OpenSSH, an open-source implementation of the SSH protocol.

Definition 3.12 (Edwards curve)**.** *An Edwards curve $\mathcal{E}$ is a non-smooth curve over the field $\mathbb{K}$ defined by the equation*

$$x^2 + y^2 = c^2\left(1 + dx^2y^2\right), \tag{3.13}$$

where $c, d \in \mathbb{K}$ and $cd(1 - c^4d) \neq 0$.

As previously stated, these curves are non-smooth, meaning that there are singular points in $\mathcal{E}$. Specifically, Edwards curves, unlike those in Weierstrass form, has two

points at infinity: $\Omega_1 = [0:1:0]$ on the x-axis and $\Omega_2 = [0:0:1]$ on the y-axis, which are the two ordinary singular points of $\mathcal{E}$.

3.3.1 Group law

In order to compute the sum of two divisors $(P - O)$ and $(Q - O)$, where $P, Q \in \mathcal{E}(\mathbb{K})$ are two $\mathbb{K}$-rational affine points of $\mathcal{E}$, and $O = (0, c) \in \mathcal{E}(\mathbb{K})$ is taken as the neutral element of the group, one has to consider the group of divisor classes $\mathrm{Div}^0(\mathcal{E})/\mathrm{Princ}(\mathcal{E})$, modulo the subgroup of principal divisors of $\mathcal{E}$.

Specifically, one has to consider the three divisors $\mathrm{div}(\kappa)$, $\mathrm{div}(l_R)$, and $\mathrm{div}(X)$, where κ is an hyperbola, l_R is a line, and $X = 0$ is the equation of the x-axis in the projective plane. In particular, κ is the unique hyperbola, intersecting $\mathcal{E}$ in a third point $R = (x_R, y_R)$, such that $\mathrm{supp}(\mathrm{div}(\kappa))$ contains $O' = (0, -c)$, $2 \cdot \Omega_1$, and $2 \cdot \Omega_2$, and passing through the points P and Q. Moreover, l_R is the line passing through R and parallel to the x-axis (thus l_R intersects the curves at the point $S = (-x_R, y_R)$ as well). It follows that

$$\begin{aligned} \mathrm{div}(\kappa) &= (P + Q + R + O' + 2 \cdot \Omega_1 + 2 \cdot \Omega_2) - (4 \cdot \Omega_1 + 4 \cdot \Omega_2), \\ \mathrm{div}(l_R) &= (R + S + 2 \cdot \Omega_1) - (2 \cdot \Omega_1 + 2 \cdot \Omega_2), \\ \mathrm{div}(X) &= (O + O' + 2 \cdot \Omega_2) - (2 \cdot \Omega_1 + 2 \cdot \Omega_2), \end{aligned} \tag{3.14}$$

and

$$\mathrm{div}\left(\frac{\kappa}{l_R \cdot X}\right) = P + Q - S - O. \tag{3.15}$$

Therefore, one has that $(P - O) + (Q - O) \equiv (S - O)$, and consequently $P \oplus Q = S$. Since Edwards curves are symmetric with respect to the y-axis, the divisor $S - O$ is the opposite of the divisor $R - O$, and $S = \ominus R$.

In summary, as illustrated in fig. 3.2, in order to sum two not necessarily distinct points $P, Q \in \mathcal{E}(\mathbb{K})$, one has to:

- Determine the unique hyperbola defined by the equation $xy + Ax + By + B = 0$ (containing $O' = (0, -c)$, $2 \cdot \Omega_1$ and $2 \cdot \Omega_2$) that passes through P and Q;
- Compute the additional intersection point $R = (x_R, y_R)$ between the hyperbola and the curve $\mathcal{E}$;
- Consider the line l_R passing through R and parallel to the x-axis, which also passes through the point $S = (-x_R, y_R) = \ominus R$ as well.

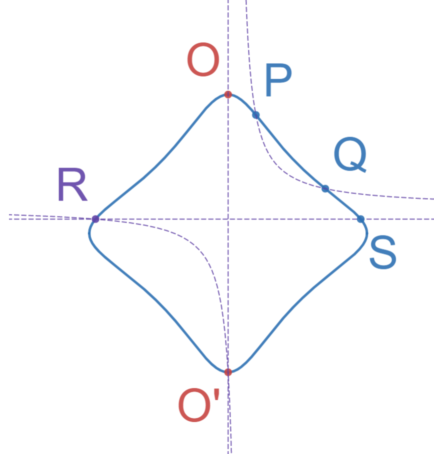

FIGURE 3.2: Sum of two points on an Edwards curve

Thus, one defines the following operations to sum or subtract the points $P_1, P_2 \in \mathcal{E}(\mathbb{K})$, where $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$:

$$\begin{aligned}
\ominus P_1 &= (-x_1, y_1),\\
n \otimes P_1 &= \bigoplus_{i=1}^{n} P_1 \qquad n \in \mathbb{Z}, n > 0,\\
n \otimes P_1 &= \bigoplus_{i=1}^{-n} (\ominus P_1) \qquad n \in \mathbb{Z}, n < 0,\\
P_1 \ominus P_2 &= P_1 \oplus (\ominus P_2),\\
P_1 \oplus P_2 &= \begin{cases} O & \text{if } P_1 = \ominus P_2\\ \left(\dfrac{x_1y_2 + x_2y_1}{c(1 + dx_1x_2y_1y_2)}, \dfrac{y_1y_2 - x_1x_2}{c(1 - dx_1x_2y_1y_2)}\right) & \text{if } P_1 \neq \ominus P_2 \end{cases}
\end{aligned} \tag{3.16}$$

Remark 3.10 (cf. [8])**.** *Note that $(P - O) + (Q - O) \equiv O - O$ if and only if $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ are such that $y_P = y_Q$ and $x_P = -x_Q$, meaning that Q is the point symmetric to P with respect to the y-axis.*

Remark 3.11. *Note that if the parameter d is a non-square, then the denominators in the addition and doubling formulas cannot vanish [8] and the affine points of the curve give in turn a subgroup of the entire group of divisor classes (cf. corollary 3.5).*

Remark 3.12. *In the curve $\mathcal{E}$, there are four noteworthy points: $O = (0, c)$, $O' = (0, -c)$, $H = (c, 0)$, $H' = (-c, 0)$. As already stated, O serves as the identity element of the group law, meaning that for any point $P = (a, b) \in \mathcal{E}(\mathbb{K})$, one has that $P \oplus O = P$. The other three points have the following properties:*

- $(a, b) \oplus O' = (-a, -b)$;
- $(a, b) \oplus H = (b, -a)$;
- $(a, b) \oplus H' = (-b, a)$.

Specifically, these four points form the cyclic group C_4, where O' has order 2, and $2 \otimes H = 2 \otimes H' = O'$.

Corollary 3.5. *The subset $\mathcal{J}^0(\mathcal{E})$ of zero degree divisors whose support contains only affine points is a subgroup of $\mathcal{J}(\mathcal{E})$.*

Proof. Let $(P - O), (Q - O) \in \mathcal{J}^0(\mathcal{E})$ be two divisors, where $P, Q \in \mathcal{E}(\mathbb{K})$. By remark 3.10 one has that $-(Q - O) \in \mathcal{J}^0(\mathcal{E})$, and by remark 3.11 one has that $(P - O) - (Q - O) \in \mathcal{J}^0(\mathcal{E})$. ■

In terms of the group of divisor classes, one finds either of the following reduced divisors in each divisor class.

Theorem 3.6 (Jacobian of Edwards curves)**.** *Let $\mathcal{E}$ be an Edwards curve, and let $\mathcal{J}(\mathcal{E})$ be the Jacobian of $\mathcal{E}$. Every divisor $D \in \mathcal{J}(\mathcal{E})$ has one of the following canonical forms:*

1. $D \equiv P - O$,
2. $D \equiv (P - O) + (\Omega_1 - O)$,
3. $D \equiv (P - O) + (\Omega_2 - O)$,
4. $D \equiv (P - O) + (\Omega_1 - \Omega_2)$,

where $P \in \mathcal{E}(\mathbb{K})$ is an affine point. In particular, the divisors equivalent to $P - O$ forms the subgroup $\mathcal{J}^0(\mathcal{E})$ (see corollary 3.5) of index 4 in $\mathcal{J}(\mathcal{E})$.

Proof. In the first part of this proof, we show that every even multiple of Ω_1 and Ω_2 is equivalent to a multiple of $O' + O$ and $H + H'$, respectively. Specifically, we have that:

$$\operatorname{div}\left(\frac{X}{Z}\right) = O' + O - 2 \cdot \Omega_1,$$
$$\operatorname{div}\left(\frac{Y}{Z}\right) = H' + H - 2 \cdot \Omega_2,$$

which implies that $O' + O \equiv 2 \cdot \Omega_1$ and $H' + H \equiv 2 \cdot \Omega_2$.

Let $D = D_1 + D_2$ be an unreduced divisor of $\mathcal{J}(\mathcal{E})$, where $\operatorname{supp}(D_1)$ consists of only affine points, and $D_2 = t_1 \cdot \Omega_1 + t_2 \cdot \Omega_2$ with $t_1, t_2 \in \mathbb{Z}$. In this context, there are only three possible cases for the pair (t_1, t_2):

1. both t_1 and t_2 are even;
2. only one between t_1 and t_2 is even;
3. both t_1 and t_2 are odd.

Since $O' + O \equiv 2 \cdot \Omega_1$ and $H' + H \equiv 2 \cdot \Omega_2$, if both t_1 and t_2 are even, then we can cancel out $t_1 \cdot \Omega_1$ and $t_2 \cdot \Omega_2$ in order to reduce D to a divisor whose support consist of only affine points. Subsequently, we may use the group law in order to get the first canonical form of D, that is, $D \equiv P - O$.

In the following, we consider the case where only one of the two values, t_1 and t_2, is even. If this is the case, then we can cancel either $t_1 \cdot \Omega_1$ or $t_2 \cdot \Omega_2$, and we obtain either $(P - O) + (\Omega_1 - O)$ or $(P - O) + (\Omega_2 - O)$. Since the curve $\mathcal{E}$ has degree equal to 4, thus these forms are not equivalent as any principal divisor on the curve $\mathcal{E}$ will always have an even number of Ω_1 and Ω_2 in its support.

In the third case, where both t_1 and t_2 are odd, we can cancel an even number of Ω_1 and Ω_2 to obtain either $(P - O) + (\Omega_1 - \Omega_2)$ or $(P - O) + (\Omega_2 - \Omega_1)$. However, the latter form is equivalent to the former as $\Omega_1 - \Omega_2 \equiv (\Omega_2 - \Omega_1) + (O' + O) - (H' + H)$.

Finally, since

$$2 \cdot (\Omega_1 - O) = 2 \cdot \Omega_1 - 2 \cdot O \equiv (O' + O) - 2 \cdot O = O' - O \in \mathcal{J}^0(\mathcal{E}),$$
$$2 \cdot (\Omega_2 - O) = 2 \cdot \Omega_2 - 2 \cdot O \equiv (H' + H) - 2 \cdot O \in \mathcal{J}^0(\mathcal{E}),$$

thus the quotient group $\mathcal{J}(\mathcal{E})/\mathcal{J}^0(\mathcal{E})$ is isomorphic to $\mathbb{Z}/2\mathbb{Z} \oplus \mathbb{Z}/2\mathbb{Z}$. ∎

Remark 3.13. *As for remark 3.2, for a given Edwards curve $\mathcal{E}$, any non-zero divisor $P - O$ in $\mathcal{J}^0(\mathcal{E})$ can be associated with the affine point $P \in \mathcal{E}(\mathbb{K})$, and the zero divisor $O - O$ can be represented by O. As a result, one can interchangeably refer to either the Jacobian or the group of $\mathbb{K}$-rational points of this curve.*

3.3.2 Equivalence between Edwards curves and Elliptic curves in Weierstrass form

In this section, we show that under particular assumptions elliptic curves in Weierstrass form are equivalent to Edwards curves.

As noted in remark 3.12, the point $H = (c, 0)$ belonging to $\mathcal{E}$ has order equal to four. In the following proposition, we identify a point of order four belonging to an elliptic curve in a suitable Weierstrass form. This latter point will be used in the birational maps between these two curves in theorem 3.8.

Proposition 1. *Let $\tilde{\mathcal{W}}$ be an elliptic curve over a field $\mathbb{K}$ of characteristic different from 2, given in the Weierstrass form*

$$y^2 = x^3 + a'x^2 + b'x.$$

The curve $\tilde{\mathcal{W}}$ has a point $P = (x_1, y_1) \in \tilde{\mathcal{W}}(\mathbb{K})$ such that the divisor $P - \Omega \in \mathcal{J}(\tilde{\mathcal{W}})$ has order 4 and $2(P - \Omega) = (0, 0) - \Omega$, if and only if $x_1^2 = b'$ (and obviously $y_1^2 = x_1^3 + a'x_1^2 + b'x_1$). Furthermore, if one defines $d = 1 - \frac{4x_1^3}{y_1^2}$, then one has that $a' = 2x_1 \frac{1+d}{1-d}$.

Proof. In this proof, we make use of the group law of the elliptic curve $\tilde{\mathcal{W}}$ to compute the point (x_3, y_3) such that $(x_3, y_3) - \Omega = 2 \cdot ((x_1, y_1) - \Omega)$. Specifically, we have that

$$\begin{cases} \lambda &= \frac{3x_1^2 + 2a'x_1 + b'}{2y_1}, \\ x_3 &= \lambda^2 - a' - 2x_1, \\ y_3 &= (x_1 - x_3)\lambda - y_1, \end{cases}$$

where λ is the slope of the tangent line at (x_1, y_1). The solution of the previous system of equations is the point $(x_3, y_3) = (0, 0)$, which always belongs to the curve $\tilde{\mathcal{W}}$, and therefore $(x_1, y_1) - \Omega$ is a divisor of order 4. ∎

Theorem 3.7 (cf. [8])**.** *Let $\mathbb{K}$ be a field of characteristic different from 2. Let $\mathcal{E}$ be a (non-smooth) Edwards curve defined over $\mathbb{K}$ by the equation*

$$\hat{x}^2 + \hat{y}^2 = 1 + d\hat{x}^2\hat{y}^2,$$

where $d(d-1) \neq 0$. For any $x_1 \in \mathbb{K}$, if one chooses $y_1 \in \mathbb{K}$ such that $y_1^2 = \frac{4x_1^3}{d-1}$, $a' = 2x_1 \frac{1+d}{1-d}$, and $b' = x_1^2$, then one has a rational map between the curve $\mathcal{E}$ and the

elliptic curve defined over $\mathbb{K}$, and given in the Weierstrass form $\tilde{\mathcal{W}}$, by the equation

$$y^2 = x^3 + a'x^2 + b'x.$$

Specifically, the point (x_1, y_1) belongs to $\tilde{\mathcal{W}}(\mathbb{K})$, and the rational map between $\mathcal{E}$ and $\tilde{\mathcal{W}}$ is given by

$$\begin{aligned} \alpha\colon \mathcal{E}(\mathbb{K}) &\longrightarrow \tilde{\mathcal{W}}(\mathbb{K}) \\ (\hat{x}, \hat{y}) &\longmapsto (x, y) = \left(x_1 \frac{1+\hat{y}}{1-\hat{y}}, y_1 \frac{(1+\hat{y})}{\hat{x}(1-\hat{y})} \right). \end{aligned} \tag{3.17a}$$

Conversely, one has the following result.

Theorem 3.8 (cf. [8]). *Let $\mathbb{K}$ be a field of characteristic different from 2. Let $\tilde{\mathcal{W}}$ be an elliptic curve defined over $\mathbb{K}$ in the Weierstrass form*

$$y^2 = x^3 + a'x^2 + b'x.$$

If the point $P = (x_1, y_1) \in \tilde{\mathcal{W}}(\mathbb{K})$ is such that $2 \cdot (P - \Omega) = (0, 0) - \Omega$, then, putting $d = 1 - \frac{4x_1^3}{y_1^2}$, there is a rational map over $\mathbb{K}$ between $\tilde{\mathcal{W}}$ and the (non-smooth) Edwards curve $\mathcal{E}$ defined by the equation

$$\hat{x}^2 + \hat{y}^2 = 1 + d\hat{x}^2\hat{y}^2.$$

The rational map between $\tilde{\mathcal{W}}$ and $\mathcal{E}$, which turns out to be the inverse of the map α in theorem 3.7, is given by

$$\begin{aligned} \beta\colon \tilde{\mathcal{W}}(\mathbb{K}) &\longrightarrow \mathcal{E}(\mathbb{K}) \\ (x, y) &\longmapsto (\hat{x}, \hat{y}) = \left(\frac{y_1 x}{x_1 y}, \frac{x - x_1}{x + x_1} \right) \end{aligned} \tag{3.18a}$$

Therefore, we arrive at the following definition that encompasses the above statements and outlines the assumptions under which there is a relationship between Edwards curves and elliptic curves in Weierstrass form.

Definition 3.13 (cf. [8]). *Let $\mathbb{K}$ be a field of characteristic different from 2. Let $\mathcal{E}$ be an Edwards curve defined over $\mathbb{K}$ by the equation*

$$\hat{x}^2 + \hat{y}^2 = 1 + d\hat{x}^2\hat{y}^2,$$

where $d(d-1) \neq 0$. Let $0 \neq x_1 \in \mathbb{K}$ be such that x_1 and $(1-d)$ are either both non-square or both square in $\mathbb{K}$, and let $y_1 \in \mathbb{K}$ such that $y_1^2 = \frac{4x_1^3}{1-d}$. Furthermore, let $\tilde{\mathcal{W}} = \tilde{\mathcal{W}}_{d,x_1}$ be the elliptic curve in Weierstrass form defined over $\mathbb{K}$ by the equation

$$y^2 = x^3 + a'x^2 + b'x,$$

where $a' = 2x_1 \frac{1+d}{1-d}$ and $b' = x_1^2$. The two rational maps, denoted α and β, between $\mathcal{E}$ and $\tilde{\mathcal{W}}$ are defined as follows:

$$\alpha \colon \mathcal{E}(\mathbb{K}) \longrightarrow \tilde{\mathcal{W}}(\mathbb{K}) \tag{3.19a}$$
$$(\hat{x}, \hat{y}) \longmapsto (x, y) = \left(x_1 \frac{1+\hat{y}}{1-\hat{y}}, y_1 \frac{(1+\hat{y})}{\hat{x}(1-\hat{y})} \right),$$
$$\alpha^{-1} = \beta \colon \tilde{\mathcal{W}}(\mathbb{K}) \longrightarrow \mathcal{E}(\mathbb{K}) \tag{3.19b}$$
$$(x, y) \longmapsto (\hat{x}, \hat{y}) = \left(\frac{y_1 x}{x_1 y}, \frac{x - x_1}{x + x_1} \right).$$

There maps make the two curves $\tilde{\mathcal{W}}$ and $\mathcal{E}$ birationally equivalent. Additionally, the definitions of α and β are extended by putting

$$\begin{aligned} \alpha((0,1)) &= \Omega, & \beta(\Omega) &= (0,1), \\ \alpha((0,-1)) &= (0,0), & \beta((0,0)) &= (0,-1); \end{aligned} \tag{3.20}$$

and (possibly)

$$\begin{aligned} \beta((t_1,0)) = \beta((t_2,0)) &= \Omega_1, \\ \beta((-x_1, \pm s_1)) &= \Omega_2, \end{aligned} \tag{3.21}$$

where $(t_1,0), (t_2,0), (-x_1, \pm s_1) \in \tilde{\mathcal{W}}(\mathbb{K})$, with $t_1, t_2 \neq 0$ and $s_1^2 = (-x_1)^3 + a'(-x_1)^2 + b'(-x_1)$.

Remark 3.14. *Note that, in definition 3.13, the affine point $P_4 = (x_1, y_1)$ belongs to $\tilde{\mathcal{W}}(\mathbb{K})$ and satisfies the condition that $P_4 - \Omega \in \mathcal{J}(\tilde{\mathcal{W}})$ is a divisor of order 4.*

Remark 3.15. *In light of the fact that there are two points mapped by β to Ω_1 and two points to Ω_2, it is not possible to coherently define $\alpha(\Omega_1)$ and $\alpha(\Omega_2)$. As a result, the maps α and β define a birational equivalence between the two curves. In addition, it is noteworthy that the curve $\tilde{\mathcal{W}}$ is actually a smooth projective resolution of the non-smooth curve $\mathcal{E}$.*

Remark 3.16. *The values of Ω_1 and Ω_2 can be directly obtained as the projective images of $\beta((t_1,0))$ and $\beta((t_2,0))$, and $\beta((-x_1, \pm s_1))$, respectively, that is, by using the homogeneous coordinates for α and β. As for the value of $\beta(\Omega)$, if $\mathbb{K} = \mathbb{C}$ and $P = \left(\wp(z) + \frac{a'}{3}, \frac{1}{2}\wp'(z) \right) \in \tilde{\mathcal{W}}(\mathbb{K})$, where $\wp(z)$ and $\wp'(z)$ are the Weierstrass elliptic functions in subsec. 3.1.2 then,*

$$\lim_{z \to 0} \beta(P) = \lim_{z \to 0} \left(\frac{2y_1}{3x_1} \frac{(3\wp(z) + a')}{\wp'(z)}, \frac{3\wp(z) + a' - 3x_1}{3\wp(z) + a' + 3x_1} \right) = (0,1) = O,$$

as $\frac{3\wp(z)+a'}{3\wp'(z)} = o(z)$, that is, β is continuous at $P = \Omega$.

Remark 3.17. *Since the map β in (3.19b) transforms a line through two points P and Q belonging to $\tilde{\mathcal{W}}(\mathbb{K})$ onto the hyperbola passing through $\beta(P)$, $\beta(Q)$, O', $2 \cdot \Omega_1$ and $2 \cdot \Omega_2$, and maps vertical lines onto horizontal lines, then β induces a group homomorphism of the corresponding divisor classes group.*

Remark 3.18. *Since the coefficient d depends on a point $P_4 \in \tilde{\mathcal{W}}(\mathbb{K})$ such that $(P_4 - \Omega) \in \mathcal{J}(\tilde{\mathcal{W}})$ is a divisor of order 4, it may be necessary to extend the field of definition of $\mathcal{E}$ in order to determine the Weierstrass elliptic curve $\tilde{\mathcal{W}}$ that projects onto $\mathcal{E}$.*

In the following, we emphasize the meaning of taking d a non-square in a field $\mathbb{K}$.

Theorem 3.9 (Isomorphism between $\mathcal{J}(\tilde{\mathcal{W}})$ and $\mathcal{J}^0(\mathcal{E})$)**.** *Let both $\mathcal{E}$ and $\tilde{\mathcal{W}}$ be defined as in definition 3.13. If d is not a square in the field $\mathbb{K}$, then an isomorphism over $\mathbb{K}$ exists between the group $\mathcal{J}(\tilde{\mathcal{W}})$ and the subgroup $\mathcal{J}^0(\mathcal{E})$ defined in theorem 3.6.*

Proof. In the following, we will demonstrate that if d is a non-square in the field $\mathbb{K}$, then the rational map β defined in (3.19b) constitutes a biregular map between the Weierstrass elliptic curve $\tilde{\mathcal{W}}$, given by the equation $y^2 = x^3 + a'x^2 + b'x$, and the subset of the Edwards curve $\mathcal{E}$ consisting of its affine points. By using the parameters x_1, y_1, a', and b' as defined in definition 3.13, we show that there is no point in $\tilde{\mathcal{W}}$ with abscissa equal to $-x_1$ and that $(0, 0)$ is the only point in $\tilde{\mathcal{W}}(\mathbb{K})$ with ordinate $y = 0$. The absence of a point in $\tilde{\mathcal{W}}$ with abscissa equal to $-x_1$ can be derived from the fact that the intersection of the line $x = -x_1$ and the curve $\tilde{\mathcal{W}}$ results in the following:

$$\begin{aligned} -x_1^3 + a'x_1^2 - b'x_1 = x_1^2(a' - 2x_1) = x_1^2\left(2x_1\frac{1+d}{1-d} - 2x_1\right) = \\ = x_1^2\left(2x_1\frac{2d}{1-d}\right) = dy_1^2. \end{aligned}$$

Since d is a non-square, dy_1^2 is also a non-square, and thus, there is no point in $\tilde{\mathcal{W}}$ with abscissa equal to $-x_1$.

The latter assertion follows from the fact that the line $y = 0$ intersects the curve $\tilde{\mathcal{W}}$ only at $x = 0$ over $\mathbb{K}$. This can be proven by examining the equation

$$x^3 + a'x^2 + b'x = x(x^2 + a'x + b') = 0.$$

Specifically, one has that

$$\begin{aligned} \Delta(x^2 + a'x + b') &= a'^2 - 4b' = a'^2 - 4x_1^2 = \\ &= 4x_1^2\left(\frac{1+d}{1-d}\right)^2 - 4x_1^2 = \\ &= 4x_1^2\frac{4d}{(1-d)^2}. \end{aligned}$$

Since d is a non-square, the discriminant is not a square in $\mathbb{K}$, which implies that $(0, 0)$ is the only point in $\tilde{\mathcal{W}}(\mathbb{K})$ with ordinate $y = 0$. ■

From this point forward, we restrict our attention to the scenario outlined in theorem 3.9. In this case, we can identify elements $P - O$ of $\mathcal{J}^0(\mathcal{E})$ with the point P so that O is the neutral element of the group, as previously mentioned in remark 3.13.

3.3.3 Twisted Edwards curves

The twisted Edwards curves were introduced by Bernstein and Lange in an attempt to generalize the original Edwards curves (that they already extend by introducing the parameter c, as stated in definition 3.12).

Definition 3.14 (Twisted Edwards curve)**.** *Let $\mathbb{K}$ be a field of characteristic different from 2. A twisted Edwards curve $\mathcal{E}_{a,d}$ over $\mathbb{K}$ is an Edwards curve defined by equation*

$$ax^2 + y^2 = 1 + dx^2y^2, \tag{3.22}$$

where $a, d \in \mathbb{K} \setminus \{0\}$ and $a \neq d$.

Remark 3.19. *Note that $\mathcal{E}_{1,d}$ is the Edwards curve defined in definition 3.12 with $c = 1$.*

The twisted Edwards curves have several advantages over the classic Edwards curves, including faster point addition and scalar multiplication algorithms. As a result, Twisted Edwards curves have found several applications in cryptography, including their use as the core of the Edwards-curve Digital Signature Algorithm (EdDSA), a high-performance digital signature scheme. In addition to digital signatures, twisted Edwards curves have also been used for key exchange, identity-based encryption, and homomorphic encryption. Many articles and scientific papers have been published about twisted Edwards curves, including works by Bernstein et al. [14], which prove that twisted Edwards curves are birationally equivalent to the Montgomery curves. This equivalence has important implications for the security of cryptographic systems that use twisted Edwards curves, as well as for the efficiency and performance of these systems.

Overall, twisted Edwards curves are a promising development in the field of elliptic curve cryptography, offering improved performance and security compared to classic Edwards curves and elliptic curves. They have already been widely adopted by several cryptographic communities and are likely to become even more popular in the future.

4 Hyperelliptic curves

Elliptic curves are one of the most important objects in modern cryptography, since they have numerous applications in the areas of secure communication and data protection. These curves have been extensively studied since the late 19th century, but have only recently gained popularity due to their important role in public-key cryptography. In recent years, the study of elliptic curves has been extended to a new type of curves known as hyperelliptic curves. One of the key advantages of hyperelliptic curves over elliptic curves is that they are more secure against certain types of attacks. For example, they are less vulnerable to attacks based on the discrete logarithm problem, since they have a higher number of points on the curve. This makes them ideal for use in public-key cryptography, where security is of the utmost importance. Another important application of hyperelliptic curves is in the area of digital signatures, which are used to verify the authenticity of electronic documents, and are an essential component of secure communication.

Definition 4.1. *A hyperelliptic curve $\mathcal{H}$ is an algebraic curve of genus $g \geq 1$ defined over field $\mathbb{K}$ by the equation*

$$y^2 + h(x)y = f(x), \tag{4.1}$$

where $f(x)$ is a monic polynomial such that $\deg(f(x)) = 2g+1$ or $\deg(f(x)) = 2g+2$, and $\deg(h(x)) \leq g+1$.

Remark 4.1. *If the ground field $\mathbb{K}$ of $\mathcal{H}$ is such that $\operatorname{char}(\mathbb{K}) \neq 2$, then the transformation $(x, y) \mapsto \left(f(x) - \frac{h^2(x)}{4}, y - \frac{h(x)}{2}\right)$ can be applied in order to have set $h(x) = 0$.*

It is noteworthy that, on one hand, when $\deg(f(x)) = 2g+1$, one has that $\deg(h(x)) \leq g$ and these curves are known as imaginary hyperelliptic curves. On the other hand, when $\deg(f(x)) = 2g+2$, these curves are known as real hyperelliptic curves which have two points at infinity, commonly denoted as Ω_1 and Ω_2. Throughout this thesis, we only consider the imaginary hyperelliptic curves.

Remark 4.2. *In accordance with the mathematical definition of an imaginary hyperelliptic curve, it is worth mentioning that such curves of genus $g = 1$ are simply referred to as ordinary elliptic curves, previously defined in definition 3.5.*

Definition 4.2 (Jacobian of an imaginary hyperelliptic curve)**.** *Let $\mathcal{H}$ be an (imaginary) hyperelliptic curve of genus g over an algebraically closed field $\overline{\mathbb{K}}$. The Jacobian $\mathcal{J}(\mathcal{H})$ is the group of all the principal divisors of $\mathcal{H}$ such that $D \in \mathcal{J}(\mathcal{H})$ is linearly equivalent to a reduced divisor of the form $\sum_{i=1}^{n} P_i - n \cdot \Omega$, where $0 \leq n \leq g$, and for all $i = 1, \ldots, n$, one has that $P_i \neq \Omega$, and $P_i + P_j - 2 \cdot \Omega \not\equiv \Omega - \Omega$ for any $i \neq j$.*

Remark 4.3. *The divisor $D = \Omega - \Omega$, or simply $D = \Omega$, denotes the identity element of the group.*

Remark 4.4. *Note that, in the case where $g = 1$, there are only two possible reduced divisors: either $D = \Omega - \Omega$ or $D = P - \Omega$, where P is an affine point belonging to the curve. These two principal divisors are denoted as $D = \Omega$ and $D = P$*

respectively, and they form the group of principal divisors for the elliptic curves, which was previously discussed in remark 3.2.

To end this introduction, we present the well-known Hasse's Theorem for hyperelliptic curves over a finite field.

Theorem 4.1 (Hasse's theorem for hyperelliptic curves)**.** *Let $\mathcal{H}$ be a hyperelliptic curve of genus g over a finite field $\mathbb{K}$ with characteristic p and cardinality $q = p^t$, where $t \in \mathbb{N}$ and p is a prime number. The Hasse bound for $\mathcal{H}$ is given by*

$$(\sqrt{q} - 1)^{2g} \leq N \leq (\sqrt{q} + 1)^{2g}, \tag{4.2}$$

where $N \in \mathbb{N}$ is the cardinality of $\mathcal{J}(\mathcal{H})$.

4.1 Group law

In order to define the general group law [22] for an (imaginary) hyperelliptic curve, we use the Mumford representation [63] of a reduced divisor belonging to the Jacobian of a hyperelliptic curve of genus $g \geq 1$. For a more in-depth analysis of this group law, the reader is referred to [17], which uses the Weierstrass $\wp$-function. We remark that the study of this group law has been the focus of numerous investigations with the goal of speeding it up for cryptographic purposes, particularly when $g = 2$ as seen in [41, 53, 54].

Theorem 4.2 (Mumford representation)**.** *Let $\mathcal{H}$ be a hyperelliptic curve of genus g over an algebraically closed field $\overline{\mathbb{K}}$, defined by the equation $y^2 + h(x)y = f(x)$. Let $D \in \mathcal{J}(\mathcal{H})$ be a divisor linearly equivalent to $\sum_{i=1}^{n} P_i - n \cdot \Omega$, where $0 \leq n \leq g$, and for $i = 1, \ldots, n$, one has that $P_i = (x_i, y_i) \in \mathcal{H}\left(\overline{\mathbb{K}}\right)$, $P_i \neq \Omega$, and $P_i + P_j - 2 \cdot \Omega \not\equiv \Omega - \Omega$ for any $i \neq j$. Then, D has a Mumford representation as pair of polynomials $\operatorname{div}(u(x), v(x)) = (u(x), v(x))$ such that:*

- $\gcd(u(x), u'(x), v(x)) = 1$,
- $\deg(v(x)) < \deg(u(x)) \leq g$,
- $u(x) \mid v(x)^2 + v(x)h(x) - f(x)$,

where $u(x)$ is the monic polynomial equal to $\prod_{i=1}^{n}(x - x_i)$ and $v(x)$ is the interpolating polynomial such that $v(x_i) = y_i$.

Remark 4.5. *The divisor D, represented in Mumford representation as $(u(x) = 1, v(x) = 0)$, is the identity element of the group, i.e., the divisor $\Omega - \Omega$.*

Remark 4.6. *If $(x_i, y_i) \in \operatorname{supp}(D)$ has multiplicity $n_i > 1$, then the following conditions must be met:*

$$\left(\left(\frac{\delta}{\delta x}\right)^j \left(v^2(x) + v(x)h(x) - f(x)\right)\right)\Bigg|_{x=x_i} = 0 \qquad \textit{with } j = 0, \ldots, n_i - 1.$$

The Cantor-Koblitz algorithm for the sum of two divisors D_1 and D_2 was first introduced by Cantor in [18]. This algorithm, which uses the Mumford representation of a divisor, allows for the calculation of the unique reduced divisor D equivalent to $D_1 \oplus D_2$ in any hyperelliptic curve of genus $g \geq 1$. Originally, Cantor only considered the case in which $h(x)$ was equal to zero. However, this was later generalized by

Koblitz in [49] to include the binary case in which $h(x) \neq 0$. We present below the Cantor-Koblitz algorithm for the sum of two divisors.

Algorithm 1: Cantor-Koblitz algorithm

Input: $D_1 = (u_1(x), v_1(x))$, $D_2 = (u_2(x), v_2(x))$, $\mathcal{H} : y^2 + h(x)y = f(x)$, where $\deg(f(x)) = 2g + 1$

Output: $D = D_1 \oplus D_2 = (u(x), v(x))$

```
    // compute the extended GCD between u1(x) and u2(x)
 1  (d1, e1, e2) ⟵ Extended-GCD(u1(x), u2(x))          // d1 = e1u1(x) + e2u2(x)
    // compute the extended GCD between d1 and v1(x) + v2(x) + h(x)
 2  (d, c1, c2) ⟵ Extended-GCD(d1, v1(x) + v2(x) + h(x))
     // d = c1d1 + c2(v1(x) + v2(x) + h(x))
 3  s1 ⟵ c1e1
 4  s2 ⟵ c1e2
 5  s3 ⟵ c2
    // composition step
    // computation of the curve passing through the points in D1 and D2
 6  u(x) ⟵ u1(x)u2(x) / d^2
    // computation of the curve passing through the points in D1 and D2, and
       the curve H
 7  v(x) ⟵ (s1u1(x)v2(x)+s2u2(x)v1(x)+s3(v1(x)v2(x)+f(x))) / d  (mod u(x))
 8  repeat// reduction step
 9  |  u(x) ⟵ (f(x)−v(x)h(x)−v^2(x)) / u(x)
10  |  v(x) ⟵ −h(x) − v(x) (mod u(x))
11  until deg(u(x)) ≤ g
    // make u(x) monic dividing it by its leading coefficient
12  u(x) ⟵ u(x) / lc(u(x))
13  return (u(x), v(x))
```

The lines 6 and 7 comprise the component of the algorithm referred to as the composition phase, where one obtains the composed, unreduced, divisor whose support is the union of $\text{supp}(D_1)$ and $\text{supp}(D_2)$. On the other hand, the loop from line 8 to 11 constitutes the reduction phase, where the reduced divisor $D = D_1 \oplus D_2$ is obtained.

In the following, we analyze the behavior of the Cantor-Koblitz algorithm. In general, to perform the sum of two divisors D_1 and D_2 belonging to $\mathcal{J}(\mathcal{H})$, one must:

1. Find the curve $y = g(x)$ passing through the k points $\{P_i = (x_i, y_i)\}$ belonging to the supports of the divisors D_1 and D_2 that need to be summed;
2. Intersect the curve $y = g(x)$ and the hyperelliptic curve $y^2 + h(x)y = f(x)$;
3. Divide the resulting polynomial $q(x) = g^2(x) + h(x)g(x) - f(x)$, obtained from in the previous step, by the polynomial $p(x) = \prod_{i=1}^{k}(x - x_i)$, as this polynomial is a multiple of $q(x)$;
4. Find the roots of the resulting polynomial $\frac{q(x)}{p(x)}$.

In order to take advantage of the Mumford representation of a divisor, the Cantor-Koblitz algorithm needs to be slightly modified. Specifically, in the first five lines of the algorithm, any common points between the divisors to be added are obtained, i.e., $P \in \text{supp}(D_1) \cap \text{supp}(D_2)$. For example, if the divisors to be summed are $D_1 = P_1 + P_2 - 2 \cdot \Omega$ and $D_2 = P_1 + Q_1 - 2 \cdot \Omega$, then point $P_1 = (x_1, y_1)$ is common to both and therefore x_1 constitutes a root with multiplicity greater than one of the polynomial

$u(x) = u_1(x)u_2(x)$ in line 6 of the pseudocode. Subsequently, it is necessary to obtain the curve B that passes through all the points $P \in \text{supp}(D_1) \cup \text{supp}(D_2)$ (line 7 of the pseudocode) in order to obtain a semi-reduced divisor D, which is the divisor $D = D_1 \oplus D_2$, but not necessarily yet in the reduced form $\sum_{i=1}^{n} P_i - n \cdot \Omega$, where $n \leq g$. The loop between lines 8 and 11 reduces the divisor D until the reduced divisor $D = D_1 \oplus D_2$ is obtained, i.e., one obtains $D = \sum_{i=1}^{n} P_i - n \cdot \Omega$, where $n \leq g$. At line 9, the equation of the curve B found at line 7 is combined (in a similar manner to a system of equations) with the hyperelliptic curve in order to obtain the resulting polynomial between these two curves. At line 10, the polynomial $v(x)$ is obtained, whose roots are the ordinates y_i of the points belonging to the support of $D = D_1 \oplus D_2$. Note that the calculation $v(x) = -h(x) - v(x) \pmod{u(x)}$ is performed since it is always necessary to take the opposite divisor, i.e., the divisor $D = \ominus D' = \sum_{P \in \text{supp}(D)} n_P \cdot (\ominus P)$, where $\ominus P = (x_i, -y_i - h(x_i))$ is the opposite point of P with respect to the line $y = h(x)$. Finally, at line 11, the polynomial $u(x)$ becomes monic so that it is uniquely determined.

In fig. 4.1, we present a graphical illustration of the sum of two divisors on a hyperelliptic curve of genus $g = 2$.

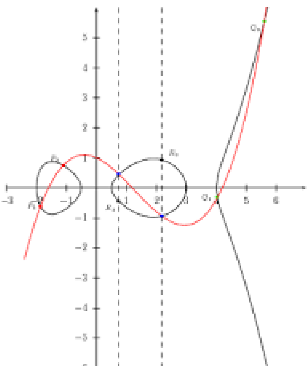

FIGURE 4.1: Sum of two divisors in a hyperelliptic curve of genus 2

Remark 4.7. *The computational complexity of this group law increases as the genus g grows. However, for genus $g = 2$ or $g = 3$, an optimized version of the Cantor-Koblitz algorithm can be defined to accelerate the computation and leverage these curves and their group law for cryptographic purposes [53].*

Remark 4.8. *Note that, in accordance with the group law, the opposite of a divisor $D = (u(x), v(x))$ is given by $\ominus D = (u(x), (-h(x) - v(x)) \pmod{u(x)})$. Furthermore, we use again the notation $n \otimes D$ in order to denote the sum $\bigoplus_{i=1}^{n} D$ if $n > 0$, or $\bigoplus_{i=1}^{-n} (\ominus D)$ if $n < 0$.*

4.2 Meaning of the Cantor-Koblitz algorithm

In this section, we aim to provide a deeper understanding of the algorithm by presenting a simple example. Consider a hyperelliptic curve of genus $g = 2$ and set $h(x) = 0$ for the moment. We also consider a simple case where the two divisors to be added have no common points and each consist of two points, that is, $D_1 = P_1 + P_2 - 2 \cdot \Omega$ and $D_2 = Q_1 + Q_2 - 2 \cdot \Omega$, where $P_i = (x_{1,i}, y_{1,i})$ and $Q_i = (x_{2,i}, y_{2,i})$.

In the Mumford representation, we have $D_1 = (u_1(x), v_1(x))$ and $D_2 = (u_2(x), v_2(x))$, where $\deg(u_1(x)) = \deg(u_2(x)) = g = 2$ and $\deg(v_1(x)) = \deg(v_2(x)) = 1$. Therefore, each divisor D_i is represented by a parabola $u_i(x)$, which contains information on the abscissas of the points in the support of D_i (i.e., $u_i(x_{i,j}) = 0$), and a line $y = v_i(x)$ passing through these points, which contains information on their ordinates (i.e., $v_i(x_{i,j}) = y_{i,j}$), where $i = 1, 2$ and $j = 1, 2$.

In fig. 4.2, we show the graphic representation of the two divisors D_1 and D_2 as pairs (parabola, line), with D_1 being represented in red and D_2 in green.

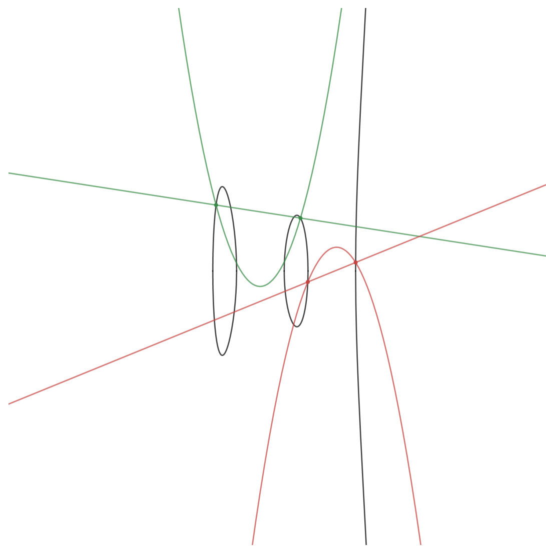

FIGURE 4.2: Representation of divisors by means of the pair (parabola, straight line)

In the present scenario, the procedures outlined in lines 1 to 5 of algorithm 1 are unnecessary due to the absence of any common points between the two divisors. As a result, $d_1 = d = s_1 = s_2 = 1$ and $s_3 = 0$. At line 6, the polynomial $u(x) = u_1(x)u_2(x)$ is calculated. This polynomial represents the quartic curve whose roots are the union of the roots of the polynomials $u_1(x)$ and $u_2(x)$, i.e., $\text{supp}(\text{div}(u(x))) = \text{supp}(\text{div}(u_1(x))) \cap \text{supp}(\text{div}(u_2(x)))$. At line 7, the cubic passing through the four points P_1, P_2, Q_1, and Q_2 is calculated.

Note that

$$F = \{\lambda \cdot p(x_1, \ldots, x_n) + \gamma \cdot q(x_1, \ldots, x_n)\},$$

where $x_1, \ldots, x_n$ are unknowns, represents a family of curves with degree $\deg(p) = \deg(q)$. Additionally, if the curve p vanishes at the points in the set $M = \{M_1, \ldots, M_k\}$ and the curve q vanishes at the points in the set $T = \{T_1, \ldots, T_h\}$, then every polynomial $l \in F$ vanishes at the points in $M \cap T$.

It is worth mentioning that the product $u_i(x)v_i(x)$, which uniquely represents the divisor D_i, constitutes is a degenerate cubic in the current scenario ($g = 2$ and D_1, D_2 consisting of four distinct points), since it can be broken down into the product of a line and a parabola.

In order to obtain the only cubic passing through the points P_1, P_2, Q_1, and Q_2, it is necessary to combine the supports of $u_i(x)$ and $v_j(x)$. At line 7, one finds the cubic as $v(x) = u_1(x)v_2(x) + u_2(x)v_1(x)$ (due to $s_1 = s_2 = 1$ and $s_3 = 0$), which is the unique cubic, passing through the above points, belonging to the family of degenerate

cubics $\{\lambda(u_1(x)v_2(x)) + \gamma(u_2(x)v_1(x))\}$, where $\lambda = s_1 = 1$ and $\gamma = s_2 = 1$. With regards to the principal divisors, one observes that

$$\begin{aligned} \{P_1, P_2\} &\subseteq \operatorname{supp}(\operatorname{div}(u_1(x))), \\ \{P_1, P_2\} &\subseteq \operatorname{supp}(\operatorname{div}(v_1(x))), \\ \{Q_1, Q_2\} &\subseteq \operatorname{supp}(\operatorname{div}(u_2(x))), \\ \{Q_1, Q_2\} &\subseteq \operatorname{supp}(\operatorname{div}(v_2(x))). \end{aligned} \tag{4.3}$$

Thus, it follows that

$$\begin{aligned} \{P_1, P_2, Q_1, Q_2\} &\subseteq \operatorname{supp}(\operatorname{div}(u_1(x)v_2(x))), \\ \{P_1, P_2, Q_1, Q_2\} &\subseteq \operatorname{supp}(\operatorname{div}(u_2(x)v_1(x))). \end{aligned} \tag{4.4}$$

As previously stated, the family of degenerate cubics is comprised of the points that are solutions for both curves. Consequently, one has that $\{P_1, P_2, Q_1, Q_2\} \in \operatorname{supp}(\operatorname{div}(v(x)))$, and $v(x)$ is the unique cubic passing through the four specified points P_1, P_2, Q_1, and Q_2.

It is possible to obtain the same result by performing some algebraic operations. As previously stated, since $d_1 = e_1u_1(x) + e_2u_2(x)$, then by substituting e_2, one has that

$$v(x) = c_1e_1u_1(x)v_2(x) + c_1\frac{d_1 - e_1u_1(x)}{u_2(x)}u_2(x)v_1(x).$$

Additionally, by evaluating $v(x)$ at $x_{1,1}$ and $x_{1,2}$, one obtains $y_{1,1}$ and $y_{1,2}$, respectively, since in this case $d_1 = c_1 = 1$. Hence, $v(x)$ is the curve that passes through the points P_1 and P_2.

Similarly, by substituting e_1, one obtains:

$$v(x) = c_1\frac{d_1 - e_2u_2(x)}{u_1(x)}u_1(x)v_2(x) + c_1e_2u_2(x)v_1(x).$$

Again, by evaluating $v(x)$ at $x_{2,1}$ and $x_{2,2}$, one obtains $y_{2,1}$ and $y_{2,2}$, respectively. Hence, the polynomial $v(x)$ also passes through the points Q_1 and Q_2 and represents the unique cubic passing through the points P_1, P_2, Q_1, and Q_2.

In addition, note that the computation $v(x) \pmod{u(x)}$ has no effect since $\deg(v(x)) = 3$, while $\deg(u(x)) = \deg(u_1(x)u_2(x)) = 4$.

The visual representation of the cubic (in light blue) passing through the points P_1, P_2, Q_1, and Q_2 can be seen in fig. 4.3 and fig. 4.4.

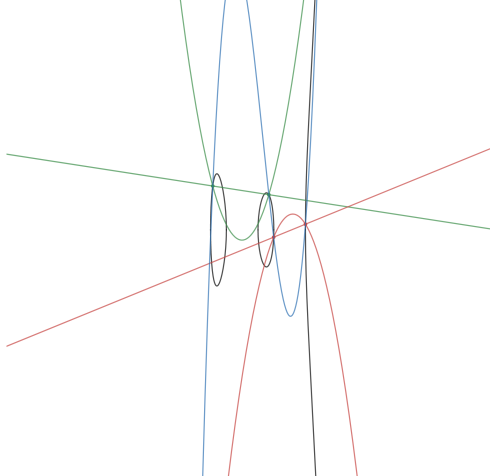

FIGURE 4.3: The only cubic (in light blue) passing through the points belonging to the supports of the two divisors $D_1 = P_1 + P_2 - 2 \cdot \Omega$ (in green) and $D_2 = Q_1 + Q_2 - 2 \cdot \Omega$ (in red)

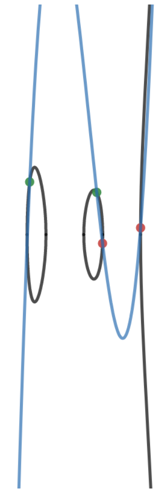

FIGURE 4.4: The only cubic (in light blue) passing through the four points P_1, P_2, Q_1, and Q_2, where P_1 and P_2 are in green, while Q_1 and Q_2 are in red

At line 9, one performs the intersection between the equation of the cubic $v(x)$ and the equation of the hyperelliptic curve $\mathcal{H}$, i.e., one has

$$\begin{cases} y & = v(x), \\ y^2 + h(x)y & = f(x), \end{cases}$$

hence $f(x) - h(x)v(x) - v^2(x) = 0$. Given that $v(x)$ is a cubic and $f(x)$ is a fifth-degree polynomial ($2g + 1 = 5$), the degree of $f(x) - h(x)v(x) - v^2(x)$ is six. The abscissas of four solutions,P_1, P_2, Q_1, and Q_2, are already known, while the other two solutions are the abscissas of the points belonging to the support of the sum divisor. Therefore, at line 9, the polynomial $f(x) - h(x)v(x) - v^2(x)$ is divided by the previously calculated polynomial $u(x)$, which vanishes at the abscissas of P_1, P_2, Q_1, and Q_2. As a result, the ratio $\frac{f(x)-v(x)h(x)-v^2(x)}{u(x)}$ returns the polynomial $u(x)$ of the Mumford representation of the sum divisor D.

In order to determine the ordinates y_i of the points belonging to $\text{supp}(u(x))$, at line 10, one computes $v(x) = -h(x) - v(x) \pmod{u(x)} = -v(x) \pmod{u(x)}$ since, in this example, we have that $h(x) = 0$. The latter leads to $v(x) = -v(x) + u(x)g(x)$, where $g(x)$ is a suitable polynomial, and $u(x)$ is the parabola passing through the two points belonging to the support of the sum divisor. Recall that the polynomial $v(x)$, computed at line 7, is the unique cubic that passes through P_1, P_2, Q_1, Q_2, and intersects the curve in two additional points belonging to the support of the sum divisor. The polynomial modulo operation of $v(x)$ with respect to $u(x)$ removes the points P_1, P_2, Q_1, and Q_2 from the cubic and retains the opposite (with respect to the line $y = h(x) = 0$, i.e., the x-axis) of the remaining two points.

4.3 Doubling of divisors without Cantor-Koblitz algorithm

The Cantor-Koblitz algorithm is particularly useful in computing the double of a divisor. Consider the simple case of a hyperelliptic curve $\mathcal{H}$ of genus $g = 2$ over a field $\mathbb{K}$ and, with $h(x) = 0$. In this scenario, the elements in the Jacobian of the curve take one of the following forms: $D = \Omega - \Omega$, $D = P - \Omega$, or $D = P_1 + P_2 - 2 \cdot \Omega$.

Doubling the divisor $D = \Omega - \Omega$ is trivial as it is the identity divisor. On the other hand, for $D = P - \Omega$, the double of D, i.e., $2 \otimes D$, is also trivial and is given by $2 \cdot P - 2 \cdot \Omega$. In particular, to compute it, one has to compute the tangent line at P to the curve $\mathcal{H}$ and find the solutions common to both curves. This reduces to a tangent line as, in this case, $2 \otimes D$ is already a reduced divisor.

In the case where $D = P_1 + P_2 - 2 \cdot \Omega$, the doubling is slightly more complicated and requires further division into two cases: $P_1 = P_2$ and $P_1 \neq P_2$. In the former case, one has $D = 2 \cdot P_1 - 2 \cdot \Omega$ and $2 \otimes D = 4 \cdot P_1 - 4 \cdot \Omega$. In order to find the reduced divisor of $2 \otimes D$, i.e., a divisor such that there are at most $g = 2$ affine points in its support, one computes the cubic tangent to $\mathcal{H}$ at P_1. This cubic could be obtained through the Taylor series approximation of order 3 of the function $y = \pm\sqrt{f(x)}$ at x_{P_1} (recall that $\mathcal{H}$ is defined by the equation $y^2 = f(x)$). In particular, one approximates the function $y = \sqrt{f(x)}$ if $\sqrt{f(x_{P_1})} = y_{P_1}$ or the function $y = -\sqrt{f(x)}$ if $-\sqrt{f(x_{P_1})} = y_{P_1}$. Note that this approximation changes when $h(x) \neq 0$.

Definition 4.3 (Tangent of degree k of a curve)**.** *Let $\mathcal{C}$ be a generic curve of degree t over $\mathbb{K}$, and let $(x_0, y_0) \in \mathcal{C}(\mathbb{K})$ be a point belonging to $\mathcal{C}$. The Taylor series approximation of $\mathcal{C}$ of order $k < t$ at x_0 is the curve H of degree k which is tangent to the curve $\mathcal{C}$ at x_0.*

The Taylor series approximation involves the calculation of a polynomial of degree k that approximates the curve $\mathcal{C}$ *locally* at x_0 and is tangent to $\mathcal{C}$ at this point.

This local approximation can be used to determine the cubic tangent to the curve $\mathcal{H}$ at P_1 in order to obtain the divisor $2 \otimes D = 4 \cdot P_1 - 4 \cdot \Omega$ in either its reduced form $2 \otimes D = A + B - 2 \cdot \Omega$ or $2 \otimes D = A - \Omega$. This can be achieved by solving a system of equations between the cubic and the hyperelliptic curve.

However, in the case where $D = P_1 + P_2 - 2 \cdot \Omega$, where $P_1 \neq P_2$, the cubic must be tangent to both P_1 and P_2, and this double tangent condition is not easily determined. It should be noted that finding the tangent cubic at P_1 and P_2 using the Taylor series approximation and combining them is not likely to result in the desired cubic. This is because the Taylor series only approximates the function locally and the approximation polynomial has the same concavity at x_0 as the approximated function $y = \pm\sqrt{f(x)}$. However, the tangent cubic may have a different concavity at P_1 and P_2 than the approximated function, as shown in fig. 4.5, where the tangent at P is in green and the tangent at Q is in blue. In fig. 4.6, the proper tangent cubic at P and Q is displayed, and the concavity at P of the tangent is opposite to the concavity at P of $\mathcal{H}$, indicating that it may not be possible to compute the proper cubic tangent at P_1 and P_2 using the Taylor series approximation.

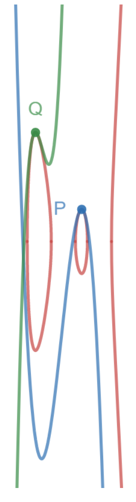

FIGURE 4.5: Tangent cubics with Taylor at points P and Q

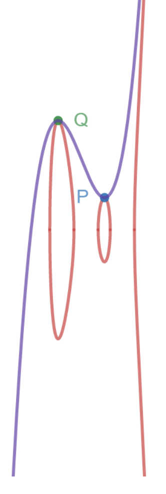

FIGURE 4.6: Tangent cubics at points P and Q

The computation of the tangent cubic at points P_1 and P_2 is performed as follows. Let $y = ax^3 + bx^2 + cx + d$ be the equation of a generic cubic curve. The tangent cubic to the curve $\mathcal{H}$ at P_1 and P_2 is computed by assuming that:

1. P_1 satisfies the equation $y = ax^3 + bx^2 + cx + d$;
2. P_2 satisfies the equation $y = ax^3 + bx^2 + cx + d$;
3. The derivative at $x = x_{P_1}$ of the equation $y = ax^3 + bx^2 + cx + d$ is equal to the derivative of $y = \pm\sqrt{f(x)}$ at x_{P_1};
4. The derivative at $x = x_{P_2}$ of the equation $y = ax^3 + bx^2 + cx + d$ is equal to the derivative of $y = \pm\sqrt{f(x)}$ at x_{P_2}.

This results in a system of four equations in the unknowns a, b, c, and d:

$$\begin{cases} y_{P_1} = ax_{P_1}^3 + bx_{P_1}^2 + cx_{P_1} + d, \\ y_{P_2} = ax_{P_2}^3 + bx_{P_2}^2 + cx_{P_2} + d, \\ 3ax_{P_1}^2 + 2bx_{P_1} + c = \frac{\delta}{\delta x}\left(\pm\sqrt{f(x)}\right)\Big|_{x_{P_1}}, \\ 3ax_{P_2}^2 + 2bx_{P_2} + c = \frac{\delta}{\delta x}\left(\pm\sqrt{f(x)}\right)\Big|_{x_{P_2}}. \end{cases}$$

The solution of this system provides the coefficients a, b, c, and d of the tangent cubic at P_1 and P_2.

It is important to note that the complexity of the calculation increases with the genus of the curve. Hence, the Cantor-Koblitz algorithm greatly simplifies these calculations as the genus increases.

5 Elliptic curves cryptography

In this section, we provide an overview of cryptography based on elliptic and hyperelliptic curves.

Elliptic Curve Cryptography (ECC) [48, 60] is a public-key cryptographic method that uses secure elliptic curves over finite fields for the purpose of encryption and decryption of messages. ECC is particularly useful for secure and efficient key agreement, a protocol in which parties can establish a common key for secure communication without the risk of third-party decryption.

Public-key cryptography, in general, relies on the difficulty of certain mathematical problems. For example, ECC takes advantage of the intractability of the discrete logarithm problem (or DLP), defined in definition 5.1.

Furthermore, ECC has several advantages over other public-key cryptographic methods, such as RSA. For example, ECC requires smaller key sizes to achieve the same level of security, which makes it more efficient in terms of computation and storage. In addition, ECC is less susceptible to quantum computer attacks, which is an increasingly important consideration as quantum computers become more advanced.

Definition 5.1 (Discrete Logarithm Problem)**.** *Let G be a cyclic group of order n, and let $a, b \in G$ be two elements of G such that n_a and n_b are their respective orders. The discrete logarithm problem involves finding the solution of the equality $\log_a(b) = x$ over the group G, i.e., one has to find $x \in G$ such that $x^a = b \pmod{n_a}$.*

Note that if a is a generator of the group G, then the discrete logarithm $\log_a(b)$ is defined for any $b \in G$. On the other hand, if a is not a generator for G, then the discrete logarithm is only defined and exists for any b belonging to the subgroup generated by a. However, if b does not belong to this subgroup, then the logarithm is not defined.

In the case of an elliptic curve $\mathcal{C}$ over the field $\mathbb{K}$, one takes $G = \mathcal{J}(\mathcal{C})$, which is an additive group. In this case, the additive discrete logarithm is clearly defined as $ax = b$. Specifically, one has that $Q = k \otimes P$, where $P \in \mathcal{C}(\mathbb{K})$ is such that $P - \Omega$ is an element of order n (possibly, a generator) of $\mathcal{J}(\mathcal{C})$, Q is the point for which the logarithm is being calculated, and $k \in \{0, \dots, n-1\}$ is the value of the logarithm. The problem of finding the number k such that $Q = k \otimes P$ is known as the elliptic curve discrete logarithm problem (ECDLP). Additionally, in the case of a hyperelliptic curve, this problem is referred as hyperelliptic curve discrete logarithm problem (HECDLP) [15, 42].

The security of ECC depends on various factors such as the chosen elliptic curve, the dimension of its underlying finite field, and others. Currently, the most efficient method to solve the ECDLP is Pollard's rho algorithm for elliptic curves $\mathcal{C}$ with a complexity of $\mathcal{O}(\sqrt{n})$, where $n \leq \text{card}(\mathcal{J}(\mathcal{C}))$ is the order of the subgroup generated by the divisor $P - \Omega$ used as the base point in the computation. The larger the

cardinality of the subgroup generated by P, the security of a cryptographic method relying on the intractability of the ECDLP.

Secure elliptic curves are difficult to find as they must meet certain criteria [36, 50, 67], such as having an order of the Jacobian of at least 2^{160} to ensure a minimum security strength and a cofactor of at most 4 The cofactor $h \in \mathbb{N}$ of an elliptic curve $\mathcal{C}$ is the number of points with the same abscissa in $\mathcal{C}$, and it must be small due to the existence of attacks that use this information to speed up the resolution of the ECDLP. Furthermore, some elliptic curves such as anomalous curves are inherently unsecure.

Definition 5.2. *Let $\mathcal{C}$ be an algebraic curve over a finite field k such that* $\mathrm{card}(\mathcal{J}_k(\mathcal{C})) = \mathrm{card}(k)$. *The curve $\mathcal{C}$ is an anomalous curve.*

Non-anomalous curves are subject to attacks by means, for instance, of pairing mappings, that is, efficiently computable, bilinear, and non-degenerate maps $e\colon G_1 \times G_2 \to G_3$, where typically G_1 and G_2 are cyclic subgroups (such as the Weil pairing, see e.g. §IX.6 [15], §III.8 in [73], and the Ate pairing, see [39]) or quotient groups (such as the Tate pairing, see e.g. [34], and the Eta pairing, see [5]) of the Jacobian of the curve, while G_3 is a subgroup of the multiplicative group of the ground field because the pairing carries the logarithm of an element in G_1 to the logarithm of an element in G_3 (see e.g. [59]). Anomalous curves are safe with respect to these attacks since all the above pairings are defined if and only if the cardinalities of G_1 and G_2 divide $q^k - 1$, where q^k is the cardinality of the ground field.

On the other hand, however, anomalous curves are also subject to attacks as it is possible to map the Jacobian of such curves to the additive group of the finite field k (see [45, 55, 66, 68, 73, 74]). We address the reader to §XI.6 in [73] for a simple polynomial algorithm able to solve the ECDLP for an anomalous curve.

Finally, ECC has been extended to hyperelliptic curves, and this extension is known as hyperelliptic curve cryptography (HCC). The advantage of using HCC lies in its use of hyperelliptic curves with genus $g = 2$, as it has been proven that the complexity of the Hyperelliptic Curve DLP (HCDLP) decreases as the genus increases. In particular, Gaudry's work in [35] demonstrated that the the Index Calculus algorithm, another method that solves the DLP, can be optimized to become competitive and even more efficient than the classic Pollard's rho algorithm for hyperelliptic curves. Specifically, given the hyperelliptic curve $\mathcal{H}$ of genus $g > 0$ over the finite field $\mathbb{F}_{q^n}$, where $n \in \mathbb{N}$ and q is a prime number, Pollard's rho algorithm has a complexity of $\mathcal{O}\left(\sqrt{\mathrm{card}(\mathcal{J}(\mathcal{H}))}\right)$, while the optimized Index Calculus, as proposed by Gaudry, has a complexity of $\mathcal{O}\left(q^{2-\frac{2}{n \cdot g}}\right)$. Note that, when $g = 1$, one has that $n \geq 2$ for this complexity.

For instance, when $n = 1$ and $g = 3$, the optimized Index Calculus has a complexity of $\mathcal{O}\left(q^{2-\frac{2}{3}}\right) = \mathcal{O}\left(q^{\frac{4}{3}}\right)$, which is lower than the complexity of Pollard's rho algorithm with the same parameters since, in this case, the cardinality of the Jacobian of $\mathcal{H}$ is $\mathcal{O}(q^3)$ (see theorem 4.1).

5.1 Diffie-Hellman and ElGamal

The cryptography based on elliptic curves employs the Diffie-Hellman protocol for the key-exchange phase. This scheme returns a tuple of five parameters for each user. Specifically, the parameters are defined as $(\mathcal{C}, q, G, n, h)$, where $\mathcal{C}$ is an elliptic curve

over a finite field $\mathbb{F}$ of characteristic q, G is such that $G - \Omega$ is a generator of $\mathcal{J}(\mathcal{C})$ or a point such that the order of the subgroup generated by $G - \Omega$ is approximately equal to card$(\mathcal{J}(\mathcal{C}))$, n is the order of G, and h is the cofactor (for cryptographic applications, it is preferable to have a cofactor equal to 1, although in general $h \leq 4$). Additionally, each user generates a pair of keys (d, Q), where $d \in \{0, \ldots, n-1\}$ is the randomly generated private-key, and $Q = d \otimes G$ the public-key. Solving the ECDLP problem is necessary in order to determine the private key d.

The Diffie-Hellman protocol offers a secure mechanism for agreeing on a common key to encrypt and decrypt messages.

Definition 5.3 (Diffie-Hellman protocol)**.** *Let G be a cyclic group, and let γ be a generator of G. Let A, B be two users that generate their respective private-keys $a, b \in G$. If γ^a and γ^b are the public keys of A and B, respectively, then γ^{ab} is the secret (or shared) key, and it is common to both A and B.*

Typically, in the case of an elliptic curve, the secret key shared by two users is the abscissa of the common point. Specifically, given the private and public keys of two users, (d_A, Q_A) and (d_B, Q_B), the common point is $d_A \otimes Q_B = d_B \otimes Q_A = d_A d_B \otimes G = (x_{AB}, y_{AB})$, where x_{AB} is the secret key.

The ElGamal encryption scheme also makes use of the Diffie-Hellman protocol to establish a common key for encryption and decryption of messages. Once the common key x_{AB} is agreed upon, the encryption phase of the ElGamal method is very simple. Specifically, the encryption process involves encoding the message m as a point belonging to $\mathcal{C}(\mathbb{F}_q)$ and computing the ciphertext C as $x_{AB} \otimes M$, where M is the encoded message. Typically, a message m is encoded by looking for a point M with abscissa equal to m or, if this point does not exist, to perform other steps involved by the standard adopted. The decryption process involves computing $M = x_{AB}^{-1} \otimes C$ and decoding it to obtain the original message m. The value of x_{AB}^{-1} is computed modulo the order n of G.

6 Code-based cryptography

In this chapter, the fundamentals of coding theory will be presented and its utilization in a cryptographic system referred to as the McElice cryptosystem. The McElice cryptosystem will be thoroughly discussed in section 6.2.

Code-based cryptography is a type of cryptography that uses error-correcting codes to encrypt messages. Furthermore, it is used for several different applications, including secure communication, secure key distribution, secure identification, and secure signature. It is particularly useful for applications that require high levels of security, such as military and government communications, financial transactions, and e-commerce.

One of the main advantages of code-based cryptography is its high level of security. Unlike traditional cryptography, which is based on mathematical problems that can be solved by brute force attacks, code-based cryptography is based on coding theory, which is much more difficult to break.

6.1 Coding theory

Coding theory is a mathematical discipline that deals with the study of error-correction codes, defined in definition 6.1, which are sequences of symbols used to transmit data over a noisy channel in such a way that errors can be detected and corrected.

Hence, coding theory is important because it provides a means of ensuring that information is transmitted accurately and efficiently in situations where noise or interference can cause errors in the transmission process. This is due to the fact that physical channels are not exempt from noise, which can result in the alteration of bits in a message during transmission, making it difficult for the receiver to properly decode the information. In order to tackle this problem, codes are employed to convert a message into a series of coded words that can still be accurately decoded even if errors take place during transmission through a channel affected by noise.

Coding theory is important because it provides a means of ensuring that information is transmitted accurately and efficiently in situations where noise or interference can cause errors in the transmission process. The main applications of coding theory can be found in the fields of telecommunications, computer networks, and data storage.

In telecommunications, coding theory is used to design efficient error-correcting codes for the transmission of voice and data over digital networks, such as satellite and cellular communications.

In computer networks, coding theory is used to design efficient error-correction algorithms for the transmission of data packets over the Internet and other computer networks.

In data storage, coding theory is used to design efficient error-correction codes for the storage of digital data on hard drives, memory chips, and other digital storage devices. This is particularly important in situations where the data stored is critical and cannot be lost, such as in the case of financial records, medical records, and other sensitive information.

In the following section, we will outline crucial definitions central to the field of coding theory.

Definition 6.1 (Code)**.** *A code $\mathcal{C}$ is a vector subspace of the vector space $\mathcal{A}^n$, where $\mathcal{A}$ is an alphabet of symbols, typically a field. Each vector $\mathbf{v} \in \mathcal{C}$ is known as a codeword of $\mathcal{C}$.*

Definition 6.2 (Weight of a codeword)**.** *Let $\mathcal{C}$ be a code over $\mathcal{A}^n$, where $\mathcal{A}$ is a finite field. The weight $|\cdot|$ of a codeword $\mathbf{a} \in \mathcal{C}$ is the number of its non-zero elements, that is, for $\mathbf{a} = (a_1, a_2, \ldots, a_n)$, with $a_i \in \mathcal{A}$, then $|\mathbf{a}| = l$ if there are $n - l$ elements of $\mathbf{a}$ equal to zero.*

Definition 6.3 (Hamming distance)**.** *Let $\mathcal{C}$ be a code over $\mathcal{A}^n$, where $\mathcal{A}$ is a finite field. The Hamming distance, or simply distance, between two codewords $\mathbf{a}, \mathbf{b} \in \mathcal{C}$ is $\mathrm{dist}(\mathbf{a}, \mathbf{b}) = |\mathbf{a} - \mathbf{b}|$, that is, the number of positions in which they differ.*

Definition 6.4 (Minimum distance)**.** *Let $\mathcal{C}$ be a code over $\mathcal{A}^n$, where $\mathcal{A}$ is a finite field. The minimum (Hamming) distance of $\mathcal{C}$ is the smallest distance between any two codewords of $\mathcal{C}$.*

Definition 6.5 (Linear code)**.** *A linear code of parameters $[n, k, d]_q$ is a code $\mathcal{C}$ that is a vector subspace of dimension k of the vector space $\mathbb{F}_q{}^n$, where $\mathbb{F}_q$ is a finite field, and such that the minimum distance of $\mathcal{C}$ is equal to d.*

Definition 6.6 (Error)**.** *Let $\mathcal{C}$ be a code over $\mathbb{F}_q{}^n$, and let $\mathbf{v}, \mathbf{w}$ be two vectors such that $\mathbf{v} \in \mathcal{C}$ and $\mathbf{v} + \mathbf{v}' = \mathbf{w} \notin \mathcal{C}$. The number of errors between the codeword $\mathbf{v}$ and the word $\mathbf{w}$ is the distance $|\mathbf{w} - \mathbf{v}|$.*

Any linear code is an error-correcting code that is able to detect and correct errors. In particular, any linear code $\mathcal{C}$ of minimum distance d is capable of detecting at most $d - 1$ errors and correcting at most $\left\lfloor \frac{d-1}{2} \right\rfloor$ errors.

In addition, any linear code can be represented using a matrix, specifically the generator matrix G or the parity-check matrix H, where $G \cdot H^T = 0$. In other words, the matrix H is the nullspace of G.

In the following, we provide two equivalent definitions for a linear code that use the above matrices.

Definition 6.7. *Let $\mathcal{C} \leq \mathbb{F}_q{}^n$ be a $[n, k, d]_q$ linear code, and let $H \in \mathbb{F}_q{}^{n-k \times n}$ be the parity-check matrix of $\mathcal{C}$. The code $\mathcal{C}$ is the set $\left\{\mathbf{v} \in \mathbb{F}_q{}^n \colon \mathbf{v} \cdot H^T = \mathbf{0}\right\}$.*

Definition 6.8. *Let $\mathcal{C} \leq \mathbb{F}_q{}^n$ be a $[n, k, d]_q$ linear code, and let $G \in \mathbb{F}_q{}^{k \times n}$ be the generator matrix of $\mathcal{C}$. The code $\mathcal{C}$ is the set $\left\{\mathbf{v} \cdot G \in \mathbb{F}_q{}^n \colon \mathbf{v} \in \mathbb{F}_q{}^k\right\}$.*

Remark 6.1. *If the columns of G are ordered in such a way that its first k of them are linearly independent, then the Gauss-Jordan method can be applied in order to reduce G to its standard form, i.e., $G = [I_k|M]$, where I_k is the identity matrix of order k, and M is a matrix of dimension $k \times (n - k)$. Upon reducing G is in standard form, the parity-check matrix H can be computed as $\left[-M^T|I_{n-k}\right]$.*

By representing $\mathcal{C}$ through a matrix, it is possible to compute easily its minimum distance by calculating the rank of the matrix H. Specifically, the minimum distance d is such that d columns of H are linearly dependent, while $d-1$ columns of H are always linearly independent.

Additionally, by the rank theorem (or dimensional theorem), one can prove that $\dim(\mathcal{C}) = k = n - r$, where r is the rank of H, i.e., the minimum distance of the code.

Theorem 6.1 (Singleton). *If $\mathcal{C} \leq \mathbb{F}_q{}^n$ is a $[n,k,d]_q$ linear code, then $d \leq n-k+1$.*

Proof. The proof follows from the rank theorem and the definition of d such that $d-1$ is the maximum number of columns of H that are linearly independent. Specifically, by the rank theorem $r = n-k$, and $d-1 \leq r$ since d is the minimum distance. ■

Definition 6.9 (MDS Code). *A Maximum Distance Separable code (MDS code) is a linear code $\mathcal{C} \leq \mathbb{F}_q{}^n$ of parameters $[n,k,d]_q$ such that $d = n-k+1$.*

6.2 McEliece cryptosystem

The McEliece cryptosystem was first introduced in 1978 and was the first asymmetric encryption algorithm to incorporate randomization during the encryption process [58]. It is considered a candidate for post-quantum cryptography as it is immune to Shor's algorithm attacks [12, 25]. This cryptography system is based on the difficulty of decoding a message using a general linear code, which is an NP-hard problem [6]. In comparison to other cryptographic schemes such as RSA, the McEliece cryptosystem is faster in both the encryption and decryption steps. Initially, the McEliece cryptosystem used binary Goppa codes, however, since 1996 it has been extended to algebraic-geometric Goppa codes (AG Goppa codes) that use curves of genus greater than zero over finite fields with characteristic $q = p^t \geq 2$ [30, 46]. However, in 2008 and 2014, attacks were developed that completely broke McEliece cryptosystems using AG Goppa codes [23, 30].

Despite being a quantum-safe method, the main disadvantage of the McEliece cryptosystem is the larger key sizes compared to other public-key encryption methods such as RSA or ECC. For example, to achieve a security level of 128-bit, the typical public-key size for this method is approximately 2.5 Mbits, while the public-key size for the same security level on ECC is only 256 bits and 2048 bits for the RSA method.

The advancement of quantum computers poses a threat to the security of modern cryptographic algorithms, making it necessary to consider quantum-safe methods, even if there is a performance downgrade.

The following outlines the steps involved in the McElice Cryptosystem. Prior to communication between a sender and a receiver, the receiver must perform the following:

- Select the number t of errors that can be corrected;
- Choose a random $[n,k,d]_q$ linear code that is capable of correcting at least t errors, with an efficient decoding algorithm D. Let G be the generator matrix of the selected code in standard form;
- Select a random non-singular matrix $S \in \mathbb{F}_q{}^{k\times k}$;
- Select a random permutation matrix $P \in \mathbb{F}_2{}^{n\times n}$;

- Compute the matrix $\tilde{G} = S \cdot G \cdot P$;
- Publish the public-key $(\tilde{G}, t)$, and keep secret the private-key (S^{-1}, P^{-1}, D).

In order to encrypt a message M, the sender have to:

- Obtain the public-key $(, t)$ of the receiver;
- Encode the message M into a vector $\mathbf{m}$ of length k;
- Compute a random error vector $\mathbf{e} \in \mathbb{F}_2^n$ such that $|\mathbf{e}| = t$;
- Send the encrypted vector $\mathbf{c} = \mathbf{m} \cdot + \mathbf{e}$ to the receiver.

Conversely, the receiver have to perform the following steps:

- Compute the vector $\mathbf{s} = \mathbf{c} \cdot P^{-1} = \mathbf{m} \cdot S \cdot G + \mathbf{e} \cdot P^{-1}$;
- Apply its private decoding algorithm D to $\mathbf{s}$ in order to remove the error vector $\mathbf{e} \cdot P^{-1}$, so that $D(\mathbf{s}) = \mathbf{m} \cdot S \cdot G$;
- Since the generator matrix G is taken in standard form, it follows that the first k element of $D(\mathbf{s})$ are equal to $\mathbf{z} = \mathbf{m} \cdot S$;
- Compute the vector $\mathbf{m} = \mathbf{z} \cdot S^{-1}$;
- Compute the original message M by decoding the vector $\mathbf{m}$.

Remark 6.2. *The sender and receiver must have a mutual understanding of the encoding process for converting a message M into a vector $\mathbf{m}$ and the decoding process for converting $\mathbf{m}$ back into M.*

Remark 6.3. *If G is not taken in its standard form, then the receiver have to solve a linear system in order to compute the vector $\mathbf{m} \cdot S$ from $D(\mathbf{s})$, since G is a rectangular matrix.*

6.3 Algebraic-geometric Goppa codes

Algebraic-geometric Goppa codes use a Riemann-Roch space over algebraic curve to compute their generator matrices. Specifically, given an algebraic curve $\mathcal{X}$ and a divisor $D \in \text{Div}(\mathcal{X})$, a Riemann-Roch space is the defined as follows:

$$\mathcal{L}(D) = \{f \in \overline{\mathbb{K}}(\mathcal{X})^* : \text{div}(f) + D \text{ is effective}\} \cup \{0\}$$

that is, it is the space of $\overline{\mathbb{K}}$-rational functions f for which $\text{div}(f) + D$ is an effective divisor (see definition 2.20).

Since $\mathcal{L}(D)$ is a vector space, it is possible to find a basis of $\mathcal{L}(D)$. If D is a divisor of positive degree k, then $\langle \mathbf{F_0}, \mathbf{F_1}, \ldots, \mathbf{F_{k-1}} \rangle$ is a base for $\mathcal{L}(D)$.

Remark 6.4. *Any divisor D have a reduced representation, that is, there exists an equivalent divisor D' such that $D \equiv D'$ and D' is unique. Thus, we may consider D to be a reduced divisor without loss of generality.*

Remark 6.5. *By definition, for any pole $Q \in \text{div}(f)$, with $f \in \mathcal{L}(D)$, one has that Q is a zero of D, meaning that the support of D contains at least all the poles of the functions in $\mathcal{L}(D)$.*

Once we have computed the basis of $\mathcal{L}(D)$, we can define an algebraic-geometric Goppa code.

Definition 6.10. *Let D be a reduced divisor of positive degree $\delta D = k$ of the curve $\mathcal{X}$ over $\mathbb{F}_q$, where $q = p^t$ and p is a prime number. Let $\langle \mathbf{F_0}, \mathbf{F_1}, \ldots, \mathbf{F_{k-1}} \rangle$ be a basis of the Riemann-Roch space $\mathcal{L}(D)$, let $T = \{P_1, \ldots, P_n\}$ be a set of n points such that $P_j \in \mathcal{X}(\mathbb{F}_q)$, and $P_j \notin \operatorname{supp}(D)$. The matrix $(G_{ij}) = G \in \mathbb{F}_q{}^{k \times n}$ such that, for $i = 1, \ldots, k$ and $j = 1, \ldots, n$, $G_{ij} = \mathbf{F_{i-1}}(P_j)$, is the generator matrix for an $[n, k, d]_q$ AG Goppa code.*

Remark 6.6. *The matrix G is well defined because all points $P_j \in T$ do not belong to the support of D, which contains at least all the poles of the basis function belonging to $\mathbf{F_i}$.*

Theorem 6.2. *Let $\mathcal{C}$ be an AG Goppa code as in definition 6.10. The minimum distance d of $\mathcal{C}$ is such that $d \geq n - \delta D = n - k$.*

Part II

Curves over $\mathbb{Q}_p$

7 Local fields

In this chapter, we introduce basic concepts about local fields. For further details, we address the reader to classic book, e.g. [69].

In mathematics, a local field $\mathbb{K}$ is a field that is locally compact and has a discrete valuation function. Overall, local fields are an important area of study in mathematics, with applications in a wide range of fields including algebraic number theory, algebraic geometry, and harmonic analysis.

One of the key properties of local fields is that they are equipped with a topology that is induced by the absolute value. In particular, $\mathbb{K}$ is a local field if it is complete with respect to a topology induced by a discrete valuation function ν, and if its residue field is finite.

There are two main types of local fields: Archimedean and non-Archimedean. Archimedean local fields are fields that are equipped with an absolute value that is compatible with the usual absolute value on the real or complex numbers. Examples of Archimedean local fields include the real numbers $\mathbb{R}$ and the complex numbers $\mathbb{C}$.

On the other hand, non-Archimedean local fields are fields that are equipped with an absolute value that is not compatible with the usual absolute value on the real or complex numbers. More precisely, non-Archimedean local fields have a discrete valuation, which means that the absolute value takes on only discrete values. These fields are typically denoted by a symbol such as $\mathbb{Q}_p$ (the field of p-adic numbers) or $\mathbb{F}_q((T))$ (the field of formal Laurent series over the finite field $\mathbb{F}_q$), where p is a prime number and q is a power of a prime number. More precisely, every non-Archimedean local field of characteristic zero is isomorphic to $\mathbb{Q}_p$, while every non-Archimedean local field of characteristic p is isomorphic to $\mathbb{F}_q((T))$, where q is a power of the prime p.

In particular, the field of formal Laurent series is an extension of the finite field $\mathbb{F}_q$, and it is equipped with a non-Archimedean absolute value, which has the property that the value of a Laurent series is determined by the lowest power of T that appears in the series. For example, in the field of formal Laurent series over the finite field $\mathbb{F}_2$, the Laurent series $T + T^2 + T^3 + \cdots$ has a value of $\frac{1}{T}$, because the lowest power of T that appears in the series is T^1. Similarly, the Laurent series $1 + T + T^2 + \cdots$ has a value of 1, because the lowest power of T that appears in the series is $T^0 = 1$.

On the other hand, the p-adic numbers are an extension of the rational numbers, and they are equipped with a non-Archimedean absolute value. This absolute value has the property that the value of a number is determined by its highest power of p in its prime factorization. For example, in the 2-adic numbers, the number 5 has a value of $\frac{1}{2}$, because the highest power of 2 in its prime factorization is $2^0 = 1$. Similarly, the number 12 has a value of $\frac{1}{4}$, because the highest power of 2 in its prime factorization is $2^2 = 4$.

Given the surjective discrete normalized valuation function $\nu\colon \mathbb{K} \to \mathbb{Z} \cup \{\infty\}$ of the local field $\mathbb{K}$, one defines the following important objects for a local field:

- $\mathcal{O}_{\mathbb{K}} = \{x \in \mathbb{K}\colon \nu(x) \geq 0\}$, that is, the ring of integers of $\mathbb{K}$, which is a discrete valuation ring;
- $\mathcal{O}_{\mathbb{K}}^{*} = \{x \in \mathbb{K}\colon \nu(x) = 0\}$, that is, the units of $\mathcal{O}_{\mathbb{K}}$;
- $\mathfrak{m}_{\mathbb{K}} = \{x \in \mathbb{K}\colon \nu(x) > 0\}$, that is, the unique non-zero prime ideal of $\mathcal{O}_{\mathbb{K}}$, which is also the unique maximal ideal of $\mathcal{O}_{\mathbb{K}}$;
- $\varpi_{\mathbb{K}}$, that is, a generator of $\mathfrak{m}_{\mathbb{K}}$ which is the uniformizer of $\mathbb{K}$;
- $\mathcal{O}_{\mathbb{K}}/\mathfrak{m}_{\mathbb{K}}$, that is, the finite residue field of $\mathbb{K}$, which is the quotient field of $\mathcal{O}_{\mathbb{K}}$ modulo $\mathfrak{m}_{\mathbb{K}}$.

By using the above objects, every non-zero element $x \in \mathbb{K}$ can be expressed with the generator $\varpi_{\mathbb{K}}$ as follows: $x = {\varpi_{\mathbb{K}}}^{t}u$, where $u \in \mathcal{O}_{\mathbb{K}}^{*}$ and $t \in \mathbb{Z}$ is a unique integer. In particular, one has that ν maps every $x \in \mathbb{K}$ onto the unique integer t that defines the above equivalence, that is, $\nu\colon x \mapsto t$ such that $x = {\varpi_{\mathbb{K}}}^{t}u$, with $\nu(0) = \infty$.

If q is the cardinality of the residue field of $\mathbb{K}$, then there is a natural definition of the absolute value on $\mathbb{K}$ induced by its structure as a local field, that is, $|x| = q^{-\nu(x)}$, where $x \in \mathbb{K}$.

Remark 7.1. *Note that the ring of integers described above is a generalization of the ring of integers over an algebraic number field $\mathbb{F}$. Specifically, in this latter case, the ring of integers $\mathcal{O}_{\mathbb{F}}$ is the ring of all the algebraic integers in $\mathbb{F}$, that is, the roots of monic polynomials $h(x) = x^n + \sum_{i=0}^{n-1} c_i x^i$, with integer coefficients c_i. However, one can find a suitable discrete valuation function in order to define $\mathcal{O}_{\mathbb{F}}$ in the same way. In particular, this ring may be defined as all the elements which are integers in every non-Archimedean completion.*

7.1 The p-adic numbers

The primary focus of this thesis is the study of algebraic curves $\mathcal{C}$ of genus $g \geq 1$ over the field $\mathbb{Q}_p$ of p-adic numbers, using an inverse limit process. Specifically, we study the same curve over $\mathbb{Z}/p\mathbb{Z}$, $\mathbb{Z}/p^2\mathbb{Z}$, ..., $\mathbb{Z}/p^k\mathbb{Z}$ approaching $\mathbb{Q}_p$ through this inverse limit with $k \to \infty$. We address the reader to the notes of J.W.S. Cassels [21] and C. Xavier in [20] to have further details regarding p-adic numbers and a series of algorithm to implement them.

Our aim is to study the Jacobian of an elliptic curve over $\mathbb{Q}$ by starting with the study of the Jacobian on the same curve over $\mathbb{Q}_p$. In the latter case, unlike in finite groups such as GF(p), going from $\mathbb{Z}/p\mathbb{Z}$ to $\mathbb{Z}/p^k\mathbb{Z}$ involves moving from a field to a ring in which there are elements that may not have an inverse. Therefore, rather than working with points in non-homogeneous coordinates (x, y), we will consider both the curve and points in homogeneous coordinates, where $P = [Z : X : Y] \equiv \left(\frac{X}{Z}, \frac{Y}{Z}\right) = (x, y)$.

As already mentioned in the previous section, the p-adic numbers are a completion of $\mathbb{Q}$ using the p-adic metric, which determines if two numbers are close. In particular, fixing a prime number p, any number can be written as a linear combination of powers

of p, that is,

$$n = \sum_{i=k}^{\infty} a_i p^i, \tag{7.1}$$

where $k \in \mathbb{Z}$, and if $a_i = 0$ for $i < 0$ and $a_i \in \{0, \ldots, p-1\}$ for $i \geq 0$, then n is a positive integer. This expansion is unique if $n \neq 0$ and $a_k \neq 0$.

Remark 7.2. *All the p-adic numbers with $k \geq 0$ form a subring of $\mathbb{Q}_p$. This subring is known as the ring $\mathbb{Z}_p$ of p-adic integers, because it is the ring of integers of the field $\mathbb{Q}_p$. Alternatively, $\mathbb{Z}_p$ can be viewed as the inverse limit of the ring $\mathbb{Z}/p^k\mathbb{Z}$ of integers modulo p^k.*

The p-adic order of a rational number n can be defined as the highest power of p for which n is divisible. More precisely, if $m(\cdot)$ is a function defined as follows:

$$m(i) = \max_l \left\{ l \in \mathbb{N} \colon p^l \mid i \right\} \tag{7.2}$$

with $i \in \mathbb{Z}$, then the p-adic evaluation function $\nu_p(\cdot)$ is defined as follows:

$$\nu_p(n) = \nu_p\left(\frac{a}{b}\right) = \begin{cases} m(a) - m(b) & \text{if } n \neq 0, \\ \infty & \text{if } n = 0, \end{cases} \tag{7.3}$$

then $\nu_p(n)$ is the p-adic order of n. Note that any rational number can be represented as the ratio of two integers multiplied by a power of p:

$$n = p^c \frac{t}{d}, \tag{7.4}$$

where $c, t, d \in \mathbb{Z}$, c is unique, and $p \nmid td$. Thus, the p-adic order of n is equal to c.

Based on this, one can define the p-adic norm of n as

$$|n|_p = p^{-\nu_p(n)}, \tag{7.5}$$

where $|0|_p = 0$. This norm allows one to define the following p-adic metric, which measures the distance of two rational numbers $n_1, n_2 \in \mathbb{Q}$:

$$d(n_1, n_2) = |n_1 - n_2|_p. \tag{7.6}$$

In particular, $n_1, n_2 \in \mathbb{Q}$ are close, according to the p-adic metric, if their difference is divisible by a high power of p. Note that this norm defines a distance because:

$$\begin{aligned} &|n|_p \geq 0 && \forall n \in \mathbb{Q}, \\ &|n|_p = 0 && \text{if } n = 0, \\ &|n|_p = |-n|_p, \\ &|n_1 n_2|_p = |n_1|_p |n_2|_p, \\ &|n_1 + n_2|_p \leq \max\left(|n_1|_p, |n_2|_p\right). \end{aligned} \tag{7.7}$$

8 Elliptic curves over local rings

Let $\mathbb{K}$ be a local field, $\mathcal{O}_{\mathbb{K}}$ its ring of integers, $\mathfrak{m}_{\mathbb{K}}$ its prime ideal, and $k = \mathcal{O}_{\mathbb{K}}/\mathfrak{m}_{\mathbb{K}}$ its residue field.

First, we remark that for elliptic curves $\overline{\mathcal{W}}$ in short Weierstrass form, defined by the equation $y^2 = x^3 + ax + b$, whose reduction modulo $\mathfrak{m}_{\mathbb{K}}$ is non-singular, the following sequence:

$$0 \longrightarrow \mathfrak{m}_{\mathbb{K}} \xrightarrow{\text{Exp}_{\overline{\mathcal{W}}}} \mathcal{J}_{\mathbb{K}}\left(\overline{\mathcal{W}}\right) \xrightarrow{\text{Mod}_{\overline{\mathcal{W}}}} \mathcal{J}_{k}\left(\overline{\mathcal{W}}\right) \longrightarrow 0 \tag{8.1}$$

is exact [40, 51, 76] (see also [73, ch. §VII]), thus $\text{Im}(\text{Exp}_{\overline{\mathcal{W}}}) = \text{Ker}(\text{Mod}_{\overline{\mathcal{W}}})$, $\text{Exp}_{\overline{\mathcal{W}}}$ is a monomorphism, $\text{Mod}_{\overline{\mathcal{W}}}$ is an epimorphism, and one has that

$$\mathcal{J}_{k}\left(\overline{\mathcal{W}}\right) \cong \mathcal{J}_{\mathbb{K}}\left(\overline{\mathcal{W}}\right)/\text{Ker}(\text{Mod}_{\overline{\mathcal{W}}}) = \mathcal{J}_{\mathbb{K}}\left(\overline{\mathcal{W}}\right)/\text{Im}(\text{Exp}_{\overline{\mathcal{W}}}). \tag{8.2}$$

Two maps were defined in the previous short exact sequence: $\text{Exp}_{\overline{\mathcal{W}}}$ and $\text{Mod}_{\overline{\mathcal{W}}}$. The map $\text{Mod}_{\overline{\mathcal{W}}}$ is simply the reduction modulo $\mathfrak{m}_{\mathbb{K}}$ of the coordinates of the points $P = [Z : X : Y]$ in $\overline{\mathcal{W}}(\mathbb{K})$ which, up to a multiplication times a suitable $t \in \mathcal{O}_{\mathbb{K}}$, have integral entries $Z, X, Y \in \mathcal{O}_{\mathbb{K}}$:

$$\begin{aligned} \text{Mod}_{\overline{\mathcal{W}}}\colon\ \mathcal{J}_{\mathbb{K}}\left(\overline{\mathcal{W}}\right) &\to \mathcal{J}_{k}\left(\overline{\mathcal{W}}\right) \\ P = [Z : X : Y] &\mapsto [Z \pmod{\mathfrak{m}_{\mathbb{K}}} : X \pmod{\mathfrak{m}_{\mathbb{K}}} : Y \pmod{\mathfrak{m}_{\mathbb{K}}}]. \end{aligned} \tag{8.3}$$

Note that $\text{Mod}_{\overline{\mathcal{W}}}$ is trivially surjective for Hensel's lemma (see proof in [73, sec. §VII.2.1]).

Furthermore, the function $\text{Exp}_{\overline{\mathcal{W}}}$ is defined as follows:

$$\begin{aligned} \text{Exp}_{\overline{\mathcal{W}}}\colon\ \mathfrak{m}_{\mathbb{K}} &\longrightarrow \mathcal{J}_{\mathbb{K}}\left(\overline{\mathcal{W}}\right) \\ z &\longmapsto \left[1 : \wp(z) : \frac{1}{2}\wp'(z)\right], \\ 0 &\longmapsto \Omega, \end{aligned} \tag{8.4}$$

where $\wp$ is the Weierstrass elliptic function (see subsec. 3.1.2).

Since $z = 0$ is the only element of $\mathfrak{m}_{\mathbb{K}}$ mapped to Ω, the homomorphism $\text{Exp}_{\overline{\mathcal{W}}}$ is into; thus, for any z in a neighborhood of zero, one can define $\text{Exp}_{\overline{\mathcal{W}}}^{-1}$ (see §IV and §VII [73] for further details) such that $\text{Exp}_{\overline{\mathcal{W}}}^{-1}(\text{Exp}_{\overline{\mathcal{W}}}(z)) = z$. Specifically, if $1 < i \leq 5$, one has that the function

$$\text{Exp}_{\overline{\mathcal{W}}}^{-1}\colon\ [T : X : Y] \longmapsto -2\,\frac{X}{Y} \tag{8.5}$$

is such that $\text{Exp}_{\overline{\mathcal{W}}}^{-1}(\text{Exp}_{\overline{\mathcal{W}}}(z)) = z \pmod{{\mathfrak{m}_{\mathbb{K}}}^i}$. As long as the domain of $\text{Exp}_{\overline{\mathcal{W}}}^{-1}$ is $\text{Im}(\text{Exp}_{\overline{\mathcal{W}}})$, one has that $-2\frac{X}{Y}$ is equivalent to $-2\frac{\wp(z)}{\wp'(z)}$, whose first terms in a Taylor

series expansion are

$$z + \frac{g_2}{10}z^5 + \frac{3g_3}{28}z^7 + \frac{g_2^2}{120}z^9 + \frac{23g_2g_3}{1540}z^{11} + O(z^{13}).$$

Hence, for any z such that $\mathrm{Exp}(z) = P = [T : X : Y]$ one has that $z - \left(-2\frac{X}{Y}\right) \in \mathfrak{m}_{\mathbb{K}}{}^i$, for $1 < i \leq 5$.

Finally, it is worth to note that the above exact sequence does not split over a field $\mathbb{K}$ if one supposes that the elliptic curve in Weierstrass form is an anomalous curve (see definition 5.2).

Theorem 8.1. *If $k = \mathcal{O}_{\mathbb{K}}/\mathfrak{m}_{\mathbb{K}}$ is finite, and $\mathcal{W}$ is not an anomalous curve, then $\mathcal{J}_{\mathbb{K}}(\mathcal{W})$ is isomorphic to the direct sum of $\mathcal{J}_k(\mathcal{W})$ and $\mathfrak{m}_{\mathbb{K}}$.*

Proof. As $k = \mathcal{O}_{\mathbb{K}}/\mathfrak{m}_{\mathbb{K}}$ is finite and $\mathcal{W}$ is not anomalous, for any $1 \leq h \in \mathbb{Z}$, the sequence:

$$0 \longrightarrow H \longrightarrow \mathcal{J}_H(\mathcal{W}) \longrightarrow \mathcal{J}_k(\mathcal{W}) \longrightarrow 0,$$

where $H = \mathfrak{m}_{\mathbb{K}}/(\varpi_{\mathbb{K}}^h\mathcal{O}_{\mathbb{K}})$, with $\varpi_{\mathbb{K}}$ the uniformizer of $\mathbb{K}$, is splitting by the Schur-Zassenhaus theorem and defines, therefore, a section $\sigma_{\mathcal{W}}^h \colon \mathcal{J}_k(\mathcal{W}) \to \mathcal{J}_H(\mathcal{W})$ which is a homomorphism. Taking the inverse limit $\sigma_{\mathcal{W}} = \lim\limits_{h\to\infty} \sigma_{\mathcal{W}}^h$, we obtain a section $\sigma_{\mathcal{W}} \colon \mathcal{J}_k(\mathcal{W}) \to \mathcal{J}_H(\mathcal{W})$ which is a homomorphism, hence the sequence is splitting. ■

In the following, we provide an example of anomalous elliptic curve in Weierstrass form for which the exact sequence does not splits.

Example 1. *Let $\mathcal{W}$ be the elliptic curve in Weierstrass form defined by the equation $y^2 = x^3 + 4x + 7$ over $k = \mathrm{GF}(53)$, whose Jacobian can be readily verified to have 53 elements. Hence, $\mathcal{J}_k(\mathcal{W})$ is isomorphic to the cyclic group C_{53}. However, $\mathcal{J}_{\mathbb{Z}/53^2\mathbb{Z}}(\mathcal{W}) \neq C_{53} \oplus C_{53}$ as the point $P = (3, 130) \in \mathcal{W}(\mathbb{Z}/53^2\mathbb{Z})$ is such that $53(P - \Omega) = [0 : 53 : 1603] - \Omega \neq \Omega - \Omega$.*

8.1 The case $\mathbb{K} = \mathbb{Q}_p$

In the case $\mathbb{K} = \mathbb{Q}_p$, one has that its ring of integers $\mathcal{O}_{\mathbb{K}}$ is the ring $\mathbb{Z}_p$ of p-adic integers, its prime ideal $\mathfrak{m}_{\mathbb{K}}$ is $p\mathbb{Z}_p$ (which uniformizer $\varpi_{\mathbb{K}}$ is equal to p), and its residue field $\mathcal{O}_{\mathbb{K}}/\mathfrak{m}_{\mathbb{K}}$ is $\mathbb{Z}/p\mathbb{Z} = \mathrm{GF}(p)$.

In this case, the exact sequence in equation (8.1) indicates that $\mathcal{J}_{\mathbb{Q}_p}\left(\overline{\mathcal{W}}\right) \cong \mathcal{J}_{\mathbb{Z}/p\mathbb{Z}}\left(\overline{\mathcal{W}}\right) \oplus \mathrm{Im}(\mathrm{Exp}_{\overline{\mathcal{W}}})$, that is, the Jacobian of the curve over $\mathbb{Q}_p$ is simply direct sum of the Jacobian of the curve over the finite field $\mathbb{Z}/p\mathbb{Z}$ and the integers modulo p^{k-1}. This result, although previously known, has significant advantages since as it allows for group operations on the curve over $\mathbb{Q}_p$ to be performed in simple way, by treating the Jacobian of the curve over $\mathbb{Z}/p\mathbb{Z}$ and modulo integers addition as separate entities.

Recall that, in equation (3.7), we gave the Laurent series expansion for the Weierstrass $\wp$-function and its derivative $\wp'$ for a complex number z. As here we are now focusing on the field $\mathbb{Q}_p$, one has to take into account the convergence radius of these series over $\mathbb{Q}_p$. In the context of the field of p-adic numbers, a convergence neighborhood of zero is given by multiples of p, that is, when $p \mid z$. In this neighborhood, these series

always converge since $c_k z^{2k-2} \equiv 0 \pmod{p^h}$, and $(2k-2)c_k z^{2k-3} \equiv 0 \pmod{p^h}$, for a suitable positive integer h.

Moreover, recall that we aim to study the field $\mathbb{Q}_p$ as an inverse limit $\mathbb{Z}_p = \varprojlim \mathbb{Z}/p^k\mathbb{Z}$, approaching $\mathbb{Z}_p$ for $k \to \infty$. In this case, the map $\mathrm{Exp}_{\overline{\mathcal{W}}}$ becomes

$$\begin{aligned} \mathrm{Exp}_{\overline{\mathcal{W}}} \colon\ p\mathbb{Z}_p/p^k\mathbb{Z}_p &\longrightarrow \overline{\mathcal{W}}\Big(\mathbb{Z}/p^k\mathbb{Z}\Big) \\ z = ph \quad &\longmapsto \left[1 : \wp(z) : \frac{1}{2}\wp'(z)\right], \end{aligned} \tag{8.6}$$

where $h = 1, 2, \ldots, p^{k-1}$, since $\mathbb{Z}_p = \varprojlim \mathbb{Z}/p^k\mathbb{Z}$.

Remark 8.1. *Note that in this case we consider, as the domain of* $\mathrm{Exp}_{\overline{\mathcal{W}}}$*, the quotient* $p\mathbb{Z}_p/p^k\mathbb{Z}_p$ *since, modulo* p^k*,* $\mathrm{Exp}_{\overline{\mathcal{W}}}(ph) = \mathrm{Exp}_{\overline{\mathcal{W}}}\Big(p(h+p^{k-1})\Big)$*, with* $h \in \mathbb{Z}$*.*

Remark 8.2. *Note that, as there is a natural isomorphism from* $\mathrm{Im}(\mathrm{Exp})$ *and* $\mathbb{Z}/p^{k-1}\mathbb{Z}$ *through the following map:*

$$\begin{aligned} p\mathbb{Z}_p/p^k\mathbb{Z}_p &\longrightarrow \mathbb{Z}/p^{k-1}\mathbb{Z} \\ ph &\longmapsto h, \end{aligned}$$

where $h = 1, 2, \ldots, p^{k-1}$*, we have that* $\mathcal{J}_{\mathbb{Z}/p^k\mathbb{Z}}\Big(\overline{\mathcal{W}}\Big) = \mathcal{J}_{\mathbb{Z}/p\mathbb{Z}}\Big(\overline{\mathcal{W}}\Big) \oplus \mathbb{Z}/p^{k-1}\mathbb{Z}$*.*

In addition, as we are approximating $\mathbb{Q}_p$ with $\mathbb{Z}/p^k\mathbb{Z}$, with $k \to \infty$, the map $\mathrm{Exp}_{\overline{\mathcal{W}}}$ for elliptic curves in short Weierstrass form should be rewritten as follows:

$$\begin{aligned} \mathrm{Exp}_{\overline{\mathcal{W}}} \colon\ p\mathbb{Z}_p/p^k\mathbb{Z}_p &\longrightarrow \mathcal{W}\Big(\mathbb{Z}/p^k\mathbb{Z}\Big) \\ z \quad &\longmapsto \left[tz^3 : tz^3\wp(z) : tz^3\frac{1}{2}\wp'(z)\right], \end{aligned} \tag{8.7}$$

where t is the least common multiple between the denominators of the series expansion of $\wp$ and $\wp'$ (see equation (3.7)). In particular, the multiplication by the factor tz^3 has to be done in order to make all coordinates integer, and therefore to avoid modular inversions when the denominator is a multiple of p as we move from the field $\mathbb{Z}/p\mathbb{Z}$ to the ring $\mathbb{Z}/p^k\mathbb{Z}$.

Since $\mathrm{Im}(\mathrm{Exp}_{\overline{\mathcal{W}}}) = \mathrm{Ker}(\mathrm{Mod}_{\overline{\mathcal{W}}})$, then $P \in \mathrm{Im}(\mathrm{Exp}_{\overline{\mathcal{W}}})$ if $\mathrm{Mod}_{\overline{\mathcal{W}}}(P) = \Omega$. Therefore, the points belonging to $\mathrm{Im}(\mathrm{Exp}_{\overline{\mathcal{W}}})$ have the following form $P = [ph_1 : ph_2 : h_3]$ with $p \nmid h_3$.

We claim that the number of elements belonging to $\mathrm{Im}(\mathrm{Exp}_{\overline{\mathcal{W}}})$ is p^{k-1}, where $\mathrm{Exp}_{\overline{\mathcal{W}}}$ is the map in equation (8.7). In particular, the Weierstrass $\wp$-function cycle modulo p^k is exactly long p^{k-1}. This is immediately evident if one considers that h goes from 1 to p^{k-1} before the function $\mathrm{Exp}_{\overline{\mathcal{W}}}(z)$ is repeated, where $z = ph$. In fact, if $h = p^{k-1} + 1$ we have that $z = p(p^{k-1}+1) \equiv p \pmod{p^k}$. Therefore, we observe that there are p^{k-1} points belonging to the image of $\mathrm{Exp}_{\overline{\mathcal{W}}}$ are none other than the points that break the Jacobian of the curve over $\mathbb{Z}/p^k\mathbb{Z}$ in the direct sum.

Thus, for a non-anomalous curve, the map $\mathrm{Exp}_{\overline{\mathcal{W}}}$ allows us to speed up the addition operation by splitting the original group $\mathcal{J}_{\mathbb{Z}/p^k\mathbb{Z}}\Big(\overline{\mathcal{W}}\Big)$ into a pair (P, c), where $P \in \overline{\mathcal{W}}(\mathbb{Z}/p\mathbb{Z})$ and $c \in \mathbb{Z}/p^{k-1}\mathbb{Z}$.

Finally, to aid in our upcoming discussions, we introduce a new function, denoted as

$$\begin{aligned} \mathrm{Mod}_{p^k}\colon\ \mathcal{J}_{\mathbb{K}}\big(\overline{\mathcal{W}}\big) &\to \mathcal{J}_{\mathbb{Z}/p^k\mathbb{Z}}\big(\overline{\mathcal{W}}\big) \\ P=[Z:X:Y] &\mapsto \left[Z \pmod{p^k} : X \pmod{p^k} : Y \pmod{p^k}\right]. \end{aligned} \tag{8.8}$$

8.2 Group law over $\mathbb{Q}_p$

Although we possess the two maps $\mathrm{Exp}_{\overline{\mathcal{W}}}$ and $\mathrm{Log}_{\overline{\mathcal{W}}} = \mathrm{Exp}_{\overline{\mathcal{W}}}^{-1}$, the direct summation of two points P and Q belonging to an elliptic curve in Weierstrass form over $\mathbb{Z}/p^k\mathbb{Z}$ is not feasible. In particular, given P and Q belonging to $\overline{\mathcal{W}}\big(\mathbb{Z}/p^k\mathbb{Z}\big)$, operations such as $P \oplus Q$ or $n \otimes P$ may not be possible, where n is a positive integer. For example, in affine coordinates, we may encounter a denominator d such that p divides d, making the inversion operation invalid. Additionally, in projective coordinates, we may obtain a point $R \notin \overline{\mathcal{W}}\big(\mathbb{Z}/p^k\mathbb{Z}\big)$ or the point $[0:0:0]$, which is not defined.

We attempted to extend the addition operation to obtain the exact point. Specifically, we attempted to sum the two points over $\mathbb{Z}/p^{3k}\mathbb{Z}$ and then reduce them over $\mathbb{Z}/p^k\mathbb{Z}$, i.e. if $P \oplus Q$ is not feasible over $\mathbb{Z}/p^k\mathbb{Z}$, we sum P and Q over $\mathbb{Z}/p^{3k}\mathbb{Z}$ and then reduce the resulting point modulo p^k. However, this approach is not viable when the denominator is not coprime with the characteristic of the defining field of $\overline{\mathcal{W}}$. In such cases, we cannot sum two points by extending the defining field. We note that since a point P can be expressed as $\left[z^3 : z^3\wp(z) : \frac{z^3}{2}\wp'(z)\right]$ over $\mathbb{C}$, then we could at most compute $P \oplus Q$ over $\mathbb{Z}/p^{3k}\mathbb{Z}$ to obtain the exact result, provided that p divides z if $P \in \mathrm{Im}(\mathrm{Exp})$ and $P \in \overline{\mathcal{W}}\big(\mathbb{Z}/p^k\mathbb{Z}\big)$. Attempting to compute $P \oplus Q$ over $\mathbb{Z}/p^{3k+c}\mathbb{Z}$, where c is a positive integer, is futile.

The group law over $\mathbb{Q}$ will always return the exact result. Specifically, we define an elliptic curve in Weierstrass form over $\mathbb{Q}$ that, when reduced modulo p^k, gives us our curve and has a point P that, when reduced modulo p^k, gives us the generator point of the reduced curve. Using this curve over $\mathbb{Q}$, we perform group law operations over $\mathbb{Q}$ and, finally, reduce modulo p^k the resulting point to obtain the exact point. In particular, let P be a point belonging to $\overline{\mathcal{W}}(\mathbb{Q})$, and let n be a positive integer, thus $(n \otimes P) \pmod{p^k}$ is surely the exact point belonging to $\overline{\mathcal{W}}\big(\mathbb{Z}/p^k\mathbb{Z}\big)$.

Although moving to $\mathbb{Q}$ gives us an exact representation of points belonging to $\overline{\mathcal{W}}\big(\mathbb{Z}/p^k\mathbb{Z}\big)$ and enables us to perform cryptographic operations with these curves, a significant issue arises due to the increasing size of the input. To understand why the input size increases, we introduce the concepts of height, logarithmic height, and canonical height of a point.

Definition 8.1 (Point height)**.** *Let $\mathcal{W}$ be an elliptic curve in Weierstrass form over $\mathbb{Q}$, and let $P=(x,y)$ be a point belonging to $\mathcal{W}(\mathbb{Q})$ such that $x=\frac{a}{b}$, where $\gcd(a,b)=1$. The height of the point P is*

$$h(P) = \max\left(|a|, |b|\right). \tag{8.9}$$

Remark 8.3. *If $P=(x,y)\in\mathcal{W}(\mathbb{K})$ is such that $x \notin \mathbb{Q}$, then $h(P)=|x|$, where $\mathbb{K}$ is a field different from $\mathbb{Q}$.*

In particular, one usually considers the logarithmic height or the canonical height of a point.

Definition 8.2 (Logarithmic height)**.** *Let $\mathcal{W}$ be an elliptic curve over $\mathbb{Q}$, and let $P = (x, y)$ be a point belonging to $\mathcal{W}(\mathbb{Q})$. The logarithmic height of the point P is*

$$h_L(P) = \log(h(P)). \tag{8.10}$$

Thus, the logarithmic height of P is a measure of the bits required to represent the x coordinate of P.

Definition 8.3 (Canonical height or Néron–Tate height)**.** *Let $\mathcal{W}$ be an elliptic curve over $\mathbb{Q}$, and let $P = (x, y)$ be a point belonging to $\mathcal{W}(\mathbb{Q})$. The Néron–Tate height (or canonical height) of the point P is*

$$\hat{h}(P) = \lim_{n\to\infty} \frac{h_L(n \otimes P)}{n^2} = h_L(P) + \mathcal{O}(1). \tag{8.11}$$

In particular, one of the properties of the canonical height (and the logarithmic height) of a point states that $\hat{h}(n \otimes P) = n^2\hat{h}(P)$.

Thus, given a point $P \in \mathcal{W}(\mathbb{Q})$ whose coordinates have a single digit, the point $n \otimes P$ over $\mathbb{Q}$ may have coordinates with hundreds of thousands of digits. This exponential increase in the size of the coordinates leads to a rapid saturation of the computer's RAM and a significant decline in terms of group law performance.

Despite this limitation, currently, there is no other method capable of computing the sum of any two points over $\mathbb{Z}/p^k\mathbb{Z}$ without error.

8.3 The Log function

In this section, we assume that the local field $\mathbb{K}$ is the field of p-adic numbers with the approximation in section 8.1.

In order to be able to efficiently perform the sum of points defined by the map $\mathrm{Exp}_{\overline{\mathcal{W}}}$, it is necessary to obtain a map that computes the value of z given $\mathrm{Exp}_{\overline{\mathcal{W}}}(z)$, i.e. the map

$$\begin{aligned} \mathrm{Log}_{\overline{\mathcal{W}}} : \mathrm{Im}(\mathrm{Exp}_{\overline{\mathcal{W}}}) &\longrightarrow \mathbb{Z}/p^k\mathbb{Z} \\ \mathrm{Exp}_{\overline{\mathcal{W}}}(z) &\longmapsto z. \end{aligned} \tag{8.12}$$

Determining the proper value of z is a complex task in this case. Despite its apparent simplicity, the module makes it impossible to determine z by starting with the definition of the exponential function and proceeding to use the series expansion of the Weierstrass $\wp$-function and its derivative. This task becomes even more challenging when attempting to compute z using the inverse function of the Weierstrass $\wp$-function. Unlike the Weierstrass $\wp$-function, there is no direct expression that allows for the algorithmically determination of the series expansion of $\wp^{-1}(z')$. Furthermore, the expansion of $\wp^{-1}(z')$ in Puiseux series

$$\wp^{-1}(z') = z'^{-\frac{1}{2}} + \frac{1}{40}g_2 z'^{-\frac{5}{2}} + \frac{1}{56}g_3 z'^{-\frac{7}{2}} + \cdots \tag{8.13}$$

involves square roots, which raises additional difficulties. The use of square roots in this context raises a number of questions and issues. Unlike the square root function in the field of real numbers, the square root modulo p^k may have more than two

solutions. For example, the square root of 0 modulo 3^5 is given by multiples of 27, i.e. $27h$, where $h = 0, 1, \ldots, 8$. Additionally, when working with powers, it is necessary to establish a specific order of operations. For instance, the expressions $(a^{\frac{1}{b}})^c$ and $(a^c)^{\frac{1}{b}}$ do not necessarily yield the same result $a^{\frac{b}{c}}$ modulo p^k. As an example, if we calculate $9^{\frac{5}{2}}$ modulo $3^4 = 81$, then we would have $9^{\frac{1}{2}} = \{3, 24, 30, 51, 57, 78\}$. If we raise each of these results to the fifth power, then we would get zero in each case. However, if we perform the operation $(9^5)^{\frac{1}{2}}$, then we would obtain all of the square roots of zero modulo 81, i.e. $\{0, 9, 18, 27, 36, 45, 64, 72\}$.

The task of computing the logarithm of a point $P \in \mathrm{Im}(\mathrm{Exp}_{\overline{\mathcal{W}}})$ is a challenging task. To this end, we attempted to compute the cubic root of the projective coordinate Z of P. Specifically, we have that $Z = tz^3$, where t is the least common multiple between $z^3\wp(z)$ and $\frac{z^3}{2}\wp'(z)$. In principle, this should allow us to compute the cubic root of Z in order to determine the value of z. However, we have seen that the least common multiple t may not have a cubic root modulo p^k, or that there may be more than one cubic root of z^3, which made these computations not successful.

As an alternative, we sought to approximate the value of z by starting with the value of z modulo p^5. Specifically, we used the series expansion of $\wp(z)$ and $\wp'(z)$ modulo p^5, taking only the first three terms of the series. By this method, we are able to obtain the value of z directly modulo p^5. In order to obtain the value of z modulo p^k, once z modulo p^5 is known, we can find $z \equiv z' \pmod{p^5}$ and thus $z = z' + hp^5$. For example, in order to get z modulo p^k, we would first calculate z modulo p^5 directly, then we would obtain the value of z modulo p^6, and so on, up to p^k.

8.4 ECDLP exploiting the map $\mathrm{Log}_{\overline{\mathcal{W}}}$

The $\mathrm{Log}_{\overline{\mathcal{W}}}$ is closely related to the discrete logarithm problem for elliptic curves. Consider the elliptic curve in Weierstrass form $\overline{\mathcal{W}}$ defined over the rational numbers, a known point Q and the generator point P of the curve. The goal is to compute the value of h that satisfies the expression $Q = h \otimes P$. We assume that P is a generator of the group in order to perform reductions of h modulo the cardinality of the Jacobian. If $\mathrm{ord}\Big(\mathrm{Mod}_{p^k}(P)\Big) \neq \mathrm{card}\Big(\mathcal{J}_{\mathbb{Z}/p^k\mathbb{Z}}\big(\overline{\mathcal{W}}\big)\Big)$, then the computation should be done modulo $\mathrm{ord}\Big(\mathrm{Mod}_{p^k}(P)\Big)$.

A simple initial approach for computing h is to move from the curve over the rational numbers to the same curve over the field of p-adic numbers. This greatly simplifies the problem, as h will also be reduced with respect to the order of this curve (if a generator point P of the group is taken). The relationship $Q = h \otimes P$ always holds, regardless of whether it is computed in a field or in a ring. The only difference will be that the points will be reduced modulo p^k and h will be reduced with respect to the order of the curve modulo p^k. We then have the relationship $h \equiv h' \pmod{t}$, with $t = \mathrm{ord}(\mathrm{Mod}_p(P)) = \mathrm{card}(\mathcal{J}_{\mathbb{Z}/p\mathbb{Z}}(\mathcal{C}))$. If the curve, reduced modulo p, has a "small" order, then it is possible to easily find h' and check if $h' \otimes P = Q$ over the rational numbers. If so, then the search for h is complete, otherwise we should calculate $h \equiv h'' \left(\mathrm{mod}\ \mathrm{card}\Big(\mathcal{J}_{\mathbb{Z}/p^2\mathbb{Z}}(\mathcal{C})\Big)\right)$. Since $\mathrm{card}\Big(\mathcal{J}_{\mathbb{Z}/p^k\mathbb{Z}}(\mathcal{C})\Big) = t \cdot p^{k-1}$, then we have that $h \equiv h'' \left(\mathrm{mod}\ p^{k-1} \cdot t\right)$. Furthermore, since h' is already known from the previous computation, it follows that $h'' = h' + m \cdot t$, for some $m \in \mathbb{Z}$. We can then proceed in this manner until we find that $h \equiv r \left(\mathrm{mod}\ p^k\right)$ such that $Q = r \otimes P = h \otimes P$ over the rational numbers.

In the following, we present a practical example.

Example 2. *Let $\overline{\mathcal{W}}$ be an elliptic curve defined over $\mathbb{Q}$ by the equation $y^2 = x^3 - x + \frac{1}{4}$. We assume that the relationship $Q = 31 \otimes P$ holds true, where $P = \left(2, \frac{5}{2}\right)$ and Q are points belonging to $\overline{\mathcal{W}}(\mathbb{Q})$ and $P - \Omega$ is the generator of the Jacobian. For the purpose of simplifying the notation in this example, we fix the prime number $p = 3$. We reduce the curve $\overline{\mathcal{W}}$ over $\mathbb{Z}/3\mathbb{Z}$ and we get $y^2 = x^3 - x + 1$. The Jacobian of this latter curve contains exactly 7 points, including the infinity point Ω, and thus constitutes a cyclic group of order 7. The goal is to determine the value of h, which is such that $Q = h \otimes P$. However, it should be noted that the value of h is unknown, and it is only known that $Q = h \otimes P$ for some $h \in \mathbb{Z}$.*

In order to compute the exact value of h, several steps are taken. First, the points P and Q are reduced over $\mathbb{Z}/3\mathbb{Z}$, and let $\text{Mod}_3(P)$ and $\text{Mod}_3(Q)$ be their respective reductions. Since $\mathcal{J}_{\mathbb{Z}/3\mathbb{Z}}\left(\overline{\mathcal{W}}\right)$ has an order of 7, it follows that $\text{Mod}_3(Q) = 3 \otimes \text{Mod}_3(P)$ over $\mathbb{Z}/3\mathbb{Z}$, as $31 \equiv 3 \pmod 7$. However, if we were to calculate $3 \otimes P$ over $\mathbb{Q}$, the result would not be equal to the point Q. Therefore, further computation is required.

Next, we calculate $\overline{\mathcal{W}}(\mathbb{Z}/3^2\mathbb{Z})$, which is defined by the equation $y^2 = x^3 - x + 7$ (as $\frac{1}{4} \equiv 7 \pmod 9$) and reduce the points P and Q over $\mathbb{Z}/3^2\mathbb{Z}$. It was previously determined that $h \equiv 3 \pmod 7$, and we already know that the cardinality of $\mathcal{J}_{\mathbb{Z}/3^2\mathbb{Z}}\left(\overline{\mathcal{W}}\right)$ is equal to 21. This allows us to avoid calculating all the possible sums of the form $t \otimes \text{Mod}_{3^2}(P)$, where $t = 3 + 7l$, and only compute the sums where $t = 3, 10, 17$. Additionally, since we have already verified that $3 \otimes P$ is not equal to Q over $\mathbb{Q}$, we can avoid this computation. Furthermore, knowing that the cardinality of $\mathcal{J}_{\mathbb{Z}/3^2\mathbb{Z}}\left(\overline{\mathcal{W}}\right)$ is 21, we can determine the value of h modulo 21, in particular, we find that $h \equiv 10 \pmod{21}$. The process is not yet complete, but by moving to $\overline{\mathcal{W}}(\mathbb{Z}/3^3\mathbb{Z})$, we can deduce that $h = 31 = 10 + 21 \pmod{27}$, thus successfully completing the search.

8.4.1 Refinement of the ECDLP over $\mathbb{Q}_p$

In the previous section, a relatively simple method was presented for computing the discrete logarithm of a point that belongs to a curve defined over $\mathbb{Q}$, by using the same elliptic curve over $\mathbb{Q}_p$ with p as the prime number.

This section illustrates a more advanced procedure which can determine the discrete logarithm under the same conditions. We address the reader to appendix A.1 and appendix B.1 for, respectively, the pseudocodes and the implementation codes of the algorithms used to compute the ECDLP. Additionally, in the latter appendix, we show the results of our implementation.

Specifically, we can use the map $\text{Exp}_{\overline{\mathcal{W}}}$ and its inverse $\text{Log}_{\overline{\mathcal{W}}}$ to determine the value of h such that $Q = h \otimes P$ over $\mathbb{Q}$. One can see that $Q \ominus (h \otimes P) = \Omega$ over $\mathbb{Q}$. Therefore,

$$Q \ominus (h \otimes P) \in \text{Im}(\text{Exp}_{\overline{\mathcal{W}}}).$$

Furthermore, over $\mathbb{Z}/p\mathbb{Z}$, we may reduce h modulo $t = \text{ord}(\text{Mod}_p(P - \Omega))$, i.e. $h \equiv \overline{h} \pmod t$. Thus, we have that $h = \overline{h} + nt$, for some $n \in \mathbb{Z}$. We can therefore express that $Q = h \otimes P = (\overline{h} + nt) \otimes P$, and thus $Q \ominus (\overline{h} \otimes P) = nt \otimes P$. Additionally,

$t \otimes P = \mathrm{Exp}_{\overline{\mathcal{W}}}(p^c m)$ for some $m \in \mathbb{Z}$, and thus

$$nt \otimes \mathrm{Mod}_{p^k}(P) = n \otimes \mathrm{Exp}_{\overline{\mathcal{W}}}(p^c m) = \mathrm{Exp}_{\overline{\mathcal{W}}}(p^c nm),$$

by definition of $\mathrm{Exp}_{\overline{\mathcal{W}}}$. As a result, we have that

$$\mathrm{Mod}_{p^k}\Big(Q \ominus (\overline{h} \otimes P)\Big) = nt \otimes \mathrm{Mod}_{p^k}(P).$$

In conclusion, if p is a small prime number, then $\overline{h}$ can be computed immediately over $\mathbb{Z}/p\mathbb{Z}$, and the value of t can be quickly determined as well. What is needed is the value of n. Thus, the points P and Q can be reduced over $\mathbb{Z}/p^k\mathbb{Z}$, and the relationship $\mathrm{Mod}_{p^k}\Big(Q \ominus (\overline{h} \otimes P)\Big) = nt \otimes \mathrm{Mod}_{p^k}(P)$ can be derived.

By taking the logarithm, we have that

$$\mathrm{Log}_{\overline{\mathcal{W}}}\Big(\mathrm{Mod}_{p^k}\Big(Q \ominus (\overline{h} \otimes P)\Big)\Big) = \mathrm{Log}_{\overline{\mathcal{W}}}\Big(nt \otimes \mathrm{Mod}_{p^k}(P)\Big).$$

From the left side of this latter equation, it follows that

$$\mathrm{Log}_{\overline{\mathcal{W}}}\Big(\mathrm{Mod}_{p^k}\Big(Q \ominus (\overline{h} \otimes P)\Big)\Big) = \mathrm{Log}_{\overline{\mathcal{W}}}\Big(\mathrm{Exp}_{\overline{\mathcal{W}}}\Big(p^d l\Big)\Big) = p^d l,$$

for some $l \in \mathbb{Z}$ and $1 \leq d \in \mathbb{N}$, while from the right side, we have that

$$\mathrm{Log}_{\overline{\mathcal{W}}}\Big(nt \otimes \mathrm{Mod}_{p^k}(P)\Big) = \mathrm{Log}_{\overline{\mathcal{W}}}(\mathrm{Exp}_{\overline{\mathcal{W}}}(p^c nm)) = p^c nm,$$

for some $m \in \mathbb{Z}$ and $1 \leq c \in \mathbb{N}$. Thus, we get at the expression

$$p^d l \equiv p^c nm \pmod{p^k},$$

hence, if $\frac{l}{m}$ is a unit of $\mathbb{Z}/p^k\mathbb{Z}$, then we obtain that

$$n \equiv p^{d-c}\frac{l}{m} \pmod{p^{k-c}}.$$

Special attention should be given to the final expression as the presence of p^c and p^d reduces the convergence radius, as n will only be found modulo p^{k-c}. Typically, $c = 1$, which implies that the solution can be found modulo p^{k-1}, despite the fact that the expressions were calculated modulo p^k. Below, we restate example 2 using the method just described.

Example 3. *Let $\overline{\mathcal{W}}$ be an elliptic curve in Weierstrass form defined over $\mathbb{Q}$ by the equation $y^2 = x^3 - x + \frac{1}{4}$ and let $P = \left(2, \frac{5}{2}\right)$ be a point belonging to $\overline{\mathcal{W}}(\mathbb{Q})$. The task is to solve the ECDLP for $Q = h \otimes P$, where h is an unknown integer. However, for the purposes of this demonstration, suppose we know that $h = 31$ and we aim to verify the efficacy of the algorithm discussed above. In table 8.1, we present the results of the aforementioned algorithms, where we exploit an approximation modulo p^k, with $p = 3$.*

As previously stated, we first calculate $\mathrm{Log}_{\overline{\mathcal{W}}}(t \otimes P) = \mathrm{Log}_{\overline{\mathcal{W}}}(7 \otimes P)$, where $t = \mathrm{card}(\mathcal{J}_{\mathbb{Z}/p\mathbb{Z}}(\mathcal{C})) = 7$ can be easily computed. Furthermore, we can compute h over $\overline{\mathcal{W}}(\mathbb{Z}/p\mathbb{Z})$ by calculating $h \pmod{t}$ and obtaining $\overline{h} = 3$, in order to compute $Q \ominus (\overline{h} \otimes P)$. Next, we can compute the logarithm of $7 \otimes P$ and $Q \ominus (3 \otimes P)$ modulo 3^k.

These two points belong to $\mathrm{Im}(\mathrm{Exp}_{\overline{\mathcal{W}}})$*, thus, their logarithm values can be easily computed using successive approximations and by exploiting the p-adic series expansions of* $\wp$ *and its derivative (see section 8.3). Additionally, we know that these logarithms are multiples of* $p = 3$*, that is,* $\mathrm{Log}_{\overline{\mathcal{W}}}(t \otimes P) = 3l_1$ *and* $\mathrm{Log}_{\overline{\mathcal{W}}}(Q \ominus (3 \otimes P)) = 3l_2$ *for some* $l_1, l_2 \in \mathbb{Z}$*. Since these two values have to be equal modulo* $p^k = 3^k$*, their ratio modulo* 3^k *yields the unknown* n *such that* $h = \overline{h} + nt$*. This is easy to verify since* $Q = h \otimes P$*, but* $h = \overline{h} + nt$*, thus* $(tn) \otimes P \equiv Q \ominus (\overline{h} \otimes P)$ *over* $\mathbb{Z}/p^k\mathbb{Z}$*. Therefore,* $\mathrm{Log}_{\overline{\mathcal{W}}}((tn) \otimes P) \equiv \mathrm{Log}_{\overline{\mathcal{W}}}\big(Q \ominus (\overline{h} \otimes P)\big)$ *over* $\mathbb{Z}/p^k\mathbb{Z}$ *from which it follows that* $pnl_1 \equiv pl_2 \pmod{3^k}$*.*

We note that this latter equivalence gives us $n \equiv \frac{l_2}{l_1} \pmod{3^{k-1}}$*. Specifically, we found the first valid* n *modulo* 3^3 *as the curve* $\overline{\mathcal{W}}$ *over* $\mathbb{Z}/3^3\mathbb{Z}$ *is the first curve whose Jacobian has a cardinality greater than* h*, thus here we have found the exact value of* h*. Therefore, we only need to compute the two logarithms of* $t \otimes P$ *and* $Q \ominus (\overline{h} \otimes P)$ *for* $\lfloor \log_3(h) \rfloor = 3$ *times. Therefore, the cost of this algorithm is* $\mathcal{O}(\lfloor \log_3(h) \rfloor \cdot r)$*, where* r *is the cost of computing the two logarithms at each step. It is worth mentioning that we assumed that* P *is a generator of* $\mathcal{J}_{\mathbb{Z}/p^k\mathbb{Z}}(\overline{\mathcal{W}})$*, thus we computed all the previous expressions modulo the cardinality of the Jacobian. If* P *were not a generator, then the computations would have to be done modulo the order of* $\mathrm{Mod}_{p^k}(P)$*.*

k	$\mathrm{Log}(7 \otimes P)$	$\mathrm{Log}(Q \ominus (3 \otimes P))$	n	$h = 3 + 7n$
1	3	0	0	3
2	6	6	1	10
3	6	24	4	31
4	33	51	4	31
5	195	51	4	31
6	681	537	4	31
7	2139	1995	4	31
8	6513	6369	4	31
9	13074	12930	4	31
10	13074	52296	4	31

TABLE 8.1: ECDLP as successive approximations modulo 3^k

8.5 ECDLP from $\mathbb{Q}$ to $\mathbb{Q}_p$

In section 8.4, we discussed the discrete logarithm problem and presented an elegant and efficient method to deal with it in subsec. 8.4.1.

From a more abstract perspective, we can consider an elliptic curve in Weierstrass form, denoted by $\mathcal{W}$, defined over the field of rational numbers. We can also consider the relationship $\alpha = a \otimes \gamma$, where $\alpha, \gamma \in \mathcal{W}(\mathbb{Q})$, and a is an integer. This relationship, being defined over $\mathbb{Q}$, remains valid as well when the map $\mathrm{Mod}_{\mathcal{W}}$ is

applied to it. In other words, if we define $\mathrm{Mod}_{p^k}(\alpha)$ and $\mathrm{Mod}_{p^k}(\gamma)$ to be the reduction modulo p^k of α and γ, respectively, then $\mathrm{Mod}_{p^k}(\alpha) = a' \otimes \mathrm{Mod}_{p^k}(\gamma)$, where $a \equiv a' \left(\bmod \operatorname{ord}\left(\mathrm{Mod}_{p^k}(\gamma)\right)\right)$. Therefore, if we have an elliptic curve in Weierstrass form $\mathcal{W}$ over $\mathbb{Q}$, and we reduce $\mathcal{W}$ modulo p^k, any relationship of the form $\alpha = a \otimes \gamma$ over $\mathbb{Q}$ will still hold in the reduced curve, with the appropriate modifications. The diagram below illustrates the concepts discussed previously:

$$
\begin{array}{ccc}
\mathcal{W}(\mathbb{Q}) & \xrightarrow{\mathrm{Mod}_{\mathcal{W}}} & \mathcal{W}(\mathbb{F}_q) \\
\Big\downarrow \mathrm{Mod}_{\mathcal{W}} & & \\
\mathcal{W}(\mathbb{F}_p) & &
\end{array}
$$

where p and q are prime numbers such that $q \gg p$. In this diagram, the top arrow represents the reduction modulo q, while the bottom arrow represents the reduction modulo p.

Suppose for the moment that we have an elliptic curve in Weierstrass form over $\mathbb{Q}$, which when reduced modulo a very large prime q, gives us one of the well-known curves currently used in cryptography, such as the curve Curve25519, in which $q = 2^{255} - 19$. At this point, we would like to solve the ECDLP for Curve25519, but we know that it is computationally infeasible to do so directly.

Since we know $\mathcal{W}(\mathbb{Q})$, in principle, we can compute the points α and γ over $\mathbb{Q}$, which, when reduced modulo q, give us the respective points over Curve25519. Now, we compute a third curve over a field with a very small prime characteristic, preferably 3, by reducing $\mathcal{W}(\mathbb{Q})$ modulo this small prime. For instance, we could obtain the elliptic curve given by the equation $y^2 = x^3 - x + 1$, which has exactly 7 points. Next, exploiting the method presented in subsec. 8.4.1, we can first obtain the value of a modulo 7. Subsequently, through an iterative process, we obtain the curve $\mathcal{W}$ reduced modulo 3^k, and we exploit this reduction in order to obtain the true value of a over $\mathbb{Q}$.

We already know, as we stated in chapter 8, that the elliptic curve, given by the equation $y^2 = x^3 - x + 1$, reduced modulo 3^k will have a cardinality equal to $7 \cdot 3^{k-1}$. Therefore, it is not difficult to understand that, from 3^2 onwards, it is necessary to perform at most two operations of the form $n \otimes P$, where n is an integer and $P \in \mathcal{J}_{\mathbb{Z}/3^k\mathbb{Z}}(\mathcal{W})$ in order to determine the solution of the ECDLP.

Assuming that $a \equiv c \pmod 7$, we know that the elliptic curve reduced modulo 3^2 will have 21 points. From the previously established relationship, we know that $a \equiv c \pmod 7$. Therefore, if the value of a does not satisfy the relationship $\alpha = a \otimes \gamma$ over $\mathbb{Q}$, then we should only verify if either $a = c + 7$ or $a = c + 14$ satisfy the relationship $\alpha = a \otimes \gamma$ over $\mathbb{Q}$. Similarly, there will be at most two other products $n \otimes P$ on the other curves reduced modulo 3^k, where $k > 2$. Hence, with these elliptic curves, we can solve the ECDLP with only $\mathcal{O}(\log_3(a))$ products of points $n \otimes P$ over elliptic curves reduced modulo 3^k.

In practice, the original elliptic curve over $\mathbb{Q}$ and the original relationship are not typically available. One only has the elliptic curve over some finite field (such as Curve25519), a relationship between two points belonging to that curve, and wants to solve the ECDLP for that relationship.

In principle, it would also be possible to derive the common ancestor of $\mathcal{W}$ over $\mathbb{Q}$, starting from the curve $\mathcal{W}$ over $\mathbb{F}_q$, but it is more complex. It turns out that if we have the relationship $\alpha = a \otimes \gamma$ over $\mathbb{F}_q$, then this relationship over $\mathbb{Q}$ becomes $\alpha' = (a' \otimes \gamma') + \omega$, where ω is a point belonging to $\mathcal{W}(\mathbb{Q})$ which reduced modulo q is equal to Ω.

9 Hyperelliptic curves over $\mathbb{Q}_p$

In order to simplify the notation in this chapter, we use the notation $\mathcal{H}_k$ to refer to $\mathcal{H}\Big(\mathbb{Z}/p^k\mathbb{Z}\Big)$, and $\mathcal{J}_k(\mathcal{H})$ to refer to the Jacobian of the hyperelliptic curve $\mathcal{H}$ over $\mathbb{Z}/p^k\mathbb{Z}$.

Based on the results obtained in the case of elliptic curves over $\mathbb{Z}/p^k\mathbb{Z}$, we have expanded our study to include more complex abelian varieties such as hyperelliptic curves (see chapter 4). Specifically, we aim to demonstrate that the Jacobian $\mathcal{J}(\mathcal{H})$ of a hyperelliptic curve $\mathcal{H}$ over $\mathbb{Z}/p^k\mathbb{Z}$ has a structure that is similar to that found in elliptic curves, that is, $\mathcal{J}_k(\mathcal{H}) \cong \mathcal{J}_1(\mathcal{H}) \oplus \mathbb{Z}/p^{g(k-1)}\mathbb{Z}$.

In order to prove this result, we must accurately count the number of divisors in $\mathcal{J}_k(\mathcal{H})$. Although this task may appear simple, it is quite complex.

In order to illustrate this point, we will consider a trivial example, namely, a hyperelliptic curve $\mathcal{H}$ of genus $g = 2$. In this case, the divisors can take only the following forms: $D = \Omega - \Omega$, $D = P - \Omega$, and $D = P + Q - 2 \cdot \Omega$. Assume we count all the n points belonging to $\mathcal{H}_k$, that is, we determine all the pairs $(x, y) \in \mathcal{H}_k$, where $x, y \in \mathbb{Z}/p^k\mathbb{Z}$.

As a result, the number of divisors of the type $D = P - \Omega$ is n, while the number of divisors of the type $D = P + Q - 2 \cdot \Omega$ is at most $\binom{n+1}{2}$, as P can be equal to Q. However, it is important to note that this calculation is not entirely accurate as divisors of the form $D = 2 \cdot P - 2 \cdot \Omega$ may not exist, or may be equivalent to other divisors, such as $D = Q - \Omega$, $D = \Omega - \Omega$, or $D = P' + Q' - 2 \cdot \Omega$.

Furthermore, by counting the divisors in the forms of $D = \Omega - \Omega$, $D = P - \Omega$, and $D = P + Q - 2 \cdot \Omega$ as $1 + n + \binom{n}{2}$ (without repetition) or $1 + n + \binom{n+1}{2}$ (with repetition), we obtain an incorrect value. According to a well-known result in the literature, e.g. in a classic book as [44], the 2-torsion subgroup of a hyperelliptic curve $\mathcal{H}$, defined as

$$\mathcal{H}[2] = \{D \in \mathcal{J}(\mathcal{H}) \colon 2 \otimes D = \Omega - \Omega\}, \tag{9.1}$$

is isomorphic to $\mathbb{Z}/2^{2g}\mathbb{Z}$. This implies that if the 2-torsion subgroup in $\mathcal{H}$ of genus $g = 2$ exists, then it must have a cardinality equal to $2^4 = 16$. However, our previous count does not yield a multiple of 16, even though the 2-torsion subgroup exists.

Thinking of divisors as formal sums of points belonging to the curve complicates this counting process because these points may not necessarily exist within the defining field $\mathbb{K}$ of $\mathcal{H}$, but they surely exist over an algebraic closure of $\mathbb{K}$. In light of this, we turned to the Mumford representation, which states that regardless of the genus of $\mathcal{H}$, a divisor is always determined by a pair of polynomials $(u(x), v(x))$. For example, if one were to operate on a curve of genus $g = 10$, then the classic representation of the divisor D would have to take into account all the at most 10 affine points that

belong to its support. On the other hand, with the Mumford representation, only a pair of polynomials is sufficient.

The Mumford representation of the principal divisors of a hyperelliptic curve simplifies the counting process, but it is still complex. By using the properties that $\deg(v(x)) < \deg(u(x)) \leq g$ and $u(x) \mid v(x)^2 + v(x)h(x) - f(x)$, it may appear possible to algorithmically determine the number of principal divisors in the Mumford form that belong to the Jacobian of the curve.

Consider again the case of $g = 2$, in which $u(x)$ is a polynomial of degree at most g and $v(x)$ is a polynomial of degree at most $g - 1$. For each $v(x) = ax + b$, where $a, b \in \mathbb{Z}/p^k\mathbb{Z}$, it is possible to determine the valid $u(x)$ polynomials by factoring the polynomial $v(x)^2 + v(x)h(x) - f(x)$ over $\mathbb{Z}/p^k\mathbb{Z}$ and taking the factors of degree n such that $\deg(v(x)) < n \leq g$. This process yields the divisors in the Mumford form $D = (u(x), v(x)) \in \mathcal{J}_k(\mathcal{H})$ that meet the criteria listed above. However, there are still some invalid divisors. These errors are ultimately due to a conjecture about the order of this Jacobian.

In order to further understand the structure of the Jacobian of hyperelliptic curves over $\mathbb{Z}/p^k\mathbb{Z}$, we make the conjecture that the cardinality of the Jacobian follows a similar pattern to that of elliptic curves. Specifically, we propose that for a hyperelliptic curve in Weierstrass form $\mathcal{H}$ of genus g over $\mathbb{Z}/p^k\mathbb{Z}$, the cardinality of the Jacobian is equal to:

$$\operatorname{card}\Big(\mathcal{J}_{\mathbb{Z}/p^k\mathbb{Z}}(\mathcal{H})\Big) = \operatorname{card}(\mathcal{J}_{\mathbb{Z}/p\mathbb{Z}}(\mathcal{H})) \cdot p^{g(k-1)}. \tag{9.2}$$

This conjecture is based on the observation that for an elliptic curve in Weierstrass form $\overline{\mathcal{W}}$ over $\mathbb{Z}/p^k\mathbb{Z}$, the cardinality of its Jacobian is given by:

$$\operatorname{card}\Big(\mathcal{J}_{\mathbb{Z}/p^k\mathbb{Z}}\big(\overline{\mathcal{W}}\big)\Big) = \operatorname{card}\Big(\mathcal{J}_{\mathbb{Z}/p\mathbb{Z}}\big(\overline{\mathcal{W}}\big)\Big) \cdot p^{k-1} \tag{9.3}$$

This idea would need further exploration and proof before it can be considered a proven results.

Determining all valid divisors of a hyperelliptic curve can be a complex process. One approach is to use the Mumford representation of the principal divisors of a curve, which simplifies the count compared to using formal sums of points belonging to the curve. However, this method still poses several challenges. First, Mumford's representation only considers divisors consisting of proper points $P = (x, y)$, where $x, y \in \overline{\mathbb{K}}$. In the case of elliptic curves over $\mathbb{Q}_p$, there are also "improper" points $P = [Z : X : Y]$ whose reduction modulo p is the point at infinity Ω. The procedure described above can only be applied to count divisors with proper points, i.e., points whose reduction modulo p is not Ω. However, divisors that include improper points or a combination of proper and improper points must also be taken into account.

Second, factoring the polynomial $v(x)^2 + v(x)h(x) - f(x)$ over $\mathbb{Z}/p^k\mathbb{Z}$ (a ring) is not always possible as current algorithms only work when the discriminant of the polynomial to factorize is non-zero. In cases where the discriminant is zero, it is possible to factor the polynomial over $\mathbb{Q}_p$ and determine the factors over $\mathbb{Z}/p^k\mathbb{Z}$ by means of the equivalent factors over $\mathbb{Q}_p$. However, this complicates the process further, as factors may not exist over $\mathbb{Z}/p^k\mathbb{Z}$.

Finally, a non-trivial issue is the ability to perform addition of divisors in the form of Mumford. Specifically, with the Cantor-Koblitz algorithm (algorithm 1), one can

determine the divisor sum $D = D_1 \oplus D_2$. Then, once a divisor $D = (u(x), v(x))$ has been determined by factoring $v(x)^2 + v(x)h(x) - f(x)$ over $\mathbb{Z}_{p^k}$, one computes $n \otimes D$, for increasing $n \in \mathbb{N}$, until either $n \otimes D = \Omega - \Omega$ or there are any errors. If there are errors, then the divisor would be considered invalid.

However, all these techniques do not yet yield the expected results or, in any case, a valid number of divisors for the curve $\mathcal{H}$ over $\mathbb{Z}_{p^k}$. Therefore, determining the exact number of divisors belonging to the Jacobian of $\mathcal{H}$ over $\mathbb{Z}_{p^k}$ remains an open problem. It is likely that the group we are searching for is a subgroup of $\mathcal{J}_k(\mathcal{H})$ such that the divisors, expressed in Mumford form, are formed by points that at most have coordinates in a quadratic extension of the defining field of $\mathcal{H}$. For example, as noted in the case of $g = 2$, the polynomial $u(x)$ could be irreducible over the defining field $\mathbb{K}$ of the curve $\mathcal{H}$ but it may have roots in a quadratic extension of $\mathbb{K}$.

9.1 "Weaknesses" of Cantor-Koblitz algorithm with hyperelliptic curves over $\mathbb{Q}_p$

The Cantor-Koblitz algorithm is applicable to any hyperelliptic curve of genus $g > 0$ over any field, and even over a ring, with the necessary precautions. It should be noted that any fraction of the form $\frac{a}{b}$ over $\mathbb{Z}/p^k\mathbb{Z}$, where $p \mid b$, is not invertible and must be treated accordingly.

The Cantor-Koblitz algorithm has two operations that can be problematic when applied to rings: the gcd and the polynomial modulo operation. Both of these operations rely on division between polynomials, which can be infeasible when one of the polynomials has a multiple of p among its coefficients. As a result, performing the gcd and polynomial modulo operation with this multiple of p could lead to a division between coefficients where the denominator is also a multiple of p, making the computation impractical.

One potential solution to this issue is to operate on the same curve over $\mathbb{Q}$ instead of $\mathbb{Z}/p^k\mathbb{Z}$, since the gcd and polynomial modulo operations will always be possible in this case. However, one may wonder how to move from $\mathbb{Q}$ to $\mathbb{Z}/p^k\mathbb{Z}$, or if the map Mod_{p^k} is sufficient for this purpose. In particular, one may wonder if $D = (u(x), v(x))$ is a divisor of the Jacobian of $\mathcal{H}$ over $\mathbb{Q}$, then does $\mathrm{Mod}_{p^k}(D) = (u(x) \pmod{p^k}, v(x) \pmod{p^k})$ represent a valid divisor for the Jacobian of $\mathcal{H}$ over $\mathbb{Z}/p^k\mathbb{Z}$. Additionally, one may question whether two divisors represent the same object. However, in general, the answer is no, as demonstrated in the following example.

Example 4. *Suppose that $k = 1$ and let $\mathcal{H}$ be the hyperelliptic curve in Weierstrass form of genus $g = 2$ over $\mathbb{Q}$ defined by the equation:*

$$y^2 = x(x-1)(x+2)(x-3)(x+4).$$

Consider the divisor $(x - 4, 24) = D \in \mathcal{J}(\mathcal{H})$. Also, suppose we are interested in finding the order of D on the same curve but reduced over $\mathbb{Z}/11\mathbb{Z}$. We will find that the order of D, reduced modulo 11, is 16. Now, suppose we calculate the divisors $2 \otimes D$, $4 \otimes D$, $8 \otimes D$, and $16 \otimes D$ over both $\mathbb{Q}$ and $\mathbb{Z}/11\mathbb{Z}$, and then we compare these results. In particular, given $h \otimes D$ over $\mathbb{Q}$, we compare $\mathrm{Mod}_p(h \otimes D)$ with $h \otimes \mathrm{Mod}_p(D)$, that is, the divisor obtained by making the same computation but directly modulo 11. We will observe that, from a certain h onwards, these divisors will be

different. For instance, $\mathrm{Mod}_p(8 \otimes D) = (x+10, 4x+7)$, *while* $8 \otimes \mathrm{Mod}_p(D) = (x+10, 0)$. *The second result is the correct one because if* D *has order* 16 *over* $\mathbb{Z}/11\mathbb{Z}$, *then* $8 \otimes D$ *has order* 2 *and, therefore, is a divisor whose points in its support lie on the* x*-axis. Let us now consider the same scenario over* $\mathbb{Q}$*. We observe that* $4x+7 = 4(x+10)$ *modulo* 11*. Furthermore, the cubic, passing through the points belonging to* $\mathrm{supp}(4 \otimes D)$ *(used for the computation of* $8 \otimes D$ *as* $(4 \otimes D) \oplus (4 \otimes D)$*), can be factored modulo* 11 *as* $(x+3)(x+10)$*. Thus, this result suggests that it was necessary to further execute the loop between lines* 8 *and* 11 *of the Cantor-Koblitz algorithm (algorithm 1) as the divisor* $\mathrm{Mod}_p(h \otimes D)$ *might not be reduced.*

The method of reducing the divisors using the previously defined map Mod_p *alone is not sufficient to establish equivalence between the two methods. For example, consider the divisor* $\mathrm{Mod}_p(16 \otimes D) = (x^2, 8x)$ *and compare it to* $16 \otimes \mathrm{Mod}_p(D) = (1, 0)$ *(the identity divisor* $\Omega - \Omega$ *in Mumford form). It is worth to note that* $\mathrm{Mod}_p(16 \otimes D)$ *is not a valid divisor in Mumford form since* $u(x)$ *does not divide* $(v^2(x) + v(x)h(x) - f(x))$*. More precisely,* $\mathrm{Mod}_p(16 \otimes D)$ *should be equal to* $2 \cdot (0,0) - 2 \cdot \Omega$*, but this latter divisor is not valid since any divisor represented in Mumford form is such that* $\gcd(u(x), u'(x), v(x)) = 1$ *(see theorem 4.2), i.e., a point lying in the* x*-axis cannot have multiplicity greater than* 1 *in the support of a divisor belonging to the Jacobian of a hyperelliptic curve. Additionally, the divisor* $\mathrm{Mod}_p(8 \otimes D)$ *is not a valid Mumford representation since* $\deg(v(x)) = \deg(u(x))$*.*

The above example highlights that the following diagram is not commutative:

$$\begin{array}{ccc} \mathcal{J}_{\mathbb{Q}}(\mathcal{H}) \times \mathcal{J}_{\mathbb{Q}}(\mathcal{H}) & \xrightarrow{\oplus} & \mathcal{J}_{\mathbb{Q}}(\mathcal{H}) \\ \downarrow \mathrm{Mod}_{p^k} \quad \downarrow \mathrm{Mod}_{p^k} & & \downarrow \mathrm{Mod}_{p^k} \\ \mathcal{J}_k(\mathcal{H}) \times \mathcal{J}_k(\mathcal{H}) & \xrightarrow{\oplus} & \mathcal{J}_k(\mathcal{H}) \end{array}$$

Specifically, when reducing elements in $\mathcal{J}_{\mathbb{Q}}(\mathcal{H})$ over $\mathbb{Z}/p^k\mathbb{Z}$, the sum of those elements may yield a divisor that does not exist (such as the divisor $\mathrm{Mod}_p(16 \otimes D)$ in example 4) or one that does not belong to the Jacobian of the curve over $\mathbb{Z}/p^k\mathbb{Z}$. This is due to the fact that when p divides the coefficients of the polynomials $f_1(x)$ or $f_2(x)$, we have:

$$\gcd(f_1(x), f_2(x)) \left(\mathrm{mod}\ p^k\right) \not\equiv \gcd(f_1(x) \left(\mathrm{mod}\ p^k\right), f_2(x) \left(\mathrm{mod}\ p^k\right)),$$

so that it is not always possible to apply the reduction while maintaining the equivalence.

It should be noted that, in lines 1-2 of the Cantor-Koblitz algorithm, one computes the gcd between the polynomials $u_1(x)$ and $u_2(x)$ of the Mumford form of the divisors to be summed, respectively. However, this computation may differ if one considers the two divisors over $\mathbb{Q}$ and their respective reductions over $\mathbb{Z}/p^k\mathbb{Z}$. For example, consider summing the two divisors $\mathrm{Mod}_{11}(15 \otimes D) = (x+7, 9)$ and $\mathrm{Mod}_{11}(D) = (x+7, 2)$, where D is the divisor considered in example 4.

On one hand, over $\mathbb{Z}/11\mathbb{Z}$, the gcd computation in line 1 of the algorithm gives us $d_1 = x+7$. On the other hand, since $15 \otimes D$ over $\mathbb{Q}$ is equal to

$$15 \otimes D = \left(x^2 + \frac{5329753222...1306367}{14623495...57664}x + \frac{1676245790...05761}{365587392...3164416}, \frac{108340762895...9428583233}{559212413009...39411712}x + \frac{104257378356...660863103}{1398031032...813609852928}\right),$$

which is a divisor whose coefficients have numerators and denominators with more than 200 digits, and since the polynomial $u(x)$ in $15 \otimes D$ does not have roots over $\mathbb{Q}$, the gcd at line 1 of the algorithm give us $d_1 = 1$, which is not equivalent to $d_1 = x - 7$ over $\mathbb{Z}/11\mathbb{Z}$.

In conclusion, this implies that the previous diagram does not commute and that the composition between the map Mod_{p^k} and the gcd computation is not always commutative. As a result, it is not possible to extend the results in chapter 8 to the case of hyperelliptic curves of genus $g > 1$.

10 Edwards curves over $\mathbb{Q}_p$

In this chapter, we extend the results of chapter 8 on elliptic curves in Weierstrass form over $\mathbb{Z}/p^k\mathbb{Z}$ to the case of Edwards curves. Specifically, we examine the properties and characteristics of Edwards curves, which are a specific type of (non-smooth) curves that have been widely studied in recent years due to their efficiency in certain cryptographic constructions. We address the reader to section 3.3 for further details about Edwards curves.

Furthermore, in the following sections, we first compute the map Exp for Edwards curves over a local field. Then, we will compute this map over the p-adic number field, which has significant implications for several number-theoretic and cryptographic applications.

Our motivations are based on an authoritative literature on the matter of lifting, which has been summarized by J. Silverman in [72]. In this survey, Silverman connects the lifting problem to the DLP for elliptic curves in Weierstrass form. Moreover, in [77] the authors defined a cryptosystem based on quotient groups of an elliptic curve in Weierstrass form over the p-adic number field, able to encrypt messages with variable lengths. This led to public-key cryptosystems with hierarchy management [78], which look interesting for their possible applications. Additionally, recently, similar topics have been investigated in [75], where the authors consider twisted Edwards curves over local fields and introduce a cryptosystem based on quotient groups of twisted Edwards curves over local fields.

For these reasons, although it is possible to extend the exponential map to other forms of elliptic curves (such as Legendre form, Jacobi form, Hessian form, Huff form), in this chapter we will focus only on the Edwards form.

10.1 The map Exp over local fields

In this section, we leverage the previously established results on elliptic curves in Weierstrass form (see also [33]) and extend them to the case of Edwards curves. As previously discussed, we are considering the scenario where there exists a birational equivalence between an elliptic curve $\mathcal{W}$ in Weierstrass form and an Edwards curve $\mathcal{E}$, such that the Jacobian $\mathcal{J}(\mathcal{W})$ of $\mathcal{W}$ is isomorphic to the Jacobian $\mathcal{J}^0(\mathcal{E})$ of $\mathcal{E}$ (as stated in theorem 3.9). This isomorphism applies to the subgroup of divisors of $\mathcal{J}(\mathcal{E})$ whose reduced form is $P - O$, where P is an affine point and $O = (0, 1)$ is taken as the neutral element of the group.

Let $\mathbb{K}$ be a local field, with $\mathcal{O}_{\mathbb{K}}$ being its ring of integers, $\mathfrak{m}_{\mathbb{K}}$ its prime ideal and $k = \mathcal{O}_{\mathbb{K}}/\mathfrak{m}_{\mathbb{K}}$ its residue field. We will take the image $\mathcal{J}_k^0(\mathcal{E})$ under the reduction modulo $\mathfrak{m}_{\mathbb{K}}$ of the group $\mathcal{J}_{\mathbb{K}}^0(\mathcal{E})$, and proceed to investigate under what conditions one has that $\mathcal{J}_{\mathbb{K}}^0(\mathcal{E}) \cong \mathcal{J}_k^0(\mathcal{E}) \oplus \mathfrak{m}_{\mathbb{K}}$.

Given the results presented in chapter 8, and under the assumption that $\mathcal{E}$ has a non-singular reduction modulo $\mathfrak{m}_\mathbb{K}$, we can establish the following theorem.

Theorem 10.1 (The map Exp for Edwards curves)**.** *Let $\mathbb{K}$ be a local field, $\mathcal{O}_\mathbb{K}$ is its ring of integers, and $\mathfrak{m}_\mathbb{K}$ is the prime ideal of $\mathcal{O}_\mathbb{K}$. If $\mathcal{E}$ is an Edwards curve as in theorem 3.9, that is, with $d \in \mathbb{K}$ a non-square, then the following map:*

$$\begin{aligned} \mathrm{Exp}_\mathcal{E} \colon \ \mathfrak{m}_\mathbb{K} &\longrightarrow \mathcal{J}^0_\mathbb{K}(\mathcal{E}) \\ z &\longmapsto \left(\frac{2}{3}\frac{y_1(3\wp(z)-a')}{x_1\wp'(z)}, \frac{3\wp(z)-a'-3x_1}{3\wp(z)-a'+3x_1}\right), \end{aligned}$$

where x_1, y_1 and a' are as in definition 3.13, is an exponential map for $\mathcal{E}$, that is, $\mathrm{Exp}_\mathcal{E}(z_1+z_2) = \mathrm{Exp}_\mathcal{E}(z_1) \oplus \mathrm{Exp}_\mathcal{E}(z_2)$.

Proof. Recall that, in definition 3.13, we have a birational equivalence of the Edwards curve $\mathcal{E}$ with elliptic curve in Weierstrass form $\mathcal{W}$ of equation $y^2 = x^3 + a'x^2 + b'x$, whereas the above map $\mathrm{Exp}_{\overline{\mathcal{W}}}$ is defined for elliptic curves in short Weierstrass form $\overline{\mathcal{W}}$ of equation $y^2 = x^3 + ax + b$.

However, we can apply the transformation $\chi \colon (x, y) \mapsto \left(x - \frac{a'}{3}, y\right)$ which, through the change of variables $\bar{x} = x - \frac{a'}{3}$, $\bar{y} = y$, changes the Weierstrass form $y^2 = x^3 + a'x^2 + b'x$ onto the short Weierstrass form $\bar{y}^2 = \bar{x}^3 + a\bar{x} + b$, that is, for any $P = (\bar{x}, \bar{y}) \in \overline{\mathcal{W}}(\mathbb{K})$, such that $\bar{y}^2 = \bar{x}^3 + a\bar{x} + b$, we have that $\chi(P) = P' \in \mathcal{W}(\mathbb{K})$. As $\chi(P) = P'$ belongs to $\mathcal{W}(\mathbb{K})$, we can now compute $\beta(P')$, where β in (3.19b), in order to get a point belonging to $\mathcal{E}(\mathbb{K})$. In particular, if $P = \mathrm{Exp}_{\overline{\mathcal{W}}}(z)$ for some $z \in \mathfrak{m}_\mathbb{K}$, then

$$\begin{aligned} \beta(\chi(P)) = \beta(\chi(\mathrm{Exp}_{\overline{\mathcal{W}}}(z))) &= \beta\left(\chi\left(\left[1 : \wp(z) : \frac{1}{2}\wp'(z)\right]\right)\right) = \\ &= \beta\left(\left[1 : \wp(z) - \frac{a'}{3} : \frac{1}{2}\wp'(z)\right]\right) = \\ &= \left(\frac{2}{3}\frac{y_1(3\wp(z)-a')}{x_1\wp'(z)}, \frac{3\wp(z)-a'-3x_1}{3\wp(z)-a'+3x_1}\right). \end{aligned}$$

Thus, the map $\mathrm{Exp}_\mathcal{E}$ for Edwards curves over the local field $\mathbb{K}$ is defined as $\mathrm{Exp}_\mathcal{E} = \beta \circ \chi \circ \mathrm{Exp}_{\overline{\mathcal{W}}}$, that is,

$$\begin{aligned} \mathrm{Exp}_\mathcal{E} \colon \mathfrak{m}_\mathbb{K} &\longrightarrow \mathcal{J}_\mathbb{K}(\mathcal{E}) \\ z &\longmapsto \mathrm{Exp}_\mathcal{E}(z) = \beta(\chi(\mathrm{Exp}_{\overline{\mathcal{W}}}(z))) = \left(\frac{2}{3}\frac{y_1(3\wp(z)-a')}{x_1\wp'(z)}, \frac{3\wp(z)-a'-3x_1}{3\wp(z)-a'+3x_1}\right). \end{aligned}$$

Note that $\chi(\Omega) = \Omega = [0 : 0 : 1]$ as the projective map χ maps $[Z : X : Y]$ onto the point $\left[Z : X - \frac{a'}{3}Z : Y\right]$, and thus we have that $\beta(\chi(\mathrm{Exp}_{\overline{\mathcal{W}}}(0))) = \beta(\chi(\Omega)) = \beta(\Omega) = O$.

Finally, we are left to prove that the map $\mathrm{Exp}_\mathcal{E} = \beta \circ \chi \circ \mathrm{Exp}_{\overline{\mathcal{W}}}$ is a one-to-one homomorphism of groups. On the one hand, the maps $\mathrm{Exp}_{\overline{\mathcal{W}}}$, β, and χ are one-to-one. Specifically, the map β here is bijective as d is not a square, and $\chi^{-1} \colon (\bar{x}, \bar{y}) \mapsto \left(x + \frac{a'}{3}, y\right)$.

On the other hand, $\mathrm{Exp}_{\mathcal{E}}$ is a homomorphism because $\mathrm{Exp}_{\overline{\mathcal{W}}}$ (see subsec. 3.1.2) and β (see remark 3.17) are homomorphisms, and χ is a translation, thus one has that:

$$\begin{aligned}\mathrm{Exp}_{\mathcal{E}}(z_1+z_2) &= \beta\circ\chi\circ\mathrm{Exp}_{\overline{\mathcal{W}}}(z_1+z_2) = \\ &= \beta\circ\chi\circ(\mathrm{Exp}_{\overline{\mathcal{W}}}(z_1)\oplus\mathrm{Exp}_{\overline{\mathcal{W}}}(z_2)) = \\ &= \beta\circ\chi\circ\mathrm{Exp}_{\overline{\mathcal{W}}}(z_1)\oplus\beta\circ\chi\circ\mathrm{Exp}_{\overline{\mathcal{W}}}(z_2) = \\ &= \mathrm{Exp}_{\mathcal{E}}(z_1)\oplus\mathrm{Exp}_{\mathcal{E}}(z_2).\end{aligned}$$

■

The following remark highlights the relationship between the curve $\mathcal{W}$ and its reduction $\overline{\mathcal{W}}$ under the transformation χ.

Remark 10.1. *Since χ transforms the curve $\mathcal{W}$ into the curve $\overline{\mathcal{W}}$, it holds that $\chi(P_1\oplus P_2)=\chi(P_1)\oplus\chi(P_2)$, where the left term uses the addition formula for $\mathcal{W}$, and the right term uses the addition formula for $\overline{\mathcal{W}}$.*

Corollary 10.2. *The following is a short exact sequence:*

$$0\longrightarrow\mathfrak{m}_{\mathbb{K}}\xrightarrow{\mathrm{Exp}_{\mathcal{E}}}\mathcal{J}^0_{\mathbb{K}}(\mathcal{E})\xrightarrow{\mathrm{Mod}_{\mathcal{E}}}\mathcal{J}^0_k(\mathcal{E})\longrightarrow 0. \tag{10.1}$$

Proof. The proof follows from the fact that, from theorem 10.1, $\mathrm{Exp}_{\mathcal{E}}$ is a monomorphism and $\mathrm{Mod}_{\mathcal{E}}$ is an epimorphism. Moreover, since, for any $z\in\mathfrak{m}_{\mathbb{K}}$, $\mathrm{Exp}_{\mathcal{E}}(z)=(\mathcal{O}(z^3),1+\mathcal{O}(z^3))$, then $\mathrm{Mod}_{\mathcal{E}}(\mathrm{Exp}_{\mathcal{E}}(z))=(0,1)$, and $\mathrm{Im}(\mathrm{Exp}_{\mathcal{E}})\subseteq\mathrm{Ker}(\mathrm{Mod}_{\mathcal{E}})$. Finally, together with $\mathrm{Exp}_{\overline{\mathcal{W}}}$, which is invertible by the equation (8.5), the map $\mathrm{Exp}_{\mathcal{E}}=\beta\circ\chi\circ\mathrm{Exp}_{\overline{\mathcal{W}}}$ is invertible for $z\in\mathfrak{m}_{\mathbb{K}}$, that is, one can write any point P in $\mathrm{Ker}(\mathrm{Mod}_{\mathcal{E}})$ as $P=\mathrm{Exp}_{\mathcal{E}}(z)$, for some $z\in\mathfrak{m}_{\mathbb{K}}$, thus $\mathrm{Ker}(\mathrm{Mod}_{\mathcal{E}})\subseteq\mathrm{Im}(\mathrm{Exp}_{\mathcal{E}})$. ■

In the following corollary, we prove that the exact sequence in corollary 10.2 splits if the Edwards curve taken into account is isomorphic to an elliptic curve in Weierstrass form which is non-anomalous.

The following corollary shows that the exact sequence in corollary 10.2 splits, provided that the Edwards curve in question is isomorphic to a non-anomalous elliptic curve in Weierstrass form. It is worth to note that splitting the exact sequence means that the group splits into a direct sum of subgroups and it is crucial in many mathematical and cryptographic applications.

Corollary 10.3. *If $k=\mathcal{O}_{\mathbb{K}}/\mathfrak{m}_{\mathbb{K}}$ is finite, $\mathcal{W}$ is not an anomalous curve, and $\mathcal{E}$ is the Edwards curve birational equivalent to $\mathcal{W}$, then $\mathcal{J}^0_{\mathbb{K}}(\mathcal{E})$ is isomorphic to $\mathcal{J}^0_k(\mathcal{E})\oplus\mathfrak{m}_{\mathbb{K}}$.*

Proof. Since we confined ourselves to the case in theorem 3.9, then we have that $\mathcal{J}^0_{\mathbb{K}}(\mathcal{E})\cong\mathcal{J}_{\mathbb{K}}(\mathcal{W})$, $\mathcal{J}^0_k(\mathcal{E})\cong\mathcal{J}_k(\mathcal{W})$, and the proof follows from theorem 8.1. ■

10.2 The map Exp over $\mathbb{Q}_p$

The objective of this section is to compute the map Exp for Edwards curves $\mathcal{E}$ over the field of p-adic numbers. Specifically, we impose the same assumptions and restrictions on the map $\mathrm{Exp}_{\mathcal{E}}$ as in section 8.1.

In this context, if $\mathcal{J}^0_k(\mathcal{E})$ denotes the image of the subgroup $\mathcal{J}^0(\mathcal{E})$ modulo p^k, then, by making similar modifications and observations as in section 8.1, the results in section 10.1 can be rephrased as follows: $\mathcal{J}^0_k(\mathcal{E})=\mathcal{J}^0_1(\mathcal{E})\oplus\mathrm{Im}(\mathrm{Exp}_{\mathcal{E}})$.

Since in theorem 10.1 we stated that $\mathrm{Exp}_{\mathcal{E}} = \beta \circ \chi \circ \mathrm{Exp}_{\overline{\mathcal{W}}}$, and since from equation (8.6) we have the representation of $\mathrm{Exp}_{\overline{\mathcal{W}}}$ whose domain is the inverse limit approximation of $\mathbb{Z}_p$, we have that the map $\mathrm{Exp}_{\mathcal{E}}$ over $\mathbb{Q}_p$ can be expressed as

As established in theorem 10.1, the map $\mathrm{Exp}_{\mathcal{E}}$ is equivalent to the composition of maps $\beta \circ \chi \circ \mathrm{Exp}_{\overline{\mathcal{W}}}$. Additionally, in equation (8.6), we provide a representation of the map $\mathrm{Exp}_{\overline{\mathcal{W}}}$ whose domain is an approximation of $\mathbb{Z}_p$ through an inverse limit. Therefore, the map $\mathrm{Exp}_{\mathcal{E}}$ over the field of p-adic numbers can be expressed as:

$$\begin{aligned} \mathrm{Exp}_{\mathcal{E}} \colon \; p\mathbb{Z}_p/p^k\mathbb{Z}_p &\longrightarrow \mathcal{E}\Big(\mathbb{Z}/p^k\mathbb{Z}\Big) \\ z = ph \qquad &\longmapsto \left(\frac{2}{3}\frac{y_1(3\wp(z) - a')}{x_1\wp'(z)}, \frac{3\wp(z) - a' - 3x_1}{3\wp(z) - a' + 3x_1}\right), \end{aligned}$$

where $h = 1, 2, \ldots, p^{k-1}$.

It is noteworthy, as stated in remark 8.1, that the domain of the map $\mathrm{Exp}_{\mathcal{E}}$ is the quotient group $p\mathbb{Z}_p/p^k\mathbb{Z}_p$. Furthermore, we have been established in remark 8.2 that this latter group is isomorphic to $\mathbb{Z}/p^{k-1}\mathbb{Z}$, and thus we can express the subgroup $\mathcal{J}^0(\mathcal{E})$ over this approximation as $\mathcal{J}^0_k(\mathcal{E}) = \mathcal{J}^0_1(\mathcal{E}) \oplus \mathbb{Z}/p^{k-1}\mathbb{Z}$.

As previously discussed in section section 8.1, the points belonging to the image of the map $\mathrm{Exp}_{\overline{\mathcal{W}}}$ are in the form $P = [ph_1 : ph_2 : h_3]$ with $p \nmid h_3$. Additionally, the point at infinity Ω is mapped through β to the neutral point O on the Edwards curve. As a result of this mapping, all points in the image of $\mathrm{Exp}_{\mathcal{E}}$ are equivalent modulo p to the neutral point O, and therefore are affine points. Therefore, if we count all the affine points (x, y) in $\mathcal{E}\Big(\mathbb{Z}/p^k\mathbb{Z}\Big)$, the number of these points will be equal to $\mathrm{card}(\mathcal{J}^0_1(\mathcal{E})) \cdot p^{k-1}$, where p^{k-1} is the cardinality of $\mathrm{Im}(\mathrm{Exp}_{\mathcal{E}})$.

Similar to the map $\mathrm{Exp}_{\overline{\mathcal{W}}}$, the function $\mathrm{Exp}_{\mathcal{E}}$ allows us to speed up the addition operation by breaking down the original group $\mathcal{J}^0_k(\mathcal{E})$ into a pair (P, c), where $P \in \mathcal{E}(\mathbb{Z}/p\mathbb{Z})$ and $c \in \mathbb{Z}/p^{k-1}\mathbb{Z}$. This representation of the group simplifies the operation of adding two points in the group $\mathcal{J}^0_k(\mathcal{E})$ by reducing it to adding two points in the simpler group $\mathcal{J}^0_1(\mathcal{E})$ and an integer c, respectively. This allows to simplify the computational process and improve performance.

It is noteworthy that the addition formula for the Weierstrass form cannot be applied when points, reduced modulo p, are mapped to the point at infinity. In such cases, the sum of two points over $\mathbb{Z}/p^k\mathbb{Z}$ may produce something like $[0 : 0 : 0]$, which is not a valid point. On the other hand, every point in $\mathcal{E}\Big(\mathbb{Z}/p^k\mathbb{Z}\Big)$ is affine, and therefore the addition formula for an Edwards curve always yields a correct result.

Part III

Coding theory

11 Goppa codes for Edwards curves

In this chapter, we present a method for constructing an Algebraic-Geometric (AG) Goppa code (as discussed in section 6.3) for Edwards curves [32] by using the birational equivalence (discussed in subsec. 3.3.2) between Edwards curves and elliptic curves in Weierstrass form. We address the reader to appendix A.2 and appendix B.2 for, respectively, the pseudocodes and the implementation codes of the algorithms used to compute a basis of the Riemann-Roch space, and a generator matrix for an AG Goppa code for these curves.

Our approach consists of several steps. Firstly, we compute a basis for the Riemann-Roch space of these non-smooth curves. Subsequently, we use this basis to compute the generator matrix of the AG Goppa code. Finally, we provide an example of an Maximum Distance Separable (MDS) AG Goppa code.

11.1 The Riemann-Roch space $\mathcal{L}(D)$

In this section,we describe a method for constructing a basis for the Riemann-Roch (vector) space for an Edwards curve.

As a reminder, the Riemann-Roch space of an algebraic curve is a linear space of rational functions on the curve, with certain degree and pole conditions.

More precisely, given a divisor $D \in \mathrm{Div}(\mathcal{E})$, under the assumption that the support of D does not contain the two singular points Ω_1 and Ω_2, the Riemann-Roch space for an Edwards curve $\mathcal{E}$ is given by

$$\mathcal{L}(D) = \{f \in \overline{\mathbb{K}}(\mathcal{E})^* : \mathrm{div}(f) + D \text{ is effective}\} \cup \{0\}.$$

.

We observe that any divisor D' on the Edwards curve $\mathcal{E}$, of degree $k+1$, such that the singular points Ω_1 and Ω_2 are not part of the support of D', is linearly equivalent to $P + k \cdot O$, for a suitable point $P \in \mathcal{E}(\mathbb{K})$ (or $(k+1)\cdot O$, in the case where $P = O$), meaning that $D' = P + k \cdot O + \mathrm{div}(g)$, for a suitable function g. As the map

$$\begin{aligned} \psi : \mathcal{L}(D') &\longrightarrow \mathcal{L}(P + k \cdot O) \\ \mathbf{F} &\longmapsto g\mathbf{F} \end{aligned}$$

is an isomorphism between $\mathcal{L}(D')$ and $\mathcal{L}(P + k \cdot O)$, we restrict ourselves to considering the latter space.

Theorem 11.1. *Let $\mathcal{E}$ be an Edwards curve defined, over a field $\mathbb{K}$ of characteristic different from 2, by the equation $x^2 + y^2 = 1 + dx^2y^2$, where d is not a square. If*

$P + k \cdot O = D \in \mathrm{Div}(\mathcal{E})$ is a divisor of positive degree $k+1$, where $P = (a, b)$, then $\dim(\mathcal{L}(D)) = k+1$ and

$$\mathcal{L}(D) = \begin{cases} \langle \mathbf{F_0}, \mathbf{F_1}, \dots, \mathbf{F_k} \rangle & \text{if } P \neq O \\ \langle \mathbf{F_0}, \mathbf{F_2}, \dots, \mathbf{F_{k+1}} \rangle & \text{if } P = O \end{cases}$$

where $\mathbf{F_0}, \mathbf{F_1}, \dots, \mathbf{F_{k+1}}$ are rational homogeneous functions defined as follows:

$$\mathbf{F_0} = \frac{Z}{Z}$$

$$\mathbf{F_1} = \begin{cases} \dfrac{Z}{X} & \text{if } P = O' = (0, -1) \\ \dfrac{(X+Z)(Y+Z)}{XY} & \text{if } P = H = (1, 0) \\ \dfrac{(X-Z)(Y+Z)}{XY} & \text{if } P = H' = (-1, 0) \\ \dfrac{(Y+bZ)\cdot X}{(X-aZ)\cdot(Y-Z)} & \text{if } P \notin \{O', H, H'\} \end{cases}$$

$$\mathbf{F_i} = \begin{cases} \dfrac{Z^h}{(Y-Z)^h} & \text{if } i = 2h \\ \dfrac{(Y+Z)Z^h}{X(Y-Z)^h} & \text{if } i = 2h+1 \end{cases}$$

for $2 \leq i \leq k+1$.

Proof. From theorem 3.9, since d is not a square, the two birational maps α and β in equations (3.19a) and (3.19b), respectively, define an isomorphism between $\mathcal{E}$ and a suitable elliptic curve in Weierstrass form $\mathcal{W}$, as detailed in definition 3.13. Additionally, as P is distinct from both Ω_1 and Ω_2, the (surjective) map β induces an (injective) homomorphism $g \mapsto g \circ \beta$ from the space $\mathcal{L}(P + k \cdot O)$ to the space $\mathcal{L}(\alpha(P) + k \cdot \Omega)$, because $\beta(\mathrm{div}(g \circ \beta)) = \mathrm{div}(g)$ for any function $g \in \mathcal{L}(P + k \cdot O)$.

Since the elliptic curve $\mathcal{W}$ is non-singular, by the formula of Riemann-Roch, the dimension of the function space $\mathcal{L}(\alpha(P) + k \cdot \Omega)$ is equal to $k+1$. Thus, it is sufficient to exhibit $k+1$ linearly independent functions belonging to the space $\mathcal{L}(P + k \cdot O)$, as manifestly the space $\mathcal{L}(P + (i-1) \cdot O)$ is contained in $\mathcal{L}(P + i \cdot O)$, for $i = 1, \dots, k+1$.

For $i = 0$, the result holds trivially, as $\mathrm{div}\left(\frac{Z}{Z}\right) = 0$ and, for any point $P \in \mathcal{E}(\mathbb{K})$, we have that the divisor $\mathrm{div}(\mathbf{F_0}) + P$ is effective.

In the case where $i = 1$, we examine the four distinct cases where P takes on the value of O', H, H', and the case where P is not equal to either of these three points. Specifically, we first consider the case where $P = O' = (0, -1)$. By evaluating the divisor of $\frac{Z}{X}$, we have that

$$\mathrm{div}\left(\frac{Z}{X}\right) = (2 \cdot \Omega_1 + 2 \cdot \Omega_2) - (O + O' + 2 \cdot \Omega_2) = 2 \cdot \Omega_1 - O - O',$$

hence the divisor $\mathrm{div}(\mathbf{F_1}) + O' + O$ is effective.

Furthermore, in the case where $P = H = (1, 0)$, we have that

$$\begin{aligned}
&\operatorname{div}\left(\frac{(X+Z)(Y+Z)}{XY}\right) = \\
&= (2\cdot H' + 2\cdot\Omega_2 + 2\cdot O' + 2\cdot\Omega_1) - (H + H' + 2\cdot\Omega_1 + O + O' + 2\cdot\Omega_2) = \\
&= H' + O' - H - O,
\end{aligned}$$

thus the divisor $\operatorname{div}(\mathbf{F_1}) + H + O$ is effective. Similarly, in the case where $P = H' = (-1, 0)$, we have that

$$\begin{aligned}
&\operatorname{div}\left(\frac{(X-Z)(Y+Z)}{XY}\right) = \\
&= (2\cdot H + 2\cdot\Omega_2 + 2\cdot O' + 2\cdot\Omega_1) - (H + H' + 2\cdot\Omega_1 + O + O' + 2\cdot\Omega_2) = \\
&= H + O' - H' - O,
\end{aligned}$$

thus the divisor $\operatorname{div}(\mathbf{F_1}) + H' + O$ is effective. Lastly, when $i = 1$, we consider the case in which $P = (a, b)$ is distinct from O', H, and H'. In this case, as illustrated in the fig. 11.1, given the two points $R = (a, -b)$ and $R' = (-a, -b)$, we have that

$$\begin{aligned}
&\operatorname{div}\left(\frac{(Y+bZ)\cdot X}{(X-aZ)\cdot(Y-Z)}\right) = \\
&= (R + R' + 2\cdot\Omega_1 + O + O' + 2\cdot\Omega_2) - (P + R + 2\cdot\Omega_2 + 2\cdot O + 2\cdot\Omega_1) = \\
&= R' + O' - P - O,
\end{aligned}$$

thus the divisor $\operatorname{div}(\mathbf{F_1}) + P + O$ is effective.

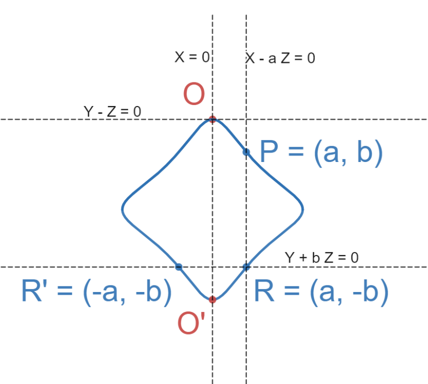

FIGURE 11.1: Edwards real curve with $d = -8$. The points P, R, and R' have the following coordinates (a, b), $(a, -b)$, and $(-a, -b)$, respectively

Below, we prove that the divisor $\operatorname{div}(\mathbf{F_i}) + P + k\cdot O$ is effective for any $P \in \mathcal{E}(\mathbb{K})$ and $k \geq 2$. Specifically, for $2 \leq i \leq k+1$, we claim that $\operatorname{div}(\mathbf{F_i}) + P + i\cdot O$ is effective in both the cases where $i \equiv 0 \pmod 2$ and $i \equiv 1 \pmod 2$. First, if i is an even integer equal to $2h$, then we have that

$$\operatorname{div}\left(\frac{Z^h}{(Y-Z)^h}\right) = (2h\cdot\Omega_1 + 2h\cdot\Omega_2) - (2h\cdot O + 2h\cdot\Omega_1) = 2h\cdot\Omega_2 - 2h\cdot O.$$

Second, if i is an odd integer equal to $2h+1$, then we remark that $\mathbf{F_{2h+1}} = \mathbf{F_{2h}} \cdot \frac{Y+Z}{X}$, and we have that

$$\begin{aligned}
&\operatorname{div}\left(\frac{(Y+Z)Z^h}{X(Y-Z)^h}\right) = \\
&\quad = (2\cdot O' + 2\cdot\Omega_1 + 2h\cdot\Omega_1 + 2h\cdot\Omega_2) - (O + O' + 2\cdot\Omega_2 + 2h\cdot O + 2h\cdot\Omega_1) = \\
&\quad = O' + 2\cdot\Omega_1 + (2h-2)\cdot\Omega_2 - (2h+1)\cdot O.
\end{aligned}$$

Therefore, the divisor $\operatorname{div}(\mathbf{F_i}) + P + i\cdot O$ is effective in both the cases where $i \equiv 0 \pmod 2$ and $i \equiv 1 \pmod 2$.

Thus far, we have proven that for each function $\mathbf{F_i}$, the divisor $\operatorname{div}(\mathbf{F_i}) + P + k\cdot O$ is effective, where P is non necessarily distinct from O. In order to fully prove our claim, it is necessary to show that all these functions are linearly independent, but this follows from standard and elementary arguments.

Furthermore, it is also worth noting that when $P = O$, removing the function $\mathbf{F_1}$ and adding the function $\mathbf{F_{k+1}}$ will also result in $k+1$ linearly independent functions.

■

Remark 11.1. *It should be noted that the proof about $\mathcal{L}(D)$ in theorem 11.1 cannot be extended to the case in which d is not a square, or when P is equal to Ω_1 or Ω_2, as the map β is not invertible at these points, as stated in remark 3.15. For a comprehensive theory on Riemann-Roch spaces on curves with singular points, we address the reader to* §IV.2 *in* [70].

11.1.1 Computational cost

In this section, we determine the computational cost of evaluating each element of the basis of the space $\mathcal{L}(P + (k-1)\cdot O)$ at a point $P \in \mathcal{E}(\mathbb{F}_q)$.

Before proceeding, we recall that the costs of modular addition, multiplication, and inversion over $\mathbb{F}_q$ are $\mathcal{O}(\ln(q))$, $\mathcal{O}\left(\ln^2(q)\right)$, $\mathcal{O}\left(\ln^3(q)\right)$, respectively.

First, we note that we can compute $\mathbf{F_{2h}}$ from $\mathbf{F_{2h-2}}$, and $\mathbf{F_{2h+1}}$ from $\mathbf{F_{2h}}$. More precisely, we have that

$$\begin{cases} \mathbf{F_{2h}} = \mathbf{F_2} \cdot \mathbf{F_{2h-2}} & \text{if } h \geq 2, \\ \mathbf{F_{2h+1}} = \dfrac{Y+Z}{X}\mathbf{F_{2h}} & \text{if } h \geq 1. \end{cases}$$

This implies that at each step we have to perform a single multiplication modulo q by the last (or the second-last) value. Additionally, we can pre-calculate the value of the function $\frac{Y+Z}{X}$ at P with a cost $\mathrm{C}\left(\frac{Y+Z}{X}\right)$ equal to $\mathcal{O}(\ln(q)) + \mathcal{O}\left(\ln^2(q)\right) + \mathcal{O}\left(\ln^3(q)\right) \approx \mathcal{O}\left(\ln^3(q)\right)$ in order to further optimize the computation.

Below, to increase readability for the reader, we denote the cost of modular addition as A, the cost of modular multiplication as M, and the cost of modular inversion as

I. Hence, if $\mathrm{C}(\mathbf{F_i})$ is the cost of evaluating $\mathbf{F_i}$ at P, then we have the following costs:

$$\begin{cases} \mathrm{C}(\mathbf{F_0}) = 0, & \\ \mathrm{C}(\mathbf{F_1}) = M + I & \text{if } P = O', \\ \mathrm{C}(\mathbf{F_1}) = 2A + 3M + I & \text{if } P \in \{H, H'\}, \\ \mathrm{C}(\mathbf{F_1}) = 3A + 5M + I & \text{if } P \notin \{O, O', H, H'\}, \\ \mathrm{C}(\mathbf{F_2}) = A + M + I, & \\ \mathrm{C}\left(\dfrac{Y+Z}{X}\right) = A + M + I, & \\ \mathrm{C}(\mathbf{F_i}) = M & \text{if } 2 < i < k. \end{cases}$$

Since the case in which $P \notin \{O, O', H, H'\}$ gives the maximum cost for $\mathbf{F_1}$, we consider the latter for the computation of the worst-case cost.

Therefore, the worst-case cost $\mathrm{C}(\mathcal{L}(D))$ of evaluating the k functions belonging to the basis of $\mathcal{L}(D)$ is

$$\mathrm{C}(\mathcal{L}(D)) = 5A + (7 + (k-3))M + 3I = \mathcal{O}\Big(k \cdot \ln^2(q) + \ln^3(q)\Big).$$

11.2 AG Goppa codes

In this section, we determine the generator matrix and the parity-check matrix for a $[n, k, d]_q$ AG Goppa code for an Edwards curve $\mathcal{E}$ over the finite field $\mathbb{F}_q$, and compute its computational cost.

We adapt the definition of an AG Goppa code in definition 6.10 to our case.

Definition 11.1. *Let D be a reduced divisor of positive degree $\delta D = k$ of an Edwards curve $\mathcal{E}$ over $\mathbb{F}_q$, where $q = p^t$ and p is a prime number. Let $\langle \mathbf{F_0}, \mathbf{F_1}, \dots, \mathbf{F_{k-1}} \rangle$ be the basis, computed in section 11.1, of the Riemann-Roch space $\mathcal{L}(D)$, let $T = \{P_1, \dots, P_n\}$ be a set of n points such that $P_j \in \mathcal{E}(\mathbb{F}_q)$, and $P_j \notin \mathrm{supp}(D)$. The matrix $(G_{ij}) = G \in {\mathbb{F}_q}^{k \times n}$ such that, for $i = 1, \dots, k$ and $j = 1, \dots, n$, $G_{ij} = \mathbf{F_{i-1}}(P_j)$, is the generator matrix for an $[n, k, d]_q$ AG Goppa code for the Edwards curve $\mathcal{E}$.*

If we order the points in T so that the first k columns of the generator matrix

$$G = \begin{pmatrix} \mathbf{F_0}(P_1) & \mathbf{F_0}(P_2) & \cdots & \mathbf{F_0}(P_n) \\ \mathbf{F_1}(P_1) & \mathbf{F_1}(P_2) & \cdots & \mathbf{F_1}(P_n) \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{F_{k-1}}(P_1) & \mathbf{F_{k-1}}(P_2) & \cdots & \mathbf{F_{k-1}}(P_n) \end{pmatrix} \in {\mathbb{F}_q}^{k \times n}$$

of the Goppa code $\mathcal{C}$ are linearly independent, e.g. by applying the Gauss-Jordan method, then G can be reduced in its standard form $[I_k | M]$, where I_k is the identity matrix of order k and $M \in {\mathbb{F}_q}^{k \times (n-k)}$. Once G is in standard form, the parity-check matrix $H \in {\mathbb{F}_q}^{(n-k) \times n}$ of this Goppa code, that is, the matrix such that $G \cdot H^T = \mathbf{0}$ and $H \cdot \mathbf{y}^T = \mathbf{0}$ for every codeword $\mathbf{y} \in \mathcal{C}$, is simply $H = [-M^T | I_{n-k}]$. Thus, the code $\mathcal{C}$ is also defined as $\{\mathbf{y} \in {\mathbb{F}_q}^n : H \cdot \mathbf{y}^T = \mathbf{0}\}$.

11.2.1 Computational cost of constructing a Goppa code over Edwards curves

In order to compute the generator matrix G, we need to evaluate each of the n points in the set T for each element of the basis of $\mathcal{L}(D)$. This results in a computational cost of $\mathcal{O}(n \cdot \mathrm{C}(\mathcal{L}(D)))$, as G is a matrix of dimension $k \times n$. Furthermore, the cost of computing the parity-check matrix depends on the method used to compute the nullspace of G. For example, if we use the Gaussian elimination method to reduce the matrix G to its standard form, then the cost is $\mathcal{O}\Big(\max(n,k)^3\Big)$.

Therefore, for $n > k$, the overall computational cost of constructing an AG Goppa code for an Edwards curve over $\mathbb{F}_q$ is

$$n \cdot \mathrm{C}(\mathcal{L}(D)) + \mathcal{O}\Big(\max(n,k)^3\Big) \approx \mathcal{O}\Big(n \cdot (k \cdot \ln^2(q) + \ln^3(q))\Big) + \mathcal{O}\Big(n^3\Big).$$

11.3 A small example

In this section, we provide a small example of an AG Goppa code for an Edwards curve $\mathcal{E}$, defined by the equation $x^2 + y^2 = 1 + dx^2y^2$, over the finite field $\mathbb{F}_{17}$.

Let d be equal to 10. The set of affine points belonging to $\mathcal{E}(\mathbb{F}_{17})$ is equal to

$$\begin{aligned}&\{(0,1),(0,16),(1,0),(2,2),(2,15),(3,6),\\&\ (3,11),(5,8),(5,9),(6,3),(6,14),(8,5),\\&\ (8,12),(9,5),(9,12),(11,3),(11,14),(12,8),\\&\ (12,9),(14,6),(14,11),(15,2),(15,15),(16,0)\}.\end{aligned}$$

Consider the divisor $D = (2,15) + 4 \cdot O$ of degree $k = 5$, and the set T of $n = 7$ affine points equal to

$$\{(5,8),(5,9),(6,3),(6,14),(8,5),(8,12),(9,5)\}.$$

As stated in theorem 11.1, a basis for $\mathcal{L}(D)$ is given by

$$\begin{aligned}\mathcal{L}(D) &= \langle \mathbf{F_0}, \mathbf{F_1}, \mathbf{F_2}, \mathbf{F_3}, \mathbf{F_4} \rangle = \\ &= \left\langle 1, \frac{x(y+15)}{(x-2)(y-1)}, \frac{1}{y-1}, \frac{y+1}{x(y-1)}, \frac{1}{(y-1)^2} \right\rangle.\end{aligned}$$

Additionally, the generator matrix $G = (G_{ij})$ of the AG Goppa code, defined by putting $G_{ij} = \mathbf{F_{i-1}}(P_j)$, is equal to

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 16 & 5 & 5 & 4 & 1 & 11 & 4 \\ 5 & 15 & 9 & 4 & 13 & 14 & 13 \\ 9 & 13 & 6 & 10 & 14 & 10 & 3 \\ 8 & 4 & 13 & 16 & 16 & 9 & 16 \end{pmatrix}.$$

Lastly, by reducing the generator matrix G to its standard form, we can compute the parity-check matrix H, which is represented by the following matrix:

$$\left(\begin{array}{ccccc|cc} 7 & 3 & 1 & 13 & 9 & 1 & 0 \\ 2 & 12 & 9 & 12 & 15 & 0 & 1 \end{array}\right).$$

It is worth to note that the lower bound for the minimum distance of this code is $d \geq n - \delta D = 7 - 5 = 2$, whereas, according to the theorem 6.1, the upper bound for the minimum distance is $d \leq n - k + 1 = 7 - 5 + 1 = 3$. Hence, the minimum distance for the above code is either 2 or 3. In particular, two columns of H are always linearly independent, however, three of them are dependent. For example, the following linear combination of the first three columns of H is equal to zero: $(7,2)^T + 5 \cdot (3,12)^T + 12 \cdot (1,9)^T = (0,0)^T$. As a consequence, the minimum distance of this code is equal to 3, and the above code is a $[7,5,3]_{17}$ AG Goppa MDS code.

11.4 A small example of a McEliece cryptosystems

As previously discussed in section 6.2, it is not safe to embed AG Goppa codes into the McEliece protocol. However, for the sake of completeness, in this section, we provide a toy example by embedding the code computed in section 11.3 into the McEliece cryptosystem.

As is typical for the illustration of a cryptographic protocol, we refer to the two parties involved in this scheme as Alice (the receiver) and Bob (the sender).

Bob initiates the communication with Alice, thus Alice generates, for instance, the $[n,k,d]_q = [7,5,3]_{17}$ code computed in section 11.3. Additionally, Alice randomly generates a non-singular matrix $S \in \mathbb{F}_q{}^{k \times k}$ and a permutation matrix P of dimension $n \times n$. For example, Alice generates the following matrices

$$S = \begin{pmatrix} 4 & 0 & 7 & 5 & 3 \\ 15 & 12 & 2 & 5 & 2 \\ 3 & 2 & 3 & 6 & 0 \\ 9 & 2 & 9 & 0 & 1 \\ 3 & 9 & 6 & 8 & 9 \end{pmatrix} \qquad P = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Therefore, the private-key of Alice is the triple (S, P, D_A), where D_A is a suitable decoding algorithm that computes the codeword $\mathbf{x}$ from the word $\mathbf{y} = \mathbf{x} + \mathbf{e}$, with $\mathbf{e}$ being an error vector, i.e., $D_A(\mathbf{y}) = \mathbf{x}$.

Additionally, the public-key of Alice is the pair $(G^\star, t)$, where $t = \left\lfloor \frac{d-1}{2} \right\rfloor = 1$ is the maximum number of error that this code can correct, and $G^\star = S \cdot G \cdot P$ is a matrix

equal to

$$\begin{pmatrix} 9 & 11 & 16 & 0 & 5 & 9 & 6 \\ 2 & 0 & 8 & 13 & 0 & 5 & 6 \\ 9 & 15 & 0 & 8 & 0 & 8 & 2 \\ 8 & 1 & 5 & 11 & 14 & 13 & 9 \\ 6 & 15 & 6 & 12 & 13 & 7 & 15 \end{pmatrix}.$$

Bob receives the public-key of Alice and generates the message $(3, 8, 4, 6, 5) = \mathbf{m} \in {\mathbb{F}_{17}}^k$ and, in order to encrypt $\mathbf{m}$, he generates the error vector $(0, 0, 0, 0, 0, 0, 1) = \mathbf{e} \in {\mathbb{F}_{17}}^n$ of weight $t = 1$. Subsequently, Bob computes the ciphertext

$$\mathbf{c}^{\star} = \mathbf{m} \cdot G^{\star} + \mathbf{e} = (4, 4, 2, 7, 11, 8, 0),$$

and send it to Alice. In order to decrypt the received message, Alice computes

$$\mathbf{h} = \mathbf{c}^{\star} \cdot P^{-1} = \mathbf{m} \cdot S \cdot G + \mathbf{e} \cdot P^{-1} = (0, 2, 7, 4, 4, 8, 11)$$

and decodes the vector $\mathbf{h}$ using the decoding algorithm D_A. Depending on the type of decoding algorithm used by Alice, she gets back either the codeword $\mathbf{m} \cdot S \cdot G = (16, 2, 7, 4, 4, 8, 11)$ or the word $\mathbf{m} \cdot S = (9, 8, 14, 0, 8)$. In the latter case, Alice computes the message as $\mathbf{m} = (9, 8, 14, 0, 8) \cdot S^{-1}$. On the other hand, in the former case, Alice has to compute the word $\mathbf{m} \cdot S$ by solving the linear system $\mathbf{x} \cdot G = (16, 2, 7, 4, 4, 8, 11)$. It is worth to note that, if G is taken in standard form, then the word $\mathbf{m} \cdot S$ is simply equal to the first k elements of the vector $\mathbf{m} \cdot S \cdot G$.

11.5 Implementation

12 Goppa codes for hyperelliptic curves

In this chapter, we propose a method for constructing an Algebraic-Geometric (AG) Goppa code (as discussed in section 6.3) for hyperelliptic curves [29] using the Mumford representation (see theorem 4.2) of the divisors of the Jacobian of these curves. We address the reader to appendix A.3 and appendix B.3 for, respectively, the pseudocodes and the implementation codes of the algorithms used to compute a basis of the Riemann-Roch space, and a generator matrix for an AG Goppa code for these curves.

Specifically, for a given (imaginary) hyperelliptic curve $\mathcal{H}$ of genus $g \geq 1$, with Weierstrass point Ω taken as the point at infinity, we determine a basis for the Riemann-Roch space $\mathcal{L}(D)$, where $\Delta + m \cdot \Omega = D \in \mathcal{J}(\mathcal{H})$, and Δ is a zero degree divisor represented in Mumford form. We then construct an AG Goppa code over $\mathbb{F}_p$, with $p \geq 2$, using the basis of $\mathcal{L}(D)$.

12.1 The Riemann-Roch space $\mathcal{L}(D)$

Let $\mathcal{H}$ be an (imaginary) hyperelliptic curve of genus $g \geq 1$ over the field $\mathbb{K}$ defined by the equation

$$y^2 + h(x)y = f(x), \tag{12.1}$$

where $\deg(f(x)) = 2g + 1$, $\deg(h(x)) \leq g$, and $\Omega = [Z : X : Y] = [0 : 0 : 1]$ is the point at infinity of $\mathcal{H}$ ([56, Prop. 1.2]). Recall that, as stated in remark 4.1, if $\mathrm{char}(\mathbb{K}) \neq 2$, then one can reduce the defining equation of $\mathcal{H}$ to $y^2 = f(x)$ by applying the transformation $(x, y) \mapsto \left(f(x) - \frac{h^2(x)}{4}, y - \frac{h(x)}{2}\right)$, while if $\mathrm{char}(\mathbb{K}) = 2$, then it is not possible to reduce $h(x)$ to zero.

Although the reduction of a divisor D to its reduced Mumford form might be an inconvenient task, involving the application of the Cantor algorithm (see remark 12.1), this difficulty does not occur in the construction of Goppa codes, as in that case one can directly take D in the reduced form $D = \Delta + m \cdot \Omega$.

It is worth noting that since the Riemann-Roch space

$$\mathcal{L}(D) = \{f \in \overline{\mathbb{K}}(\mathcal{H})^* : \mathrm{div}(f) + D \text{ is effective}\} \cup \{0\}.$$

is null in cases where D has negative degree or where D has degree zero and $D \notin \mathrm{Princ}(\mathcal{H})$, and $\mathcal{L}(D) = \left\langle F_0^{-1} \right\rangle$ in the case where $D = \mathrm{div}(F_0)$, from now on we will assume D has positive degree m. Thus, any divisor class $D + \mathrm{Princ}(\mathcal{H}) \in$

$\mathrm{Div}^0(\mathcal{H})/\mathrm{Princ}(\mathcal{H})$ can be reduced to the form

$$D = \sum_{i=1}^{t} P_i + (m-t)\cdot\Omega + \mathrm{div}(\psi(x,y))$$

where $t \leq g$ is the number of points P_j in $\mathcal{H}$ distinct from Ω, and $\psi(x,y)$ is a suitable function in $\overline{\mathbb{K}}(\mathcal{H})$.

In order to extend the use of Mumford representation to divisors of arbitrary degree, we will make use of the following observation.

Remark 12.1. *Note that any divisor of degree zero, represented as*

$$\Delta = \sum_{i=1}^{s} l_i \cdot (x_i, y_i) - (l_1 + \ldots + l_s)\cdot\Omega,$$

on the curve $\mathcal{H}$, where $l_i > 0$ for any index i, uniquely determines the polynomial $a(x) = \prod_{i=1}^{s}(x-x_i)^{l_i}$ and the polynomial $b(x)$ which is the interpolating polynomial such that $b(x_t) = y_t$, with $t = 1,\ldots,s$. The pair $(a(x), b(x))$ satisfy the property that $b^2(x) + h(x)b(x) - f(x)$ is a multiple of $a(x)$ and the degree of $b(x)$ is less than the degree of $a(x)$. Conversely, for any pair of polynomials $a(x)$ and $b(x)$ such that $b^2(x) + h(x)b(x) - f(x)$ is a multiple of $a(x)$ and the degree of $b(x)$ is less than the degree of $a(x)$, there exists a unique divisor of degree zero, represented as $\Delta = \mathrm{div}(a(x), b(x))$ (cf. [18]). It is worth noting that an intersection point of the curve with the x-axis is included in the support of Δ if and only if $\gcd(a(x), a'(x), b(x)) \neq 1$ (as stated in theorem 4.2). If $\gcd(a(x), a'(x), b(x)) = 1$ and the degree of $a(x)$ is not greater than the genus g of the curve (or equivalently, if the support of Δ contains at most g affine points that are mutually non-opposite), one says that $\mathrm{div}(a(x), b(x))$ is in the Mumford form (or reduced form).

We note that any divisor $D = D_1 - D_2$ (with D_i effective of degree $m_i \in \mathbb{Z}$) can be written as

$$D = \Delta + m\cdot\Omega + \mathrm{div}(\psi(x,y)),$$

where $m = m_1 - m_2$, $\Delta = \mathrm{div}(u(x), v(x))$ is in Mumford form, and $\psi(x,y)$ is a suitable function obtained with the following arguments.

First, by considering the vertical lines $x - x_i$ passing through the points in the support of D_2, we can write

$$(-D_2) = D_2' - 2m_2\cdot\Omega - \mathrm{div}(\phi),$$

with $\phi = \prod(x - x_i)$ and D_2' is an effective divisor. Consequently, we can rewrite D as follows:

$$D = D_1 - D_2 = D_3 - 2m_2\cdot\Omega - \mathrm{div}(\phi),$$

where $D_3 = D_1 + D_2'$ is an effective divisor of degree $m_1 + m_2$, and thus of the form

$$D_3 = \mathrm{div}(a(x), b(x)) + (m_1 + m_2)\cdot\Omega.$$

Secondly, by applying the reduction step of Cantor's algorithm (cf. [18], and [49] for the case where $\mathrm{char}(\mathbb{K}) = 2$), we change the divisor D_3 with the divisor

$$D_3' = D_3 - \mathrm{div}(y - b(x)) = \mathrm{div}(a'(x), b'(x)),$$

which belongs to the same divisor class, where

$$a'(x) = \frac{f(x) - b(x)h(x) - b^2(x)}{a(x)},$$

$$b'(x) = (-h(x) - b(x)) \pmod{a'(x)}.$$

This way, one has that $\deg(a'(x)) < \deg(a(x))$*. Therefore, after finitely many iterations, one obtains* $\deg(b'(x)) < \deg(a'(x)) \le g$*, and one can write*

$$D = \Delta + m \cdot \Omega + \operatorname{div}(\psi(x, y)),$$

where $\psi(x, y)$ *is the resulting function from the above reduction process.*

Finally, the function Φ *defined as*

$$\begin{aligned} \Phi \colon \mathcal{L}(D) &\longrightarrow \mathcal{L}(\Delta + m \cdot \Omega) \\ F &\longmapsto \psi(x, y) \cdot F \end{aligned}$$

is an isomorphism. From here on, up to the latter isomorphism, we will assume that $D = \Delta + m \cdot \Omega$*, where* $m > 0$*.*

In the following theorem, we determine a basis for the Riemann-Roch space $\mathcal{L}(D)$, where $D = \Delta + m \cdot \Omega$, and $\Delta = \operatorname{div}(u(x), v(x))$ is in Mumford representation, with $t := \deg(u(x)) \le g$. Furthermore, the varying of dimension of this space, depending on m, becomes manifest: in order to determine $\dim(\mathcal{L}(D))$, in [24, Lemma 2.1.] it is distinguished the case $m \ge 2g - t - 1$, where $\dim(\mathcal{L}(D)) = m - g + 1$, and the case $t \le m < 2g - t - 1$, where $\dim(\mathcal{L}(D)) = \lfloor \frac{m-t}{2} \rfloor + 1$ (cf. remark 12.2).

Theorem 12.1. *Given the hyperelliptic curve* $\mathcal{H}$ *of genus* $g \ge 1$ *and degree* $d = 2g + 1$*, defined by the equation (12.1), and given the divisor* $D = \Delta + m \cdot \Omega$ *of positive degree* m *on* $\mathcal{H}$*, as defined in remark 12.1, where* $\Delta = \operatorname{div}(u(x), v(x))$ *is in the Mumford representation, let* $t := \deg(u(x)) \le g$ *and let*

$$\Psi(x, y) = \frac{y + v(x)}{u(x)} \qquad \text{if } \operatorname{char}(\mathbb{K}) \neq 2,$$

$$\Psi(x, y) = \frac{y + v(x) + h(x)}{u(x)} \qquad \text{if } \operatorname{char}(\mathbb{K}) = 2.$$

If $m < d - t$*, then a basis of* $\mathcal{L}(D)$ *is equal to* $\langle x^i \rangle$*, where* $0 \le i \le \frac{m-t}{2}$*, otherwise, if* $m \ge d - t$*, then a basis of* $\mathcal{L}(D)$ *is equal to* $\langle x^i, \Psi(x, y) \cdot x^j \rangle$*, where* $0 \le i \le \frac{m-t}{2}$ *and* $0 \le j \le \frac{m-(d-t)}{2}$*.*

Proof. In order to compute $\operatorname{div}(\Psi(x, y))$, it is necessary to recall that $\deg(v(x)) < \deg(u(x)) \le g$ and that, in the case where $\operatorname{char}(\mathbb{K}) = 2$, $\deg(h(x)) \le g$, as well.

Since $l = \max(\deg(v(x)), \deg(h(x))) \le g$, it follows that

$$\deg\Big((-v(x) - h(x))^2\Big) < \deg(f(x)) = d = 2g + 1.$$

As a result, there are d intersection points of the curve $y + v(x) + h(x) = 0$ and $\mathcal{H}$ in the affine plane, with the remaining $d(l - 1)$ intersection points coinciding with

Ω. More precisely, t intersection points in the affine plane belong to the support of the divisor $\widehat{\Delta} = \operatorname{div}(u(x), w(x))$ in Mumford representation, where $w(x) = (-v(x) - h(x)) \pmod{u(x)}$. Therefore, we have that

$$\operatorname{div}(y + v(x) + h(x)) = \widehat{\Delta} + W + (t + d(l-1)) \cdot \Omega,$$

where W is the effective divisor of degree $d - t$, whose support consists of the remaining intersection points in the affine plane. Note that, in the case $t = 0$, the divisor Δ has the Mumford representation $(1, 0)$, the degree of W is equal to d and the support of W coincides with the intersections of $\mathcal{H}$ with the curve $y + h(x) = 0$.

On the other hand, the intersection of $u(x) = 0$ and $\mathcal{H}$ is simply

$$\operatorname{div}(u(x)) = \Delta + \widehat{\Delta} + (td) \cdot \Omega.$$

In summary, we have that

$$\begin{aligned}\operatorname{div}(\Psi(x,y)) &= \operatorname{div}(y + v(x) + h(x)) - \operatorname{div}(u(x)) = \\ &= W - \Delta - (d-t) \cdot \Omega.\end{aligned} \tag{12.2}$$

In the case where $t = 0$, this result can be simplified to

$$\operatorname{div}(\Psi(x,y)) = \operatorname{div}(y + h(x)) - \operatorname{div}(1) = W - d \cdot \Omega,$$

since $\Delta = (1, 0)$ and $\Psi(x, y) = y + h(x)$. Therefore, for $t \geq 0$, the equality in equation (12.2) holds and

$$\Psi(x,y) \in \mathcal{L}(D) \text{ if and only if } m \geq d - t. \tag{12.3}$$

In the remaining part of this proof, we determine a basis for $\mathcal{L}(D)$ in the cases $m \geq d-t$, and $m < d-t$, respectively. Specifically, consider the case $m \geq d-t$, which implies that $\Psi(x, y) \in \mathcal{L}(D)$. First, if $t \in \{0, 1, 2\}$, then $m \geq d - 2$ and, according to the theorem of Riemann-Roch, the dimension of $\mathcal{L}(D)$ is equal to $m - g + 1$. Thus, since the dimension is known, in order to prove that

$$\mathcal{L}(D) = \left\langle x^i, \Psi(x,y) \cdot x^j \right\rangle, \text{ with } 0 \leq i \leq \frac{m-t}{2} \text{ and } 0 \leq j \leq \frac{m-(d-t)}{2}, \tag{12.4}$$

it is sufficient to note that, for each possible value of the parameters i and j, the functions in equation (12.4) belong to $\mathcal{L}(D)$. Since

$$1 + \left\lfloor \frac{m-t}{2} \right\rfloor + 1 + \left\lfloor \frac{m-(d-t)}{2} \right\rfloor = m - g + 1,$$

the claim follows from dimensional considerations. Additionally, we have that

$$D + \operatorname{div}\left(x^i\right) = (\Delta + m \cdot \Omega) + i \cdot \operatorname{div}(x), \tag{12.5}$$

and

$$\begin{aligned}D + \operatorname{div}\left(\Psi(x,y) \cdot x^j\right) &= (\Delta + m \cdot \Omega) + j \cdot \operatorname{div}(x) + (W - \Delta - (d-t) \cdot \Omega) = \\ &= W + j \cdot \operatorname{div}(x) - (d - t - m) \cdot \Omega,\end{aligned} \tag{12.6}$$

are effective divisors, as $\operatorname{div}(x) = \operatorname{div}((x - x_1)(x - x_2), 0)$, where $(x_i, 0) \in \mathcal{H}(\mathbb{K})$, hence the functions belong to $\mathcal{L}(D)$.

Second, in the case where $d-t \leq m < d-2$, the dimension of $\mathcal{L}(D)$ is not necessarily equal to $m-g+1$, but still $\Psi(x,y) \in \mathcal{L}(D)$. In order to proceed, let ϵ and m_ϵ be such that $0 \leq \epsilon \leq t-2$, and $m = m_\epsilon = d-2-\epsilon$. Then, putting for short

$$\mathcal{L}_\epsilon := \mathcal{L}(\Delta + m_\epsilon \cdot \Omega),$$

we have that the space $\mathcal{L}_0 = \mathcal{L}(\Delta + (d-2)\cdot\Omega)$ is generated by the functions x^i and $\Psi(x,y)\cdot x^j$, with $0 \leq i \leq \frac{m_0-t}{2}$ and $0 \leq j \leq \frac{m_0-(d-t)}{2}$, as established in the previous case. Equations (12.5) and (12.6) show that $\mathcal{L}_{\epsilon+1} \leq \mathcal{L}_\epsilon$ since the functions x^i and $\Psi(x,y)\cdot x^j$ of $\mathcal{L}_\epsilon$ also belong to $\mathcal{L}_{\epsilon+1}$ as long as $0 \leq i \leq \frac{m_{\epsilon+1}-t}{2}$, and $0 \leq j \leq \frac{m_{\epsilon+1}-(d-t)}{2}$. Thus, since $m_{\epsilon+1} = m_\epsilon - 1$ and

$$\dim(\mathcal{L}_\epsilon) = 1 + \left\lfloor \frac{m_\epsilon - t}{2} \right\rfloor + 1 + \left\lfloor \frac{m_\epsilon - (d-t)}{2} \right\rfloor,$$

it follows that

$$\dim(\mathcal{L}_{\epsilon+1}) = 1 + \left\lfloor \frac{m_{\epsilon+1} - t}{2} \right\rfloor + 1 + \left\lfloor \frac{m_{\epsilon+1} - (d-t)}{2} \right\rfloor = \dim(\mathcal{L}_\epsilon) - 1.$$

The missing function is either x^i or $\Psi(x,y)\cdot x^j$, as a result of d being odd and thus changing the parity of $m_{\epsilon+1} - t$ and $m_{\epsilon+1} - (d-t)$.

In the remaining part of this proof, we consider the cases where $m < d-t$, that is, where $\Psi(x,y) \notin \mathcal{L}(D)$, as stated in equation (12.3). Specifically, we focus on the cases where $t = 0$ and $m \in \{d-2, d-1\}$ or where $t = 1$ and $m = d-2$. In these cases, on the one hand, we have that $\lfloor \frac{m-t}{2} \rfloor = m-g$ and, on the other hand, by the theorem of Riemann-Roch, the dimension of $\mathcal{L}(D)$ is $m-g+1$. Therefore, by dimensional reason, $\mathcal{L}(D) = \langle x^i \rangle$, where $0 \leq i \leq \lfloor \frac{m-t}{2} \rfloor$.

Finally, in order to prove that $\mathcal{L}(D) = \langle x^i \rangle$, where $0 \leq i \leq \frac{m-t}{2}$, we consider the remaining cases where $t \in \{0,1\}$ and $m < d-2$, or $2 \leq t \leq m < d-t$. In order to simplify the notation, we write $m = m_\epsilon = d-t-\epsilon$, with $1 \leq \epsilon \leq d-2t$, and define

$$\mathcal{L}_\epsilon := \mathcal{L}(\Delta + m_\epsilon \cdot \Omega).$$

Note that, for $\epsilon = 0$, that is, $m = m_0 = d-t$, by equation (12.4), we have that $\mathcal{L}_0 = \langle x^i, \Psi(x,y) \rangle$, with $0 \leq i \leq \frac{m_0-t}{2}$.

According to equation (12.3), $\Psi(x,y) \notin \mathcal{L}_\epsilon$ for any $\epsilon > 0$. Furthermore, as per equation (12.5), we have that $\mathcal{L}_{\epsilon+1} \leq \mathcal{L}_\epsilon$ for any $0 \leq \epsilon \leq d-2t$, since the functions x^i of $\mathcal{L}_\epsilon$ belong to $\mathcal{L}_{\epsilon+1}$ for $0 \leq i \leq \frac{m_{\epsilon+1}-t}{2}$. Additionally, given that $m_{\epsilon+1} = m_\epsilon - 1$, it follows that

$$\dim(\mathcal{L}_{\epsilon+1}) = 1 + \left\lfloor \frac{m_{\epsilon+1} - t}{2} \right\rfloor = \begin{cases} \dim(\mathcal{L}_\epsilon) & \text{if } m_\epsilon - t \text{ is odd,} \\ \dim(\mathcal{L}_\epsilon) - 1 & \text{if } m_\epsilon - t \text{ is even.} \end{cases} \tag{12.7}$$

It is worth highlighting that the equalities in equation (12.7) demonstrate, as well, that the theorem holds true for any value of m. ■

Remark 12.2. *It is remarkable that the bounds presented in* [24, Lemma 2.1.] *are different from those established in theorem 12.1. Specifically, for* $m \in \{2g-t, 2g-t-1\}$*, our theorem states that* $\dim(\mathcal{L}(D)) = 1 + \lfloor \frac{m-t}{2} \rfloor$*, while in* [24,

Lemma 2.1.] *it is stated that* $\dim(\mathcal{L}(D)) = m - g + 1$. *It is clear that the two values are equal for the aforementioned values of* m, *that is,* $m \in \{2g - t, 2g - t - 1\}$.

Additionally, the necessary condition in [24, Lemma 2.1.] *for* $\dim(\mathcal{L}(D)) \neq m - g + 1$, *namely* $m < d - t - 2$, *is also sufficient as per our theorem.*

Furthermore, an interesting phenomenon occurs when $g < m < 2g - 1$ *and* $t \in \{g, g-1, g-2\}$, *as in these cases we have that* $m \geq d - t - 2$, *thus leading to* $\dim(\mathcal{L}(D)) = m - g + 1$, *regardless of the theorem of Riemann-Roch.*

Remark 12.3. *Recall that* Ω *was a given Weierstrass point of the curve* $\mathcal{H}$, *and in fact* $\mathcal{L}(2(g-\epsilon)\cdot\Omega) = \mathcal{L}((2(g-\epsilon)+1)\cdot\Omega)$ *have both dimension equal to* $g - \epsilon + 1$. *The sequence* $\dim(\mathcal{L}(m\cdot\Omega))$, *where* $m \geq 0$, *is as follows:*

$$1, 1, 2, 2, \ldots, g-1, g-1, g, g, g+1, g+2, g+3, \ldots$$

It is clear that the numerical semigroup of non-gaps is that of the natural numbers without the odd numbers smaller than $2g$.

12.2 AG Goppa codes

In this section, we construct an AG Goppa code using the basis computed in section 12.1.

Remark 12.4. *Note that, for* $p \leq \frac{m-t}{2}$, *the polynomials* x *and* x^{p^c} *in the basis of* $\mathcal{L}(D)$ *assume the same values in the field* $\mathbb{K} = \mathbb{F}_{p^c}$. *Similarly, for* $p \leq \frac{m-(d-t)}{2}$, *the polynomials* $\Psi(x,y)\cdot x$ *and* $\Psi(x,y)\cdot x^{p^c}$ *also assume the same values. It is important to consider this fact when constructing a Goppa code.*

Theorem 12.2. *Let* $\mathbb{F}$ *be a field of characteristic* $p \geq 2$, *let* $u(x)$ *be a monic polynomial of degree* t *and* $v(x)$ *be a polynomial with* $\deg(v(x)) < t$, *such that* $\gcd(u(x), u'(x), v(x)) = 1$, *and let* $P_s = (x_s, y_s)$ *be* n *pairs such that* $u(x_s) \neq 0$, *for all* $s = 1, \ldots, n$.

If $g \geq t$, *then for any* $g - t + 2 \leq k < n$, *the matrix* $G = (\gamma_{rs})$ *defined by*

$$\begin{cases} \gamma_{rs} = x_s^{r-1} & \text{for } 1 \leq r \leq \eta + 1 \\ \gamma_{rs} = \Psi(x_s, y_s)\cdot x_s^{r-\eta} & \text{for } \eta + 2 \leq r \leq k \end{cases} \left(\text{where } \eta = \left\lfloor \frac{k+g-1-t}{2} \right\rfloor\right) \tag{12.8}$$

is the generator matrix of a $[n, k, \delta]$ *Goppa code, with* $n - k + 1 - g \leq \delta \leq n - k + 1$, *and with*

$$\Psi(x_s, y_s) = \frac{y_s + v(x_s) + h(x_s)}{u(x_s)}$$

where $h(x) = 0$ *for* $p > 2$, *or* $h(x)$ *is an arbitrary non-zero polynomial with* $\deg(h(x)) \leq g$ *for* $p = 2$.

Proof. Consider the polynomial $c(x)$ of degree $2g + 1 - t$ such that

$$c(x_s) = \frac{v(x_s)^2 + h(x_s)v(x_s) - y_s^2 - y_s h(x_s)}{u(x_s)},$$

for any (x_s, y_s) with $s = 1, \ldots, n$. It is worth noting that the degree of $c(x)$ may be greater than n, meaning that it may be necessary to introduce additional pairs in order to compute $c(x)$.

Given the polynomial $c(x)$, we can construct a hyperelliptic curve of genus g defined by the equation

$$y^2 + yh(x) = f(x) = v(x)^2 + h(x)v(x) - c(x)u(x),$$

that passes through the n points (x_s, y_s) and the points belonging to the support of the divisor $\operatorname{div}(u(x), v(x))$.

The claims of the theorem follow from the fact that the functions considered in the theorem form a basis of the Riemann-Roch space $\mathcal{L}(D)$, where $D = \operatorname{div}(u(x), v(x)) + (k + g - 1) \cdot \Omega$, where the dimension of $\mathcal{L}(D)$ is equal to k. ∎

Remark 12.5. *Note that, as long as $k < g - t + 2$ and the n points $P_s = (x_s, y_s)$ where the functions of the basis of $\mathcal{L}(D)$ are evaluated have distinct abscissæ x_s, the Goppa code coincides with the Reed-Solomon code of parameters $[n, k, n - k + 1]$ on the n values $\{x_1, \ldots, x_n\} \subset \mathbb{F}_q$, where q is a power of a prime number $p \geq 2$.*

12.3 A small example

In this section, we present a small example of an AG Goppa code for hyperelliptic curves.

It is noteworthy that for any polynomials $u(x)$ and $v(x)$, where $u(x)$ has degree t, and $v(x)$ has degree smaller than t, and an arbitrary non-zero polynomial $h(x)$ (if $p = 2$), we can construct a hyperelliptic curve of arbitrary genus $g \geq \max\{t, \deg(h(x))\}$, defined by the equation $y^2 = v^2(x) + v(x)h(x) - c(x)u(x)$, for any polynomial $c(x)$ of degree $2g + 1 - t$, that passes through the support of the divisor $D = \Delta + m \cdot \Omega$, with $\Delta = \operatorname{div}(u(x), v(x))$ in Mumford representation. All of these curves determine the same Riemann Roch space $\mathcal{L}(D)$. This means that in order to provide a basis of the space $\mathcal{L}(D)$, one does not have to know the specific curve containing the support of D. Additionally, one does not need to explicitly provide the points in the support of D, which is an advantage in the construction of AG Goppa codes, as shown in the example 5.

Example 5. *Let $\mathbb{F}_{101}$ be a finite field, we choose the pair of polynomials $(u(x), v(x))$ such that $\deg(v(x))$ is less than $\deg(u(x))$ and $\gcd(u(x), u'(x), v(x)) = 1$. More precisely, we consider the pair $(u(x), v(x)) = (x^{11} + 1, x^6 + 1)$. Next, we consider the function*

$$\Psi(x, y) = \frac{y + v(x)}{u(x)} = \frac{y + x^6 + 1}{x^{11} + 1}.$$

Additionally, we choose five pairs (x_s, y_s) such that $x_r \neq x_l$ for any $r \neq l$, and $u(x_s) \neq 0$ for any s. Specifically, we choose $(15, 45)$, $(53, 48)$, $(58, 10)$, $(64, 13)$, and $(80, 2)$. By evaluating the functions $\{1, x, x^2, \Psi(x, y), x \cdot \Psi(x, y)\}$ on the ten points $(15, \pm 45)$, $(53, \pm 48)$, $(80, \pm 2)$, $(58, \pm 10)$, and $(64, \pm 13)$, one obtains the following

matrix

$$G := \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 15 & 15 & 53 & 53 & 80 & 80 & 58 & 58 & 64 & 64 \\ 23 & 23 & 82 & 82 & 37 & 37 & 31 & 31 & 56 & 56 \\ 73 & 41 & 35 & 92 & 1 & 45 & 99 & 71 & 48 & 21 \\ 85 & 9 & 37 & 28 & 80 & 65 & 86 & 78 & 42 & 31 \end{pmatrix},$$

which is a generator matrix of a code $\mathcal{C}$ *of length* 10*, and dimension* 5*. Since the all the* 5×5 *minors of* G *have full rank, it follows that the minimal distance of this code is equal to* 6*, making* $\mathcal{C}$ *a* $[10, 5, 6]_{101}$ *MDS code.*

In the following, we explicitly construct a hyperelliptic curve $\mathcal{H}$ *that realizes the code* $\mathcal{C}$ *as an AG Goppa Code. First, we note that the genus* g *of* $\mathcal{H}$ *must be at least equal to* $\deg(u(x))$*, that is, the degree of* $\mathcal{H}$ *must be at least equal to* 23 *(in particular, the degree of* $f(x)$*). By fixing* $g = \deg(u(x))$*, we have the five points* (x_s, y_s) *and the eleven points (in the affine plane) belonging to the support of* $\operatorname{div}(u(x), v(x))$*. As a result, we need eight* $(23 - 11 - 5 + 1)$ *additional points in order to construct the equation that defines the curve* $\mathcal{H}$*. Specifically, we arbitrarily choose eight pairs* (x_s, y_s)*, with* $s = 6, \ldots, 13$*, such that* $x_r \neq x_l$ *for any* $r \neq l$*, and* $u(x_s) \neq 0$*. For instance, we choose* $(48, 80)$*,* $(58, 91)$*,* $(64, 88)$*,* $(89, 16)$*,* $(95, 33)$*,* $(53, 4)$*,* $(51, 85)$*, and* $(71, 35)$*. Given these* 24 *points, the curve* $\mathcal{H}$ *of degree* 23*, which passes through the* 13 *points* (x_s, y_s) *and the eleven points (in the affine plane) of the support of* $\operatorname{div}(u(x), v(x))$*, is given by*

$$y^2 = v^2(x) - c(x)u(x),$$

where $c(x)$ *is the polynomial such that*

$$c(x_s) = \frac{v(x_s)^2 - y_s^2}{u(x_s)},$$

for $s = 1, \ldots, 13$*. Therefore, the curve* $\mathcal{H}$ *realizes the* $[10, 5, 6]_{101}$ *MDS code as the AG Goppa code defined by* $\mathcal{L}(D)$*, where* $D = \operatorname{div}(u(x), v(x)) + 15 \cdot \Omega$*, and the ten points* $(x_s, \pm y_s)$*, for* $s = 1, \ldots, 5$*.*

13 HL-codes and their decoding algorithm

In this chapter, we provide a brief introduction to Reed-Muller codes (or RM-codes), which were developed by D.E. Muller and I.S. Reed in 1954 [62, 64]. Since HL-codes are sub-codes of Reed-Muller codes, it is important to understand the basics of Reed-Muller codes. For more detailed information on Reed-Muller codes, we address the reader to chapter 13 of [57]. We address the reader to appendix A.4 and appendix B.4 for, respectively, the pseudocodes and the implementation codes of the algorithms used to define our cryptosystem. Additionally, in the latter appendix, we show the performance of our implementation.

13.1 Reed-Muller codes

The Reed-Muller codes can be easily defined in terms of Boolean functions, which are multivariate functions whose variables $\mathbf{v_i}$ are Boolean.

More precisely, Boolean functions are mathematical functions that take on only two possible values: 1 (true) or 0 (false). They are named after George Boole, an English mathematician who first formalized the concept of Boolean algebra in 1854, and as far the study of these functions has had many key contributions by mathematicians and computer scientists. Moreover, Boolean functions have important properties related to their algebraic structure, linearity, symmetry and symmetry-breaking, and degree of nonlinearity. These properties make Boolean functions useful in a wide range of applications, including digital logic, error-correcting codes, and cryptography. The study of Boolean functions has been an active area of research for many years and continues to be an important field of study in computer science, engineering, and mathematics.

The domain of a Boolean function is typically a set of binary inputs, such as a set of bits or a set of Boolean variables. In particular, a Boolean function is a function $f : \{0,1\}^n \to \{0,1\}$, where n is the number of variables of the function, 0 means false, and 1 means true. The set of all Boolean functions on a given domain is known as the Boolean algebra of that domain, which has several important properties such as commutativity, associativity, distributivity, and completeness. The Boolean algebra is closed under the basic logic operations of conjunction (AND), disjunction (OR), and negation (NOT) which are defined as follows:

$$\begin{aligned} f(x_1, x_2, \ldots, x_n) &= x_1 \wedge x_2 \wedge \ldots \wedge x_n, \\ g(x_1, x_2, \ldots, x_n) &= x_1 \vee x_2 \vee \ldots \vee x_n, \\ h(x_1, x_2, \ldots, x_n) &= \neg x_i, \end{aligned} \tag{13.1}$$

where $\wedge$ and $\vee$ are respectively the logical AND and OR operations, and $\neg$ is the logical NOT operation. Recall that the conjunction of Boolean variables is true (or

1) if and only if both variables are true (or 1), the disjunction is true (or 1) if at least one variable is true (or 1), and the negation is true (or 1) if the variable is false (or 0). It is worth to note that the algebraic structure of Boolean functions allows for the manipulation of Boolean expressions using the same rules as algebraic expressions.

One of the most important properties of Boolean functions is their linearity. A Boolean function is said to be linear if it satisfies the principle of superposition, which states that the output of the function is the sum of its inputs modulo 2, that is, if it satisfies the following equations:

$$\begin{aligned} f(cx_1, cx_2, \ldots, cx_n) &= cf(x_1, x_2, \ldots, x_n), \\ f(x_1 \oplus y_1, x_2 \oplus y_2, \ldots, x_n \oplus y_n) &= f(x_1, x_2, \ldots, x_n) \oplus f(y_1, y_2, \ldots, y_n), \end{aligned} \tag{13.2}$$

where $\oplus$ is the exclusive OR (XOR) operation, which is equivalent to the following operations: $a \oplus b = (\neg a \wedge b) \vee (a \wedge \neg b)$, with a and b being two Boolean variables. Linear Boolean functions have a number of desirable properties, such as the property that the outputs for each input combination can be represented by a linear combination of the input variables. This latter property makes easy to implement these functions in hardware, and having efficient decoding algorithms.

Additionally, Boolean functions have important properties related to their symmetry and symmetry-breaking. More precisely, a Boolean function is symmetric if it is invariant under permutations of its input variables. Symmetric Boolean functions are often used in the design of cryptographic systems (e.g. during the encoding) because they are resistant to certain types of attacks. On the other hand, symmetry-breaking Boolean functions are used to construct trapdoor functions, i.e., functions that are easy to compute in one direction but hard to compute in the reverse direction.

Lastly, Boolean functions also have important properties related to their degree of nonlinearity, which is a measure of how far the function is from being linear. A Boolean function with a high degree of nonlinearity is considered to be more resistant to certain types of attacks, such as linear and differential cryptanalysis.

Boolean functions are closely related to binary operations on binary vectors, i.e., vectors whose entries are either 0 or 1, as they can be defined in terms of such operations. In particular, the equivalence between Boolean functions and binary vectors can be understood through the concept of a truth table, i.e., a table that lists all possible input combinations for a Boolean function, along with the corresponding output value. Each row of the table corresponds to a specific input combination, and the output value is determined by the Boolean function. Furthermore, together with the linearity property, the Walsh-Hadamard Transform (WHT) and its inverse ensure that it is possible to transform any Boolean function into a binary vector and vice versa, through a linear transformation.

Therefore, a binary vector can also be used to represent a Boolean function, by specifying the output value of the function for each possible input combination.

The equivalence between Boolean functions and binary vectors can be formalized through the concept of a Boolean function space, that is, a vector space over the binary field, where each vector corresponds to a Boolean function. The dimension of this space is equal to the number of possible input combinations for the Boolean function.

One of the most fundamental binary operations on binary vectors is the dot product ($\times$), also known as the scalar product or inner product, which is defined as follows:

$$f \times g = \sum_{i=1}^{n} f_i \times g_i, \tag{13.3}$$

where $f = (f_1, \ldots, f_n)$ and $g = (g_1, \ldots, g_n)$ are binary vectors of length n. The dot product of two binary vectors is a scalar value, which is also a binary value.

Another important binary operation on binary vectors is the modulo-2 addition ($+$), which is defined as follows:

$$f + g = (f_1 + g_1, \ldots, f_n + g_n). \tag{13.4}$$

The result of this operation is also a binary vector of the same length as the input vectors.

Additionally, one may define the modulo-2 bitwise product ($\cdot$), which is equivalent to the bitwise logic AND, between two binary vectors as follows:

$$f \cdot g = (f_1 \cdot g_1, \ldots, f_n \cdot g_n). \tag{13.5}$$

Furthermore, the NOT operation of a binary vector f is equivalent to the complement to 1 of the vector f, i.e., $\neg f = \mathbf{1} + f = \overline{f}$, where $\mathbf{1}$ is a vector of ones of the same length of f.

In summary, there is a correspondence between the logic operations and the binary operations given by

- $f \wedge g = f \cdot g$;
- $f \oplus g = f + g$;
- $f \vee g = f + g + (f \cdot g)$;
- $\neg f = \overline{f} = \mathbf{1} + f$,

where the left-hand side represents the logic operations between Boolean functions, while the right-hand side represents the operations between the corresponding binary vectors. For instance, the logic OR between the Boolean variables f and g, i.e., $f \vee g$, is equivalent to the modulo-2 addition ($+$) between the binary vectors representing f, g, and $f \cdot g$ (the bitwise binary product). This correspondence can be verified using a truth table.

By a well-known theorem of Boolean algebra, any Boolean function f can be uniquely represented in its so called algebraic normal form (ANF), known also as Reed-Muller expansion.

Theorem 13.1 (cf. theorem 1 in §13 [57]). *Let $\{v_1, \ldots, v_m\}$ be a set of m Boolean variables. Any Boolean function $f(v_1, \ldots, v_m)$ can be uniquely expressed as a binary addition of powers of v_i as follows:*

$$f(v_1, \ldots, v_m) = \sum_{(e_1,\ldots,e_m)\in\{0,1\}^m} f(e_1, \ldots, e_m) \cdot {v_1}^{e_1} \cdot \ldots \cdot {v_m}^{e_m} = \sum_{I\subseteq\{1,\ldots,m\}} a_I \prod_{i\in I} v_i,$$

where $f(e_1, \ldots, e_m)$ *is the value of* f *at* $(e_1, \ldots, e_m)$*, the product of* ${v_i}^{e_i}$*, for* $i = 1, \ldots, m$*, is known as a minterm of* f*, and* ${v_i}^{e_i} = v_i$ *if* $e_i = 1$*, otherwise,* ${v_i}^{e_i} = \overline{v_i}$ *if* $e_i = 0$*.*

Proof. First, we provide the following property:

$$f(v_1, \ldots, v_m) = g(v_1, \ldots, v_{m-1}) + v_m \cdot h(v_1, \ldots, v_{m-1}),$$

where

$$\begin{aligned} g(v_1, \ldots, v_{m-1}) &= f(v_1, \ldots, v_{m-1}, 0), \\ h(v_1, \ldots, v_{m-1}) &= f(v_1, \ldots, v_{m-1}, 0) + f(v_1, \ldots, v_{m-1}, 1), \end{aligned}$$

which is true as

$$\begin{aligned} f(v_1, \ldots, v_{m-1}, 0) &= g = f(v_1, \ldots, v_{m-1}, 0) \qquad &\text{if} \quad v_m = 0, \\ f(v_1, \ldots, v_{m-1}, 1) &= g + g + f(v_1, \ldots, v_{m-1}, 1) \qquad &\text{if} \quad v_m = 1. \end{aligned}$$

Therefore, we can prove the theorem by induction using the property previously mentioned. Let m be equal to 1, then the claim follows as

$$\begin{aligned} f(v_1) &= f(0) + v_1 \cdot (f(0) + f(1)) = \\ &= f(0)(1 + v_1) + f(1) = \\ &= f(0)\overline{v_1} + f(1)v_1 = \\ &= \sum_{e_i \in \{0,1\}} f(e_i) v_1^{e_i}. \end{aligned}$$

By induction step, it holds that

$$f(v_1, \ldots, v_{m-1}) = \sum_{(e_1, \ldots, e_{m-1}) \in \{0,1\}^{m-1}} f(e_1, \ldots, e_{m-1}) \cdot {v_1}^{e_1} \cdot \ldots \cdot {v_{m-1}}^{e_{m-1}}.$$

Thus, it follows that

$$\begin{aligned} f(v_1, \ldots, v_m) &= f(v_1, \ldots, v_{m-1}, 0) + v_m \cdot (f(v_1, \ldots, v_{m-1}, 0) + f(v_1, \ldots, v_{m-1}, 1)) = \\ &= \overline{v_m} \cdot f(v_1, \ldots, v_{m-1}, 0) + v_m \cdot f(v_1, \ldots, v_{m-1}, 1). \end{aligned}$$

■

As a consequence of the above theorem, any Boolean function in the Boolean variables v_i can be represented as an expanded polynomial of the form

$$f(v_1, \ldots, v_m) = \sum_{I \subseteq \{1, \ldots, m\}} a_I \prod_{i \in I} v_i.$$

It is important to note that the function f can also be represented as a binary vector $\mathbf{f}$ of length 2^m given by the values taken by the function f at all of its 2^m arguments. This vector is commonly referred as the corresponding row of the truth table

associated with the function f. For instance, we have that

$$\begin{array}{lcllllllll} v_3 & = & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ v_2 & = & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ v_1 & = & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ \mathbf{f} & = & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{array}.$$

Thus, by applying the equivalence between Boolean operations and binary operations, one has that

$$\begin{aligned} \mathbf{f} \equiv f(v_1, \dots, v_m) &= \overline{v_1} \wedge \overline{v_2} \wedge \overline{v_3} \bigvee v_1 \wedge \overline{v_2} \wedge v_3 \bigvee v_1 \wedge v_2 \wedge v_3 = \\ &= 1 + v_1 + v_2 + v_3 + v_1v_2 + v_2v_3 + v_1v_2v_3. \end{aligned}$$

Note that, as stated in remark 13.1, by representing the Boolean variables v_i as their corresponding binary vectors in the truth table, the above vector $\mathbf{f}$ can be written as $\mathbf{1} + \mathbf{v_1} + \mathbf{v_2} + \mathbf{v_3} + \mathbf{v_1 v_2} + \mathbf{v_2 v_3} + \mathbf{v_1 v_2 v_3}$.

Remark 13.1. *It is important to note that each Boolean variable $\mathbf{v_i}$, with $i = 1, \dots, m$, can be represented as a binary vector in the truth table. Specifically, each Boolean variable can be represented as a binary vector in the form $\mathbf{v_i} = \left(\mathbf{0}^{2^{i-1}} \mid \mathbf{1}^{2^{i-1}}\right)^{\frac{n}{2^i}}$, where $\mathbf{0}^{2^{i-1}}$ is a vector of length 2^{i-1} consisting of zeros, and $\mathbf{1}^{2^{i-1}}$ is a vector of length 2^{i-1} consisting of ones. Hence, $\mathbf{v_i}$ is the concatenation of $\frac{n}{2^i}$ sequences of 2^{i-1} zeros and 2^{i-1} ones.*

In the following, we define a Reed-Muller code by using Boolean functions.

Definition 13.1 (Reed-Muller code)**.** *Let r and m be two positive integers such that $0 \leq r \leq m$, and let F be the set all binary vectors $\mathbf{f}$ such that $f(v_1, \dots, v_m)$ is a Boolean function in m variables v_i represented as a polynomial of degree at most r over $\mathbb{F}_2[v_1, \dots, v_m]/(v_1^2 - 1, \dots, v_m^2 - 1)$. The set F is a binary Reed-Muller code of order r, denoted as $\mathfrak{R}(r, m)$, of parameter $[n, k, d]$, where $n = 2^m$, $k = \sum\limits_{i=0}^{r} \binom{m}{i}$, and $d = 2^{m-r}$.*

For instance, the $\mathfrak{R}(1, m)$ code consists of all binary vectors of the form $\sum\limits_{i=0}^{m} a_i \mathbf{v_i}$, where $a_i \in \{0, 1\}$ and $\mathbf{v_0} = \mathbf{1} \in \mathbb{F}_2^n$. In general, the $\mathfrak{R}(r, m)$ code is given by the set of all the binary vectors of the form

$$a_0 \mathbf{v_0} + \sum_{\substack{L \subseteq \{1, \dots, m\} \\ |L| \leq r}} a_L \prod_{i \in L} \mathbf{v_i},$$

where $a_0, a_L \in \{0, 1\}$.

This is because the vectors $\mathbf{v_i}$ are linearly independent and thus represent the basis vectors of the Reed-Muller code.

In order to clearly define the generator matrix of a Reed-Muller code, we introduce the following auxiliary functions. For $i \in \mathbb{N}$, let ν be the function defined as

$$\nu(i) = \max_{t \in \mathbb{N}} \left\{ i \geq \sum_{j=0}^{t} \binom{m}{j} \right\}, \tag{13.6}$$

and let λ be the function defined as

$$\lambda(i) = i - \sum_{j=0}^{\nu(i)} \binom{m}{j}. \tag{13.7}$$

Additionally, let M be the set of integers $\{1,\ldots,m\}$, and let $\binom{M}{j}$ be the set of j-combination of the elements in M such that $e \in \binom{M}{j}$ is a tuple $e = (l_1,\ldots,l_j)$ of j indices. Furthermore, we define a natural ordering for $\binom{M}{j}$ such that, for any $(e_1,e_2) \in \binom{M}{j} \times \binom{M}{j}$, one has that

$$(l_{1,1},\ldots,l_{1,j}) = e_1 \leq e_2 = (l_{2,1},\ldots,l_{2,j}) \iff l_{1,s} \leq l_{2,s} \text{ for all } 1 \leq s \leq j. \tag{13.8}$$

In addition, we refer to the i-th element e_i of $\binom{M}{j}$ as the element such that

$$e_i \geq e_{i-1} \geq \ldots \geq e_1,$$

where e_1 is the minimum element in $\binom{M}{j}$.

Definition 13.2 (Generator matrix of a Reed-Muller code). *Let $\mathfrak{R}(r,m)$ be a $[n,k,d]_2$ Reed-Muller code of order r, with $0 \leq r \leq m$, and let M be the set of integers $\{1,\ldots,m\}$. A generator matrix $G \in \mathbb{F}_2^{k\times n}$ for this code is defined such that the i-th row of G, for $i = 0,\ldots,k-1$, is $G_i = \prod_{j\in e_{\lambda(i)}} \mathbf{v_j}$, where $e_{\lambda(i)} \in \binom{M}{\nu(i)+1}$.*

13.2 HL-codes

The HL-codes are self-dual codes C such that $\mathfrak{R}(\frac{m}{2}-1,m) \subset C \subset \mathfrak{R}(\frac{m}{2},m)$. For further details, we address the reader to [27, 38].

In order to properly define HL-codes, we need the following two definitions.

Definition 13.3 (Complement of a binary vector). *Let $\mathbf{v}$ be a vector in $\mathbb{F}_2^n$, where n is a positive integer. The complement at 1 of $\mathbf{v}$ is the vector $\mathbf{v}' = \mathbf{1} - \mathbf{v}$, where $\mathbf{1} \in \mathbb{F}_2^n$.*

Definition 13.4 (Complement-free set). *Let X be the set of all vectors in $\mathbb{F}_2^n$ of weight $t < n$, where n and t are positive integers. A subset Y of X is complement-free if for each $\mathbf{v} \in Y \implies \mathbf{1} - \mathbf{v} \notin Y$, where $\mathbf{1} \in \mathbb{F}_2^n$. If $\mathrm{card}(Y) = \frac{1}{2}\binom{n}{t}$, then Y is maximal.*

Remark 13.2. *Note that, for a complement-free set Y in definition 13.4, if $\{j_1,\ldots,j_t\} \subseteq \{1,\ldots,n\}$ is the set of indices of $(y_1,\ldots,y_n) = \mathbf{y} \in Y$ such that $y_{j_r} = 1$, with $r = 1,\ldots,t$, then we may represent $\mathbf{y}$ with the tuple $(j_1,\ldots,j_t)$.*

From now on, we consider the elements of a complement-free set Y as tuple of indices as in remark 13.2.

Definition 13.5 (HL-code). *Let m be a positive even integer, and let Y be a maximal random complement-free set whose vectors in $\mathbb{F}_2^m$ have weight equal to $\frac{m}{2}$. A HL-code of parameter $n = 2^m$, $k = 2^{m-1}$, and $d = 2^{\frac{m}{2}}$ is a Reed-Muller code $\mathfrak{R}(\frac{m}{2}-1,m)$, whose generator matrix is extended by the vectors obtained by the products $\prod_{i=1}^{\frac{m}{2}} \mathbf{v_{j_i}}$,*

where the vectors $\mathbf{v_i}$ *are the basis vectors of the Reed-Muller code, and* $\left(j_1, \ldots, j_{\frac{m}{2}}\right)$ *is the tuple representation of the vectors* $\mathbf{y} \in Y$ *as in remark 13.2.*

The randomness of the complement-free set Y in definition 13.5 makes these codes useful for embedding into code-based cryptosystems such as the McEliece cryptosystem [58].

In order to define the generator matrix for a HL-code, we define the following auxiliary function:

$$f(i) = \begin{cases} e_{\lambda(i)} \in \binom{M}{\nu(i)+1} & \text{if} \quad 1 \leq i < k - \frac{1}{2}\binom{m}{m/2}, \\ e_{\lambda(i)} \in Y\text{-set} & \text{if} \quad k - \frac{1}{2}\binom{m}{m/2} \leq i < k, \end{cases} \tag{13.9}$$

where $M = \{1, \ldots, m\}$, $e_{\lambda(i)} \in \binom{M}{\nu(i)+1}$ is taken with the rule in equation (13.8), and $e_{\lambda(i)} \in Y$-set is the tuple representation of the $\lambda(i)$-th element of the random complement-free set Y (see remark 13.2).

Example 6. *Let* m *be equal to* 6*, and let* M *be the set of integers* $\{1, 2, 3, 4, 5, 6\}$*. If* $i = 7$*, then* $f(7) = e_{\lambda(7)} = (1, 2)$ *is the first element (*$\lambda(7) = 1$*) of* $\binom{M}{2}$*. Furthermore, if* $i = 28$*, then* $f(28) = e_{\lambda(28)}$ *is equal to the sixth element (*$\lambda(28) = 6$*) of the random complement-free set* $Y \subset \binom{M}{3}$*.*

Definition 13.6 (Generator matrix for a HL-code)**.** *Let* C *be a* $[n, k, d]_2$ *HL-code, where* $n = 2^m$*,* $k = 2^{m-1}$*, and* $d = 2^{\frac{m}{2}}$*, and let* Y *be the maximal complement-free set used to construct* C*. A generator matrix* $G \in \mathbb{F}_2^{k \times n}$ *for* C *is defined such that the* i*-th row of* G*, for* $i = 0, \ldots, k-1$*, is* $G_i = \prod_{j \in f(i)} \mathbf{v_j}$*, where* f *is the function in equation (13.9).*

Note that, every codeword belonging to a linear code can be represented as a linear combination of the rows of the generator matrix of the code. Specifically, a codeword $\mathbf{x}$ of a HL-code can be written as

$$\mathbf{x} = \mathbf{a} \cdot G = a_0 \mathbf{v_0} + \sum_{i=1}^{k-1} a_i \left(\sum_{j \in f(i)} \mathbf{v_j} \right), \tag{13.10}$$

where $(a_0, \ldots, a_{k-1}) = \mathbf{a} \in \mathbb{F}_2^k$ is a binary weight vector, and f is the function in equation (13.9).

13.3 Decoding

In this section, we describe the process of decoding a Reed-Muller code.

Definition 13.7 (Decode a linear code)**.** *Let* C *be a* $[n, k, d]_q$ *a linear code whose generator matrix is* $G \in \mathbb{K}^{k \times n}$*, where* $\mathbb{K}$ *is a finite field of cardinality equal to a prime power* q*. Decoding a word* $\mathbf{x} = \mathbf{a} \cdot G + \mathbf{e}$*, where* $\mathbf{a} \in \mathbb{K}^k$*, and* $\mathbf{e} \in \mathbb{K}^n$ *is an error word whose Hamming weight* $\mathrm{wt}(\mathbf{e})$ *is at most* $\left\lfloor \frac{d-1}{2} \right\rfloor$*, consist in computing the error word* $\mathbf{e}$ *and returning the correct codeword* $\mathbf{a} \cdot G$ *or the word* $\mathbf{a}$*.*

The Reed-Muller decoding algorithm is a **majority-based binary decoding** method that relies on the structure of the generator matrix G.

It should be noted that, as stated in equation (13.10), a codeword $\mathbf{x}$ of a HL-code may be represented a linear combination of bitwise products of the basis vector $\mathbf{v_i}$ of the code. A bitwise product of j different vectors $\mathbf{v_i}$ is considered to be an element

of **degree** j for the codeword $\mathbf{x}$. Hence, every coefficient a_i in equation (13.10) is the **coefficient of degree** j of $\binom{m}{j}$ bitwise binary products of the basis vectors $\mathbf{v_i}$, where $1 \leq i \leq m$.

Given the expression in equation (13.10), it is possible to determine a set of binary equations, known as **redundancy relations**, that are related to each coefficient a_i. These equation takes the form:

$$\left\{ \sum_{e\in I_1} x_e, \sum_{e\in I_2} x_e, \ldots, \sum_{e\in I_l} x_e \right\} \tag{13.11}$$

where $(x_0,\ldots,x_{n-1}) = \mathbf{x}$ represents the codeword, and for $1 \leq s \leq l$, $I_s \neq \emptyset$, $\bigcup_{s=1}^{l} I_s = \{0,1,\ldots,n-1\}$, and $I_h \cap I_s = \emptyset$ whenever $h \neq s$. Having the set in equation (13.11), if r is the number of the equations in (13.11) whose outcome is equal to 1, then

$$a_i = \begin{cases} 1 \text{ if } r > \frac{l}{2}, \\ 0 \text{ if } r < \frac{l}{2}. \end{cases}$$

Note that if $r = \frac{l}{2}$, then it is not possible to determine the correct value of a_i (as stated in theorem 13.2 and corollary 13.3).

Theorem 13.2. *Let a_j be a coefficient of the word $\mathbf{x} = \mathbf{a}\cdot G + \mathbf{e}$, where $G \in \mathbb{F}_2^{k\times n}$ is the generator matrix of a HL-code, and $\mathbf{e}$ is an error vector of weight $t \leq \left\lfloor \frac{d-1}{2} \right\rfloor$. Let*

$$S_i = \sum_{e\in I_i} x_e, \qquad 1 \leq i \leq l,$$

be the i-th redundancy relation of a_j, where $I_i \neq \emptyset$, for every i, $\bigcup_{i=1}^{l} I_i = \{1,\ldots,n\}$, and $I_h \cap I_s = \emptyset$ whenever $h \neq s$. There are at least $l-t$ redundancy relations for a_j with the same outcome, that is, $S_h = S_s$ for any $h, s \in R \subseteq \{1,\ldots,l\}$, with $|R| \geq l-t$.

Proof. Since the redundancy relations for each coefficient a_j form a partition of the n components of the vector $\mathbf{x}$, we have that each the component $e_i = 1$ of $\mathbf{e}$ affects only one of the redundancy relations. Therefore, if the weight of $\mathbf{e}$ is equal to 1, then only one relation has a different outcome compared to the other outcomes.

If the weight of $\mathbf{e}$ is equal to 2, then either zero or two relations outcomes differ from the other outcomes. This is because, if $e_r = 1$ and $e_s = 1$ affects the same relation S_i, then in S_i, one has that $e_r + e_s = 0$ and the outcome of S_i does not change. However, if $e_r = 1$ and $e_s = 1$ affects two different relations, then both relations outcomes differ from the other outcomes.

In general, by induction, if the weight of the error vector $\mathbf{e}$ is equal to t, then there are at least $l-t$ relations with the same outcome. ■

Corollary 13.3. *Let a_j be a coefficient of degree u of the word $\mathbf{x} = \mathbf{a}\cdot G + \mathbf{e}$, where $G \in \mathbb{F}_2^{k\times n}$ is the generator matrix of a HL-code whose parameter m is an even number, with $n = 2^m$, and $\mathbf{e}$ is an arbitrary binary error vector of weight $t \leq \left\lfloor \frac{d-1}{2} \right\rfloor$. Let l be the number of redundancy relations for a_j, and let r be the number of redundancy*

relations for a_j whose outcomes are equal to 1. If $r > \frac{l}{2}$, then $a_j = 1$, while if $r < \frac{l}{2}$, then $a_j = 0$.

Proof. Given that m is an even number, it follows that $t \le \left\lfloor \frac{d-1}{2} \right\rfloor = \left\lfloor \frac{2^{\frac{m}{2}}-1}{2} \right\rfloor = 2^{\frac{m}{2}-1} - 1$. Additionally, it can be easily verified that $l = \frac{n}{2^u} = \frac{2^m}{2^u} = 2^{m-u}$ (see equation (13.15)), where $1 \le u \le \frac{m}{2}$. As a result, we have that $2^{\frac{m}{2}} \le l \le 2^{m-1}$ and $t < \frac{l}{2}$. In light of this and theorem 13.2, there are at least $r = l - t > \frac{l}{2}$ redundancy relations with the same outcome $b \in \{0,1\}$, and a_j is exactly equal to b. ■

Remark 13.3. *As a consequence of theorem 13.2 and corollary 13.3, if one has that $\mathrm{wt}(\mathbf{e}) \le \left\lfloor \frac{d-1}{2} \right\rfloor$, then the number l of redundancy relations for a_j with the same outcome cannot be exactly equal to $\frac{l}{2}$. More specifically, for the coefficient a_j with $j = 1, \ldots, k-1$, if $r = \frac{l}{2}$, then $\mathrm{wt}(\mathbf{e}) > \left\lfloor \frac{d-1}{2} \right\rfloor$ and it is not possible to determine the correct value of a_j.*

The decoding process begins with the highest-degree coefficients a_i and proceeds iteratively to the lowest-degree ones.

Specifically, since the elements of highest-degree in a HL-code have a degree equal to $\frac{m}{2}$ (in particular, the elements from the complement-free set defined in definition 13.4), these elements have to be decoded from $\mathbf{x} = \mathbf{a} \cdot G + \mathbf{e}$, and then removed from $\mathbf{x}$ in order to repeat the decoding process on

$$\mathbf{x}' = \mathbf{x} - \sum_{T \in Y\text{-set}} a_T \prod_{i \in T} \mathbf{v_i},$$

where Y-set is the complement-free set of indices T associated with the coefficients a_T of degree $\frac{m}{2}$. Since it is possible to determine the value of each a_T from corollary 13.3, this process is repeated by identifying the next highest-degree coefficients a_i, and removing them from $\mathbf{x}'$. At the end of this process, one is left with the vector $\mathbf{x}' = a_0\mathbf{v_0} + \mathbf{e}$. However, since $\mathbf{v_0} = \mathbf{1} \in \mathbb{F}_2^n$, in order to find a_0, one has to count the number r of ones in $\mathbf{x}'$ (as stated in theorem 13.4). Since $\mathbf{x}' \in \mathbb{F}_2{}^n$, one has that

$$a_0 = \begin{cases} 1 \text{ if } r > \frac{n}{2}, \\ 0 \text{ if } r < \frac{n}{2}. \end{cases}$$

Note that if $r = \frac{n}{2}$, then a_0 cannot be determined and, as a result, the decoding of $\mathbf{x}$ is unsuccessful. Once the value of a_0 is determined, the error vector can be computed as $\mathbf{e} = \mathbf{x}' - a_0\mathbf{v_0}$, and the codeword can be recovered as $\mathbf{a} \cdot G = \mathbf{x} - \mathbf{e}$.

Theorem 13.4. *Let $\mathbf{a} = (a_0, a_1, \ldots, a_{k-1})$ be an arbitrary binary weight vector, and let $\mathbf{x} = \mathbf{a} \cdot G + \mathbf{e}$ be the codeword $\mathbf{a} \cdot G$ modified by an arbitrary binary vector $\mathbf{e}$ such that $\mathrm{wt}(\mathbf{e}) \le \left\lfloor \frac{d-1}{2} \right\rfloor$, where $G \in \mathbb{F}_2^{k \times n}$ is the generator matrix of a HL-code. The value of the coefficient a_0 can be determined by evaluating the weight of the vector $a_0\mathbf{v_0} + \mathbf{e}$. Specifically, if $\mathrm{wt}(a_0\mathbf{v_0} + \mathbf{e}) > \frac{n}{2}$, then $a_0 = 1$, while if $\mathrm{wt}(a_0\mathbf{v_0} + \mathbf{e}) < \frac{n}{2}$, then $a_0 = 0$.*

Proof. According to theorem 13.2 and corollary 13.3, it is possible to determine each coefficient a_j, for $j = 1, \ldots, k-1$, by using the redundancy relations of a_j. Therefore, one can compute the vector $\mathbf{x}' = \mathbf{x} - \sum_{i=1}^{k-1} a_i \prod_{j \in f(i)} \mathbf{v_j} = a_0\mathbf{v_0} + \mathbf{e} \in \mathbb{F}_2^n$. Given that the

weight t of the error vector is such that $t \leq 2^{\frac{m}{2}-1} - 1$ and $\frac{n}{2} = 2^{m-1}$, it follows that $t < \frac{n}{2}$. Therefore, if $a_0 = 0$, then $\mathbf{x}' = \mathbf{e}$ and $\mathrm{wt}(\mathbf{x}') = t < \frac{n}{2}$, while if $a_0 = 1$, then $\mathbf{x}' = \mathbf{v_0} + \mathbf{e}$ and $\mathrm{wt}(\mathbf{x}') = n - t > \frac{n}{2}$. ■

Remark 13.4. *Note that theorem 13.2, corollary 13.3 and theorem 13.4 demonstrate that the majority rule can correctly determine each coefficient a_j, for $j = 0, \ldots, k-1$.*

Remark 13.5. *Given that the correct value of each coefficient a_j can be determined at each step, it is possible to store these results in order to directly return the weight vector* $\mathbf{a}$*, thus avoiding the computation of* $\mathbf{a}$ *from the vector* $\mathbf{a} \cdot G$*.*

The redundancy relations related to a coefficient a_i can be determined by a recursive relation (denoted as Δ by I.S. Reed in [64]). In order to describe this relation, we define the function

$$\begin{array}{rrl} \psi\colon & \mathbb{N} \times \mathbb{N} & \longrightarrow \{0,1\} \\ & (i,k) & \longmapsto i_{k-1} \end{array}, \tag{13.12}$$

where i_{k-1} is the k-th least significant bit (LSB) of the binary representation of the integer i, that is, $i = (i_{n-1}, \ldots, i_k, \ldots, i_1, i_0)_2$, and the function

$$\begin{array}{rrl} \phi\colon & \mathbb{N} \times \mathbb{N} & \longrightarrow \mathbb{N} \\ & (i,k) & \longmapsto i + (-1)^{\psi(i,k)} \cdot 2^{k-1} \end{array}. \tag{13.13}$$

Hence, $\phi(i,k)$ changes the k-th LSB of the integer i. Specifically, if $\psi(i,k)$ is equal to 0, then $\phi(i,k) = i + 2^{k-1}$, while if $\psi(i,k)$ is equal to 1, then $\phi(i,k) = i - 2^{k-1}$.

We also extend the function ϕ as follow:

$$\begin{array}{rrl} \Phi\colon & \mathbb{N} \times \mathbb{N}^l & \longrightarrow \mathbb{N} \\ & (i,(k_1,\ldots,k_l)) & \longmapsto i + \sum\limits_{j=1}^{l} (-1)^{\psi(i,k_j)} \cdot 2^{k_j - 1} \end{array}. \tag{13.14}$$

Therefore, the function $\Phi(i,(k_1,\ldots,k_l))$ changes, for $s = 1, \ldots, l$, the k_s-th LSB of the integer i.

Hence, the recursive relation used to find the redundancy relations is as follows:

$$\begin{array}{rcrcr} \underset{j}{\Delta}\, x_i & = & x_i & + & x_{\phi(i,j)}, \\ \underset{(j_1,\ldots,j_t)}{\Delta} x_i & = & \underset{(j_1,\ldots,j_{t-1})}{\Delta} x_i & + & \underset{(j_1,\ldots,j_{t-1})}{\Delta} x_{\phi(i,j_t)}, \end{array} \tag{13.15}$$

for $i = 0, \ldots, k-1$, and $(j_1, \ldots, j_t) \in \binom{M}{t}$, where $M = \{1, \ldots, m\}$.

More specifically, by applying the recursion in equation (13.15), one obtains that

$$\underset{(j_1,\ldots,j_t)}{\Delta} x_i = x_i + \sum_{r=1}^{t} \left(\sum_{e \in \binom{M}{r}} x_{\Phi(i,e)} \right). \tag{13.16}$$

The operator Δ calculates the redundancy relations for each coefficient a_j in the word $\mathbf{x} = \mathbf{a} \cdot G + \mathbf{e}$, with G the generator matrix of the code. In particular, for $i = 0, \ldots, k-1$, if $1 \leq j \leq m$, then the redundancy relations for the coefficients a_j

are computed with $\underset{j}{\Delta} x_i$, while if $m < j < k$, then the redundancy relations for the coefficient a_j are computed with $\underset{(j_1,\dots,j_t)}{\Delta} x_i$, where $(j_1, \dots, j_t) \in \binom{M}{t}$.

In the following example, we show how to find the redundancy relations for each coefficient a_i both with and without the use of the operator Δ.

Example 7. *Let m be equal to 4, so $n = 2^m = 16$, and $k = 2^{m-1} = 8$. Furthermore, suppose for simplicity that the Y-set is equal to $((1,4),(1,3),(1,2))$ in this order (see remark 13.2). In this case, we have that*

$$G = \begin{pmatrix} \mathbf{v_0} \\ \mathbf{v_1} \\ \mathbf{v_2} \\ \mathbf{v_3} \\ \mathbf{v_4} \\ \mathbf{v_1} \cdot \mathbf{v_4} \\ \mathbf{v_1} \cdot \mathbf{v_3} \\ \mathbf{v_1} \cdot \mathbf{v_2} \end{pmatrix} = \begin{pmatrix} 1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1 \\ 0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1 \\ 0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1 \\ 0&0&0&0&1&1&1&1&0&0&0&0&1&1&1&1 \\ 0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1 \\ 0&0&0&0&0&0&0&0&0&1&0&1&0&1&0&1 \\ 0&0&0&0&0&1&0&1&0&0&0&0&0&1&0&1 \\ 0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1 \end{pmatrix}.$$

Therefor, we have that

$$\mathbf{a} \cdot G = a_0\mathbf{v_0} + a_1\mathbf{v_1} + a_2\mathbf{v_2} + a_3\mathbf{v_3} + a_4\mathbf{v_4} + a_5\mathbf{v_1} \cdot \mathbf{v_4} + a_6\mathbf{v_1} \cdot \mathbf{v_3} + a_7\mathbf{v_1} \cdot \mathbf{v_2}.$$

In order to determine each a_i, one does not take into account the error vector $\mathbf{e}$, that is, one takes $\mathbf{x} = \mathbf{a} \cdot G$, and not $\mathbf{x} = \mathbf{a} \cdot G + \mathbf{e}$, as the majority rule for the redundancy relations will give back the correct value of each coefficient a_i. Therefore, the components x_i of the vector $\mathbf{x} = \mathbf{a} \cdot G$ are

$$\begin{aligned} x_0 &= a_0, & x_8 &= a_0 + a_4, \\ x_1 &= a_0 + a_1, & x_9 &= a_0 + a_1 + a_4 + a_5, \\ x_2 &= a_0 + a_2, & x_{10} &= a_0 + a_2 + a_4, \\ x_3 &= a_0 + a_1 + a_2 + a_7, & x_{11} &= a_0 + a_1 + a_2 + a_4 + a_5 + a_7, \\ x_4 &= a_0 + a_3, & x_{12} &= a_0 + a_3 + a_4, \\ x_5 &= a_0 + a_1 + a_3 + a_6, & x_{13} &= a_0 + a_1 + a_3 + a_4 + a_5 + a_6, \\ x_6 &= a_0 + a_2 + a_3, & x_{14} &= a_0 + a_2 + a_3 + a_4, \\ x_7 &= a_0 + a_1 + a_2 + a_3 + a_6 + a_7, & x_{15} &= a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7. \end{aligned} \tag{13.17}$$

Initially, we begin by determining the redundancy relations of the highest-degree coefficients a_5*,* a_6*, and* a_7 *using the equation* (13.17)*. We find that*

$$
\begin{aligned}
a_5 &= x_0 + x_1 + x_8 + x_9 = & a_6 &= x_0 + x_1 + x_4 + x_5 = \\
&= x_2 + x_3 + x_{10} + x_{11} = & &= x_2 + x_3 + x_6 + x_7 = \\
&= x_4 + x_5 + x_{12} + x_{13} = & &= x_8 + x_9 + x_{12} + x_{13} = \\
&= x_6 + x_7 + x_{14} + x_{15}, & &= x_{10} + x_{11} + x_{14} + x_{15}, \\
a_7 &= x_0 + x_1 + x_2 + x_3 = \\
&= x_4 + x_5 + x_6 + x_7 = \\
&= x_8 + x_9 + x_{10} + x_{11} = \\
&= x_{12} + x_{13} + x_{14} + x_{15}.
\end{aligned} \tag{13.18}
$$

In order to proceed to the next highest-degree coefficient, we consider the word

$$\mathbf{x}' = \mathbf{x} - (a_5\mathbf{v_1} \cdot \mathbf{v_4} + a_6\mathbf{v_1} \cdot \mathbf{v_3} + a_7\mathbf{v_1} \cdot \mathbf{v_2}).$$

Since $a_5 = a_6 = a_7 = 0$ *for* $\mathbf{x}'$*, we can remove them from the equation* (13.17) *and we find that*

$$
\begin{aligned}
x'_0 &= a_0, & x'_8 &= a_0 + a_4, \\
x'_1 &= a_0 + a_1, & x'_9 &= a_0 + a_1 + a_4, \\
x'_2 &= a_0 + a_2, & x'_{10} &= a_0 + a_2 + a_4, \\
x'_3 &= a_0 + a_1 + a_2, & x'_{11} &= a_0 + a_1 + a_2 + a_4, \\
x'_4 &= a_0 + a_3, & x'_{12} &= a_0 + a_3 + a_4, \\
x'_5 &= a_0 + a_1 + a_3, & x'_{13} &= a_0 + a_1 + a_3 + a_4, \\
x'_6 &= a_0 + a_2 + a_3, & x'_{14} &= a_0 + a_2 + a_3 + a_4, \\
x'_7 &= a_0 + a_1 + a_2 + a_3, & x'_{15} &= a_0 + a_1 + a_2 + a_3 + a_4.
\end{aligned} \tag{13.19}
$$

We determine the redundancy relations of the next coefficients a_1*,* a_2*,* a_3 *and* a_4 *from the equation* (13.19)*, and we find that*

$$
\begin{aligned}
a_1 &= x'_0 + x'_1 &&= x'_2 + x'_3 &&= x'_4 + x'_5 &&= x'_6 + x'_7 = \\
&= x'_8 + x'_9 &&= x'_{10} + x'_{11} &&= x'_{12} + x'_{13} &&= x'_{14} + x'_{15}, \\
a_2 &= x'_0 + x'_2 &&= x'_1 + x'_3 &&= x'_4 + x'_6 &&= x'_5 + x'_7 = \\
&= x'_8 + x'_{10} &&= x'_9 + x'_{11} &&= x'_{12} + x'_{14} &&= x'_{13} + x'_{15}, \\
a_3 &= x'_0 + x'_4 &&= x'_1 + x'_5 &&= x'_2 + x'_6 &&= x'_3 + x'_7 = \\
&= x'_8 + x'_{12} &&= x'_9 + x'_{13} &&= x'_{10} + x'_{14} &&= x'_{11} + x'_{15}, \\
a_4 &= x'_0 + x'_8 &&= x'_1 + x'_9 &&= x'_2 + x'_{10} &&= x'_3 + x'_{11} = \\
&= x'_4 + x'_{12} &&= x'_5 + x'_{13} &&= x'_6 + x_{14} &&= x_7 + x_{15}.
\end{aligned} \tag{13.20}
$$

Therefore, using the equations (13.18) *and* (13.20) *on* $\mathbf{x}$ *and* $\mathbf{x}'$ *respectively, we determine*

$$\mathbf{x}'' = \mathbf{x}' - (a_1\mathbf{v_1} + a_2\mathbf{v_2} + a_3\mathbf{v_3} + a_4\mathbf{v_4}) = a_0\mathbf{v_0}.$$

Recall that, as stated in theorem 13.2, corollary 13.3 and theorem 13.4, if $\mathbf{x} = \mathbf{a} \cdot G + \mathbf{e}$*, then the majority rule guarantees that* $\mathbf{x}'' = a_0 \mathbf{v_0} + \mathbf{e}$.

From here on, we apply the operator Δ *in order to compute the redundancy relations of each coefficient* a_i*. For instance, we directly find the first relation of* a_1 *as*

$$\underset{1}{\Delta}\, x_0 = x_0 + x_{\phi(0,1)} = x_0 + x_1.$$

Recall that, as previously stated in equation (13.11), the redundancy relations form a partition of the components $\{x_0, \dots, x_{15}\}$ *of the vector* $\mathbf{x}$*. Therefore, the next relation of* a_1 *is given by*

$$\underset{1}{\Delta}\, x_2 = x_2 + x_{\phi(2,1)} = x_2 + x_3.$$

Hence, after the computation of each redundancy relation, one is aware of the next component x_i *to consider. Note that calculating* $\underset{1}{\Delta}\, x_1$ *is futile as*

$$\underset{1}{\Delta}\, x_1 = x_1 + x_{\phi(1,1)} = x_1 + x_0 = \underset{1}{\Delta}\, x_0.$$

In order to summarize, by using the function f *defined in equation (13.9), we have that the redundancy relations for each coefficient* a_i *are as follows:*

$$\begin{aligned}
a_1 &= \underset{f(1)}{\Delta}\, x_0 = \underset{f(1)}{\Delta}\, x_2 = \underset{f(1)}{\Delta}\, x_4 = \underset{f(1)}{\Delta}\, x_6 = \\
&= \underset{f(1)}{\Delta}\, x_8 = \underset{f(1)}{\Delta}\, x_{10} = \underset{f(1)}{\Delta}\, x_{12} = \underset{f(1)}{\Delta}\, x_{14}, \\
\\
a_2 &= \underset{f(2)}{\Delta}\, x_0 = \underset{f(2)}{\Delta}\, x_1 = \underset{f(2)}{\Delta}\, x_4 = \underset{f(2)}{\Delta}\, x_5 = \\
&= \underset{f(2)}{\Delta}\, x_8 = \underset{f(2)}{\Delta}\, x_9 = \underset{f(2)}{\Delta}\, x_{12} = \underset{f(2)}{\Delta}\, x_{13}, \\
\\
a_3 &= \underset{f(3)}{\Delta}\, x_0 = \underset{f(3)}{\Delta}\, x_1 = \underset{f(3)}{\Delta}\, x_2 = \underset{f(3)}{\Delta}\, x_3 = \\
&= \underset{f(3)}{\Delta}\, x_8 = \underset{f(3)}{\Delta}\, x_9 = \underset{f(3)}{\Delta}\, x_{10} = \underset{f(3)}{\Delta}\, x_{11}, \\
\\
a_4 &= \underset{f(4)}{\Delta}\, x_0 = \underset{f(4)}{\Delta}\, x_1 = \underset{f(4)}{\Delta}\, x_2 = \underset{f(4)}{\Delta}\, x_3 = \\
&= \underset{f(4)}{\Delta}\, x_4 = \underset{f(4)}{\Delta}\, x_5 = \underset{f(4)}{\Delta}\, x_6 = \underset{f(4)}{\Delta}\, x_7, \\
\\
a_5 &= \underset{f(5)}{\Delta}\, x_0 = \underset{f(5)}{\Delta}\, x_2 = \underset{f(5)}{\Delta}\, x_4 = \underset{f(5)}{\Delta}\, x_6, \\
\\
a_6 &= \underset{f(6)}{\Delta}\, x_0 = \underset{f(6)}{\Delta}\, x_2 = \underset{f(6)}{\Delta}\, x_8 = \underset{f(6)}{\Delta}\, x_{10}, \\
\\
a_7 &= \underset{f(7)}{\Delta}\, x_0 = \underset{f(7)}{\Delta}\, x_4 = \underset{f(7)}{\Delta}\, x_8 = \underset{f(7)}{\Delta}\, x_{12}.
\end{aligned} \tag{13.21}$$

For instance, the first redundancy relation for a_6 is computed as $\underset{f(6)}{\Delta} x_0$. Since $f(6) = e_{\lambda(6)} = e_2 = (1,3) \in Y$-set, we have that

$$\begin{aligned}
\underset{f(6)}{\Delta} x_0 = \underset{(1,3)}{\Delta} x_0 &= \underset{1}{\Delta} x_0 + \underset{1}{\Delta} x_{\phi(0,3)} = \\
&= \Big(x_0 + x_{\phi(0,1)}\Big) + \Big(x_{\phi(0,3)} + x_{\Phi(0,(1,3))}\Big) = \\
&= x_0 + x_1 + x_4 + x_5 = \\
&= x_0 + \sum_{r=1}^{2} \left(\sum_{e \in \binom{(1,3)}{r}} x_{\Phi(0,e)} \right) = x_0 + x_{\Phi(0,(1))} + x_{\Phi(0,(3))} + x_{\Phi(0,(1,3))},
\end{aligned}$$

where the last equality comes from the expansion of the operator Δ in equation (13.16). Note that, by using the operator Δ, it is not necessary to explicitly find the redundancy relations as in equations (13.17) and (13.19).

Conclusions

In this chapter, we present the conclusions drawn from the research conducted for this thesis. In part II, we provided an explicit map to define easily the group law for an elliptic curve in Weierstrass form with the ground field $\mathbb{Z}/p^k\mathbb{Z}$. We also extended the exponential map Exp for elliptic curves in Weierstrass form to Edwards curves over local fields. Based on this extension, we produced an article that has been published in [31]. Additionally, we provided the inverse of the exponential map, denoted as Log, for elliptic curves in Weierstrass form to define a new algorithm to compute the elliptic curve discrete logarithm by exploiting the structure of $\mathcal{J}_{\mathbb{Z}/p^k\mathbb{Z}}(\mathcal{W})$. Lastly, we extended the results obtained for elliptic curves in Weierstrass form over local fields to hyperelliptic curves. In this case, however, we have seen that the extension is not possible using the Cantor-Koblitz algorithm to add two divisors belonging to the Jacobian of these curves since this algorithm requires computing the greatest common divisor (GCD) between two polynomials, and the GCD does not commute with the modulo-reduction operation.

In part III, we focused on coding theory. Specifically, we provided a basis for the Riemann-Roch space $\mathcal{L}(D)$ for a divisor D belonging to the Jacobian of either an Edwards curve or a hyperelliptic curve over finite fields. Moreover, we used these bases to compute the generator matrices for an algebraic-geometric Goppa code for these curves. We proved that a basis for the Riemann-Roch space for hyperelliptic curves involves the Mumford representation of the divisors of the Jacobian of these curves. In particular, the Mumford representation provides an elegant and simple formulation for the theorems that we provided, freeing the user to effectively know the points belonging to the support of the divisor used to compute the basis. This property is particularly useful when one wants to construct an algebraic-geometric Goppa code, as one may generate a pair of random polynomials that, under particular assumptions, allows computing a generator matrix for this code. Lastly, we extended the basic decoding algorithm for Reed-Muller codes to the HL-codes to develop a quantum-safe code-based cryptosystem. We rewrote the original Reed's decoding algorithm to provide a simplified formulation and gave theorems and corollaries that prove its correctness. We implemented this algorithm and embedded it into a cryptosystem of McEliece-type, showing that its performance is competitive with other code-based cryptosystems of the same type. The results of the chapters in part III have been collected in three articles submitted for possible publication.

In conclusion, this thesis provides original insights into the area of cryptography. The results of our work can find application to enhance the security of modern cryptosystems. While there are limitations to our study, it can contribute to a more comprehensive understanding of these fields. For future research, it would be interesting to explore other methodologies to extend our results in chapter 8 to hyperelliptic curves to overcome the non-commutativity of the GCD and the modulo-reduction operations between polynomials. Furthermore, it is essential to investigate the lifting problem

further to ensure that the security of modern cryptosystems based on the difficulty of the ECDLP will not be threatened by the algorithm proposed in section 8.4.

Part IV

Appendices and index

A Pseudocodes

In this chapter, we show all the pseudocodes for the algorithms used throughout this thesis.

A.1 ECDLP using the Log function

In this section, we present the pseudocodes used to compute the Exp function (see algorithm 3), the Log function (see algorithm 4), and the elliptic curve discrete logarithm over $\mathbb{Z}/p^k\mathbb{Z}$ (see algorithm 5). We address the reader to appendix B.1 for the codes used for our implementation.

Specifically, in algorithm 2, we compute the s-th coefficient of the series expansion of $\wp(z)$ as in equation (3.7). Then, in algorithm 3, we compute the point $\left[z^3 : z^3\wp(z) : \frac{z^3}{2}\wp'(z)\right] \in \mathrm{Im}(\mathrm{Exp}_{\overline{\mathcal{W}}})$ for an elliptic curve in short Weierstrass form $\overline{\mathcal{W}}$ over $\mathbb{Z}/p^k\mathbb{Z}$, where $p \mid z$. In particular, we compute the point $\left[z^3 : z^3\wp(z) : \frac{z^3}{2}\wp'(z)\right]$ by determining the p-adic evaluation of the terms of the series expansion of $\wp'(z)$ multiplied by z^3, since its convergence radius greater than $\wp(z)$, as recalled in equation (7.3).

In algorithm 4, we compute the inverse of the map Exp by using the reduction modulo a small power of p by using the series expansions of $\wp(z)$ and $\wp'(z)$ modulo p^5. Specifically, we have that

$$\begin{aligned}
\wp(z) &\equiv \frac{1}{z^2} + \frac{g_2}{20}z^2 + \frac{g_3}{28}z^4 \pmod{p^5},\\
\frac{1}{2}\wp'(z) &\equiv \frac{-1}{z^3} + \frac{g_2}{20}z + \frac{g_3}{14}z^3 \pmod{p^5}.
\end{aligned}$$

Since $[Z : X : Y] = P \in \mathrm{Im}(\mathrm{Exp}_{\overline{\mathcal{W}}})$ is such that $\frac{X}{Z} \equiv \wp(z)$, and $\frac{Y}{Z} \equiv \frac{1}{2}\wp'(z)$, by multiplying both sides of the above equations by z^3, we get

$$\begin{aligned}
z^3\frac{X}{Z} &\equiv z^3\wp(z) \equiv z \pmod{p^5},\\
z^3\frac{Y}{Z} &\equiv \frac{z^3}{2}\wp'(z) \equiv -1 + \frac{g_2}{20}z^4 \pmod{p^5}.
\end{aligned}$$

Thus, the first equivalence implies that $z^2 \equiv \frac{Z}{X} \pmod{p^5}$, while the second equivalence yields

$$z^3\frac{Y}{Z} \equiv z\frac{Z}{X}\frac{Y}{Z} \equiv -1 + \frac{g_2}{20}z^4 \equiv -1 + \frac{g_2}{20}\left(\frac{Z}{X}\right)^2 \pmod{p^5},$$

which implies

$$z \equiv \frac{X}{Y}\left(-1 + \frac{g_2}{20}\left(\frac{Z}{X}\right)^2\right) \pmod{p^5}.$$

Remark A.1. *Since $p \mid z$, the solution we obtain is modulo $p^{5-\nu_p(z)}$, where $\nu_p(z)$ is the p-adic valuation of z.*

Furthermore, if $Z = 0$ or $k \leq 4$, then the logarithm of P is $\frac{X}{Y} \pmod{p^k}$. Since we have a solution modulo p^5, we can find the real z as $z + h \cdot p^5$ for some $h \in \mathbb{N}$. We can then compute the value of z modulo p^6, p^7, and so on, until we determine the logarithm of the point $P \in \operatorname{Im}(\operatorname{Exp}_{\overline{\mathcal{W}}})$ modulo p^k. In order to determine this reduction, we computed the series expansion of $\wp(z)$ modulo p^i for $i = 6, \ldots, k$, since $\frac{X}{Z} \equiv \wp(z) \pmod{p^i}$.

Finally, in algorithm 5, we compute the discrete logarithm for a point $Q \in \overline{\mathcal{W}}(\mathbb{Q})$ such that $Q = h \otimes P$, where $P \in \overline{\mathcal{W}}(\mathbb{Q})$. In order to determine the discrete logarithm of Q, we use the approximation over $\overline{\mathcal{W}}\big(\mathbb{Z}/p^k\mathbb{Z}\big)$. Specifically, we compute the reduction of h modulo the $t = \operatorname{ord}(\operatorname{Mod}_p(P))$ of $\operatorname{Mod}_p(P) - \Omega \in \mathcal{J}_{\mathbb{Z}/p\mathbb{Z}}\big(\overline{\mathcal{W}}\big)$. Since $Q = h \otimes P$, the points $R = Q \ominus (a \otimes P)$ and $S = t \otimes P$ belong to $\operatorname{Im}(\operatorname{Exp}_{\overline{\mathcal{W}}})$, where $h \equiv a \pmod t$. As proven in subsec. 8.4.1, by computing the logarithm of the points R and S, we can determine the discrete logarithm of Q.

Algorithm 2: Get-Exp-Coefficient

```
Function Get-Exp-Coefficient(c, g2, g3, s) { // this function computes the coefficient of the
 s-th term of the series expansion of Exp_W(z)
{
  Data: c is the list over (Z/p^kZ) of the previous coefficients of the series expansion, g2 and g3 are
        the elliptic invariants for the function, and s ∈ N
  Result: the coefficient c of the s-th term of the series expansion
  if(s < 2)
  {
    return 0
  }
  elif(s = 2)
  {
    return g2/20
  }
  elif(s = 3)
  {
    return g3/28
  }
  else
  {
    c ⟵ 0                                                    // initialize coefficient
    for(i = 2 to s − 2)do
    {
      if(i ≤ s − i)
      {
        if(i ≠ s − i)
        {
          c ⟵ c + 2 · c_i · c_{s−i}
        }
        else
        {
          c ⟵ c + c_i · c_{s−i}
        }
      }
    }
    c ⟵ c · 3/((2s+1)(s−3))
    return c
  }
}
```

Algorithm 3: Exp

```
Function Exp(W̄, p, k, z) { // this function computes the series expansion of Exp_W̄(z) for an
  elliptic curve W̄ in short Weierstrass form over Z/p^k Z
{
  Data: W̄ is an elliptic curve in short Weierstrass form given by the equation y^2 = x^3 + ax + b, p is
        an odd prime number, k ∈ N, and z ∈ N such that p | z
  Result: Exp_W̄(z) = [z^3 : z^3 ℘(z) : (z^3/2) ℘'(z)]

  (a, b) ⟵ Get-Params(W̄)                              // get the parameters of the curve W̄
  // compute the elliptic invariants g2 and g3 for the elliptic curve given by the equation
     ((1/2)℘'(z))^2 = ℘^3(z) − (g2/4)℘(z) − g3/4

  g2 ⟵ −4a
  g3 ⟵ −4b
  s ⟵ 2                                      // initial power of z in the series expansion
  ℘_z ⟵ 0                                             // initialize the value of z^3 · ℘(z)
  ℘'_z ⟵ −2                                           // initialize the value of z^3 · ℘'(z)
  c ⟵ (0, 0)     // initialize vector of coeffients c = (c0, c1, c2, ...) of the series expansion
  // initialize the p-adic evaluation for the s-th term of the series expansion of z^3 · ℘'(z)
  e ⟵ 0
  e1 ⟵ max {1, P-Adic-Evaluation(z, p)}                // compute the p-adic evaluation of z
  while( e < k )do
  {
    // compute the s-th coefficient of the series expansion and append it to the vector c

    c ⟵ Concat(c, Get-Exp-Coefficient(c, g2, g3, s))
    // compute the p-adic evaluation of the s-th element of c
    r ⟵ P-Adic-Evaluation(c_s, p)
    // compute the exponent of the z for the s-th term of the series expansion for z^3 · ℘'(z)
    e2 ⟵ 2s − 2
    // compute the p-adic evaluation of the s-th term of the series expansion for z^3 · ℘'(z)
    e ⟵ e1 · ((e2 − 1) + 3) + r
    if( e < k )
    {
      // z^e ≢ 0 (mod p^k)
      ℘_z ⟵ ℘_z + c_s · z^3 · z^e2
      ℘'_z ⟵ ℘'_z + e2 · c_s · z^3 · z^(e2−1)
    }
    s ⟵ s + 1
  }
  return [z^3 : ℘_z : (1/2)℘'_z]
}
```

Algorithm 4: Log

```
Function Log(W̄, p, k, P) { // this function computes the inverse of Exp_W̄(z) for an
 elliptic curve W̄ in short Weierstrass form over Z/p^kZ
{
  Data: W̄ is an elliptic curve in short Weierstrass form given by the equation y^2 = x^3 + ax + b, p is
        an odd prime number, k ∈ N, and P ∈ Exp_W̄(z) for some suitable z
  Result: z ∈ N such that P = Exp_W̄(z)
  (a, b) ⟵ Get-Params(W̄)                         // get the parameters of the curve W̄
  // compute the elliptic invariants g2 and g3 for the elliptic curve given by the equation
     ((1/2)℘'(z))^2 = ℘^3(z) − (g2/4)℘(z) − g3/4

  g2 ⟵ −4a
  g3 ⟵ −4b
  (X, Y, Z) ⟵ Get-Projective-Coordinates(P)
  if( k ≤ 4 or Z = 0 )
  {
    return −X/Y (mod p^k)
  }
  z ⟵ X/Y · (−1 + g2/20 · (Z/X)^2) (mod p^5)
  if( k = 5 )
  {
    return z
  }
  C ⟵ {z}            // initialize the set of elements which are equivalent to z modulo p^k
  for( i = 6 to k )do
  {
    C_curr ⟵ ∅
    foreach( z ∈ C )do
    {
      for( j = 0 to p − 1 )do
      {
        z_curr ⟵ z + j · p^(i−1)
        if( ℘(z_curr) ≡ X/Z (mod p^i) )
        {
          if( P ≡ Exp(z_curr) )
          {
            return z_curr
          }
          else
          {
            C_curr ⟵ C_curr ∪ {z_curr}
          }
        }
      }
    }
    C ⟵ C_curr
  }
  foreach( z ∈ C )do
  {
    if( z < p^k and P ≡ Exp(z) )
    {
      return z
    }
  }
  return (Error: it was not possible to compute z)
}
```

Algorithm 5: ECDLP

Function ECDLP($\overline{\mathcal{W}}$, p, k, P, Q, o) **{**

{

Data: $\overline{\mathcal{W}}$ is an elliptic curve in short Weierstrass form given by the equation $y^2 = x^3 + ax + b$, p is an odd prime number, $\mathsf{k} \in \mathbb{N}$, $P, Q \in \mathcal{J}_{\mathbb{Q}}\left(\overline{\mathcal{W}}\right)$, and o is the cardinality of $\mathcal{J}_{\mathbb{Z}/\mathsf{p}\mathbb{Z}}\left(\overline{\mathcal{W}}\right)$

Result: $\mathsf{h} \in \mathbb{N}$ such that $P = hQ$

if($P \equiv Q$)
{
  return 1
}
// compute the integer h reduced modulo $\mathcal{O}\left(\overline{P}\right)$

$a \longleftarrow 0$
$\overline{P} \longleftarrow \mathrm{Mod}_{\overline{\mathcal{W}}}(P)$
$\overline{Q} \longleftarrow \mathrm{Mod}_{\overline{\mathcal{W}}}(Q)$
for($i = 1$ **to** o)**do**
{
  if($\overline{Q} = i \otimes \overline{P}$)
  {
    $a \longleftarrow i$
  }
}
$i \longleftarrow 2$
$h \longleftarrow a$
$R \longleftarrow Q \ominus (a \otimes P)$ // R is a point belonging to $\mathrm{Im}\left(\mathrm{Exp}_{\overline{\mathcal{W}}}\right)$
$S \longleftarrow o \otimes P$ // S is a point belonging to $\mathrm{Im}\left(\mathrm{Exp}_{\overline{\mathcal{W}}}\right)$
while($h \otimes P \neq Q$)**do**
{
  $S_{\mathsf{mod}} \longleftarrow \mathrm{Mod}_{\mathsf{p}^i}(S)$ // reduce S modulo p^i
  $R_{\mathsf{mod}} \longleftarrow \mathrm{Mod}_{\mathsf{p}^i}(R)$ // reduce R modulo p^i
  $l_1 \longleftarrow \frac{\mathtt{Log}(\overline{\mathcal{W}},\ \mathsf{p},\ i,\ S_{\mathsf{mod}})}{p}$ // compute the logarithm of S modulo p^i
  $l_2 \longleftarrow \frac{\mathtt{Log}(\overline{\mathcal{W}},\ \mathsf{p},\ i,\ R_{\mathsf{mod}})}{p}$ // compute the logarithm of R modulo p^i
  if($l_1 \neq 0$)
  {
    $m \longleftarrow i - 1$
    if($\frac{l_2}{l_1}$ *is a unit of* $\mathbb{Z}/p^m\mathbb{Z}$)
    {
      $r \longleftarrow \frac{l_2}{l_1} \pmod{\mathsf{p}^m}$
      $h \longleftarrow a + r \cdot o$
    }
  }
  $i \longleftarrow i + 1$
}
return h
}

A.2 AG Goppa codes for Edwards curves

In this section, we present the pseudocode for the algorithm used to compute a basis of the Riemann-Roch space $\mathcal{L}(D)$ for an Edwards curve $\mathcal{E}$ (see algorithm 6), where $P + t \cdot O = D \in \mathcal{J}^0_{\mathbb{Z}/p\mathbb{Z}}(\mathcal{E})$, with $t \in \mathbb{N}$, and $O = (0, 1)$. Additionally, we present the pseudocode for the algorithm used to compute a generator matrix for an algebraic-geometric Goppa code for Edwards curves (see algorithm 7). We address the reader to appendix B.2 for the codes used for our implementation.

Algorithm 6: Edwards-Riemann-Roch-Basis

Function `Edwards-Riemann-Roch-Basis(`P, t`)` **{**

{

Data: P is a point belonging to an Edwards curve given by the equation $x^2 + y^2 = 1 + dx^2y^2$ over $\mathbb{Z}/p\mathbb{Z}$, and $t+1$ is the number of functions to compute

Result: the set of functions $\mathbf{F}$ defining a basis for $\mathcal{L}\,(P + t \cdot O)$

$(a, b) \longleftarrow$ `Get-Projective-Coordinates(`P`)`

$\mathbf{F_0} \longleftarrow \frac{X}{X}$

$\mathbf{F_2} \longleftarrow \frac{Z}{Y-Z}$

$\mathbf{F_3} \longleftarrow \frac{Y+Z}{X} \cdot \mathbf{F_2}$

if($P = O = (0, 1)$)

{

$\mathbf{F} \longleftarrow (\mathbf{F_0}, \mathbf{F_2}, \mathbf{F_3})$

$t \longleftarrow t + 1$

}

elif($P = O' = (0, -1)$)

{

$\mathbf{F_1} \longleftarrow \frac{Z}{X}$

$\mathbf{F} \longleftarrow (\mathbf{F_0}, \mathbf{F_1}, \mathbf{F_2}, \mathbf{F_3})$

}

elif($P = H = (1, 0)$)

{

$\mathbf{F_1} \longleftarrow \frac{(X+Z)(Y+Z)}{XY}$

$\mathbf{F} \longleftarrow (\mathbf{F_0}, \mathbf{F_1}, \mathbf{F_2}, \mathbf{F_3})$

}

elif($P = H' = (-1, 0)$)

{

$\mathbf{F_1} \longleftarrow \frac{(X-Z)(Y+Z)}{XY}$

$\mathbf{F} \longleftarrow (\mathbf{F_0}, \mathbf{F_1}, \mathbf{F_2}, \mathbf{F_3})$

}

else

{

$\mathbf{F_1} \longleftarrow \frac{X \cdot (Y+bZ)}{(X-aZ)(Y-Z)}$

$\mathbf{F} \longleftarrow (\mathbf{F_0}, \mathbf{F_1}, \mathbf{F_2}, \mathbf{F_3})$

}

while(`Length(`$\mathbf{F}$`)` $< t$)**do**

{

$i \longleftarrow$ `Length(`$\mathbf{F}$`)`

$h \longleftarrow \left\lfloor \frac{i}{2} \right\rfloor$

if($i \equiv 0 \pmod 2$) // $i = 2h$

{

$\mathbf{F_i} \longleftarrow \mathbf{F_2} \cdot \mathbf{F_{2h-2}}$

}

else // $i = 2h + 1$

{

$\mathbf{F_i} \longleftarrow \frac{Y+Z}{X} \cdot \mathbf{F_{2h}}$

}

$\mathbf{F} \longleftarrow$ `Concat(`$\mathbf{F}$, $\mathbf{F_i}$`)`

}

return F

}

Algorithm 7: Goppa-Edwards-Generator-Matrix

```
Function Goppa-Edwards-Generator-Matrix(P, F, p) {
{
  Data: P = {P_1, ..., P_m} is a set of m points belonging to an Edwards curve given by the equation
        x^2 + y^2 = 1 + dx^2y^2 over Z/pZ such that P_i ∉ supp(F_j) for any F_j ∈ F, and
        F = {F_1, ..., F_n} is a basis for a Riemann-Roch space L(Q + n·O), where Q, O ∈ E(Z/pZ),
        and O = (0, 1)
  Result: the generator matrix G for an algebraic-geometric Goppa code for E
  G ⟵ 0 ∈ Z/pZ^{n×m}
  for( i = 1 to n )do
  {
    for( j = 1 to m )do
    {
      G_{i,j} ⟵ F_i(P_j)                                        // evaluate F_i at P_j
    }
  }
  // reduce G to its standard form, i.e.  G = [I_n | M]
  G ⟵ Echelon-Form(G)
  return G
}
```

A.3 AG Goppa codes for hyperelliptic curves

In this section, we present the pseudocode for the algorithm used to compute a basis of the Riemann-Roch space $\mathcal{L}(D)$ for a hyperelliptic curve $\mathcal{H}$, where $D = \Delta + n \cdot \Omega$, with $\Delta = (u(x), v(x))$ be the Mumford representation of a divisor in $\mathcal{J}_{\mathbb{Z}/p\mathbb{Z}}(\mathcal{H})$ for a randomly chosen pair of suitable polynomials $(u(x), v(x))$, and $n \in \mathbb{N}$. We also use the same pseudocode to compute a generator matrix for an algebraic-geometric Goppa MDS-code for hyperelliptic curves (see algorithm 8 and algorithm 9). The code used for our implementation can be found in appendix B.3.

Specifically, in algorithm 8, we obtain a basis for $\mathcal{L}(D)$ using a randomly chosen pair of suitable polynomials $(u(x), v(x))$, as described in section 12.1. On the other hand, in algorithm 9, we obtain a list of m randomly chosen linearly independent points P_i to compute a generator matrix for an AG Goppa MDS-code. Specifically, we choose a random divisor $R = \alpha_1 \cdot P_1 + \ldots + \alpha_m \cdot P_m$, where $P_i \in \mathcal{H}(\mathbb{Z}/p\mathbb{Z})$, subject to the following conditions: for all i, $\alpha_i = 1$, P_i does not belong to the support of the functions in the basis of $\mathcal{L}(D)$, and the polynomial $u(x)$ does not vanish at $x = x_{P_i}$. As the algorithm is random, it may choose a set of linearly dependent points. Therefore, we check whether the code generated by the basis $\mathbf{F}$ and the set of points belonging to $\text{supp}(R)$ is MDS. If this test fails for MAX-TRIES times, we discard the points in $\text{supp}(R)$ and repeat the process to obtain another set of m points. Note that if $\text{card}(\mathcal{H}(\mathbb{Z}/p\mathbb{Z}))$ is sufficiently large, the probability of computing an MDS-code is high.

Algorithm 8: Goppa-MDS-HEC-Generator-Matrix - Part 1

Function `Goppa-MDS-HEC-Generator-Matrix(`n, m, s, p`)` **{**

{

Data: $n \in \mathbb{Z}$ is a positive integer defining the number of elements for a basis of the Riemann-Roch space for a hyperelliptic curve of genus at least s over $\mathbb{Z}/p\mathbb{Z}$, and $m \in \mathbb{N}$ is the number of random points used to compute the generator matrix $\mathbf{G} \in \mathbb{Z}/p\mathbb{Z}^{n\times m}$

Result: the generator matrix $\mathbf{G}$ for an algebraic-geometric Goppa code for a hyperelliptic curve $\mathcal{H}$ such that $\Delta \in \mathcal{J}_{\mathbb{Z}/p\mathbb{Z}}(\mathcal{H})$

```
if( s = 0 )
{
    u(x) ⟵ 1
    v(x) ⟵ 0
}
else
{
    l ⟵ Random-Integer(0, s - 1)              // compute a random integer in [0, s − 1]
    u(x) ⟵ 1
    for( i = 1 to s )do
    {
        u(x) ⟵ u(x) · (x − Random-Integer(1, p))
    }
    v(x) ⟵ Random-Polynomial(l)      // compute a random polynomial in Z/pZ[x] of degree l
    while( gcd(u(x), u^2(x), v(x)) ≠ 1 )do
    {
        v(x) ⟵ Random-Polynomial(l)
    }
}
// compute the true genus of the curve
g ⟵ s
while( (2g + 2) − s − ⌊m/2⌋ − (m (mod 2)) < 0 )do
{
    g ⟵ g + 1
}
δ_1 ⟵ 2g + 1
δ_2 ⟵ n + g − 1
h ⟵ ⌊(δ_2 − s)/2⌋
k ⟵ ⌊(δ_2 − (δ_1 − s))/2⌋
Ψ(x, y) ⟵ (y + u(x))/v(x)
// compute the set of functions F defining a basis for L(Δ + n · Ω), where Δ = (u(x), v(x))
F ⟵ ()                                                        // initialize an empty list
for( i = 0 to h )do
{
    F ⟵ Concat(F, x^i)
}
for( j = 0 to k )do
{
    F ⟵ Concat(F, Ψ(x, y) · x^j)
}
// continue on algorithm 9
}
```

Algorithm 9: Goppa-MDS-HEC-Generator-Matrix - Part 2

```
Function Goppa-MDS-HEC-Generator-Matrix(n, m, s, p) {
{
  // see algorithm 8 for the first part of the algorithm
  P ⟵ ()                                                        // empty list
  // initialize e as {0} to not include the abscissa x = 0
  e ⟵ {0}                                  // the set of already used abscissas
  c ⟵ 0                                                  // counter for guesses
  MAX-TRIES ⟵ 10        // maximum number of guesses in finding points to get a MDS-code
  // compute m random points P_i such that, for j = 1,...,n,, P_i ∉ supp(F_j), u(x_{P_i}) ≠ 0, and
     x_{P_i} ∉ e
  while( Length(P) < m )do
  {
    P ⟵ Random-Point(F, u(x), e)
    if( m ≡ 1 (mod 2) and Length(P) + 1 = m )
    {
      R ⟵ Concat(P, (x_P, y_P))                          // append (x_P, y_P) to P
    }
    else
    {
      R ⟵ Concat(P, (x_P, y_P), (x_P, −y_P))   // append (x_P, y_P) and (x_P, −y_P) to P
    }
    // the function Check-MDS check if the code generated by F and R is MDS
    if( Length(P) ≥ n − 2 and Check-MDS(F, R) is false )
    {
      // the set R contains linearly dependent points
      c ⟵ c + 1
    }
    else
    {
      c ⟵ 0
      P ⟵ Copy(R)
      e ⟵ e ∪ {x_P}
    }
    if( c ≥ MAX-TRIES )
    {
      // reset points
      P ⟵ ()
      e ⟵ {0}
    }
  }
  G ⟵ 0 ∈ Z/pZ^{n×m}
  for( i = 1 to n )do
  {
    for( j = 1 to m )do
    {
      G_{i,j} ⟵ F_i(P_j)                                      // evaluate F_i at P_j
    }
  }
  // reduce G to its standard form, i.e.  G = [I_n | M]
  G ⟵ Echelon-Form(G)
  return G
}
```

A.4 HL-codes

In this section, we present the pseudocodes to compute the generator matrix G of a HL-code, as defined in definition 13.6, and for determining all redundancy relations for each a_j. Additionally, we provide the pseudocodes to decode a received word $\mathbf{x} = \mathbf{a} \cdot G + \mathbf{e}$, where $\mathbf{e}$ is an error word such that $\mathrm{wt}(\mathbf{e}) = \left\lfloor \frac{d-1}{2} \right\rfloor$, d is the minimum distance of the HL-code, and $\mathrm{wt}(\cdot)$ is the function that gives in turn the (Hamming) weight of a vector. We address the reader to appendix B.4 for the results of our implementation for HL-code with parameters $m = 10$, and $m = 12$.

First, we defined a function in algorithm 10 that computes

$$\phi(i,k) = i + (-1)^{\psi(i,k)} \cdot 2^{k-1},$$

where ψ and ϕ are the functions defined in equations (13.12) and (13.13), respectively.

Algorithm 10: Change-Bit

```
Function Change-Bit(i, k) { // this function computes φ(i,k)
{
  Data: i ∈ ℕ, k ∈ ℕ
  Result: i ± 2^{k−1}
  (j_{n−1}, j_{n−2}, ..., j_1, j_0)_2 ⟵ i          // get the binary representation of i
  if ( j_{k−1} = 1 ) // j_{k−1} == 1
  {
    return i − 2^{k−1}
  }
  else // j_{k−1} == 0
  {
    return i + 2^{k−1}
  }
}
```

In algorithm 11, we defined a pseudocode that computes the functions λ and ν as defined in equations (13.7) and (13.6), respectively. Specifically, this algorithm returns the tuple $(\lambda(i), \nu(i))$ for a given integer $i \in \{0, \dots, k-1\}$.

Algorithm 11: Row-To-Comb-Index

```
Function Row-To-Comb-Index(i, m) { // this function computes λ(i)
{
  Data: i ∈ {0,...,k−1}, m ∈ ℕ, where i is the row of the generator matrix G ∈ F_2^{k×n}, k = 2^{m−1},
        and n = 2^m
  Result: (i', t) such that e_{i'} ∈ binom(M, t+1) or e_{i'} ∈ Y-set
  t ⟵ −1
  s ⟵ 0
  s' ⟵ 0
  while ( i ≥ s ) {
  {
    s' ⟵ s
    t ⟵ t + 1
    s ⟵ s + binom(m, t)
  }
  // t − 1 is the maximum integer such that i ≥ s' = Σ_{j=0}^{t−1} binom(m, j)
  return (i − s', t − 1)
}
```

In algorithm 12, we defined the pseudocode to compute $f(i)$ as outlined in equation (13.9). In particular, the pseudocode returns the tuple (i), if $1 \leq i \leq m$, the $\lambda(i)$-th element of the set $\binom{\{1,\dots,m\}}{\nu(i)+1}$ (sorted according to the rule defined in equation (13.8)), if $m < i < k - \frac{1}{2}\binom{m}{m/2}$, and the $\lambda(i)$-th element of the complement-free set Y, if $k - \frac{1}{2}\binom{m}{m/2} \leq i < k$.

Algorithm 12: Row-To-Comb

```
Function Row-To-Comb(i, m, Y-set) { // this function computes f(i)
{
  Data: i ∈ {1, ..., k − 1}, m ∈ ℕ, where i is the row of the generator matrix G ∈ 𝔽₂^(k×n), k = 2^(m−1),
        and n = 2^m
  Result: f(i), where f is the funcion in equation (13.9)
  if ( 1 ≤ i ≤ m )
  {
    return (i)
  }
  k′ ⟵ k − ½ (m choose m/2)
  (i′, t) ⟵ Row-To-Comb-Index(i, m)
  if ( m < i < k′ )
  {
    return e_i′ ∈ ({1,...,m} choose t+1)
  }
  else
  {
    return e_i′ ∈ Y-set
  }
}
```

In algorithm 13, we computed the operator $\underset{e}{\Delta}\, x_i$, where $e \in \mathbb{N}^l$, for a coefficient a_j. We used the recursive definition in equation (13.15) for this operator because it is more efficient than the iterative definition in equation (13.16). Specifically, the recursive definition allows for easy implementation of caching, where the previously computed results can be stored and reused to avoid recomputation. For instance, we have that

$$\begin{aligned}\underset{(1,2,3,4)}{\Delta} x_0 &= \underset{(1,2,3)}{\Delta} x_0 + \underset{(1,2,3)}{\Delta} x_8 = \\ &= \left(\underset{(1,2)}{\Delta} x_0 + \underset{(1,2)}{\Delta} x_4\right) + \left(\underset{(1,2)}{\Delta} x_8 + \underset{(1,2)}{\Delta} x_{12}\right) = \\ &= \left(\underset{1}{\Delta} x_0 + \underset{1}{\Delta} x_2 + \underset{1}{\Delta} x_4 + \underset{1}{\Delta} x_6\right) + \left(\underset{1}{\Delta} x_8 + \underset{1}{\Delta} x_{10} + \underset{1}{\Delta} x_{12} + \underset{1}{\Delta} x_{14}\right).\end{aligned}$$

Therefore, starting from a_1 to a_{k-1}, one can cache all the intermediate results of the operator Δ in order to speed up the subsequent computations. For instance, in the above example, by caching the results of $\underset{(1,2,3)}{\Delta} x_0$ during a previous computation, it is possible to compute only a half of the elements in $\underset{(1,2,3,4)}{\Delta} x_0$. In contrast, it is not straightforward to implement caching with the iterative definition of Δ.

Additionally, in order to improve the efficiency of algorithm 14 (explained below), we return $\underset{e}{\Delta}\, x_i$ as a set of indices of the components of x taken into account. For instance, $\underset{(1,2,3,4)}{\Delta} x_0 = \sum_{i=0}^{15} x_i$ will be represented as the set $\{0, \ldots, 15\}$.

In algorithm 14, we computed the set of all redundancy relations of a_j as defined in equation (13.11). As the redundancy relations of a_j form a partition of the n components of the word $\mathbf{x} = \mathbf{a} \cdot G + \mathbf{e}$, we used a set of indices $I_n = \{0, \ldots, n-1\}$ to determine the next component x_i of $\mathbf{x}$ to be considered. For each computed redundancy relation, we remove the components x_j that considered in the last computed relation from I_n. For instance, $\underset{(1,2,3,4)}{\Delta} x_0 = \sum_{i=0}^{15} x_i$, therefore we remove the indices

Algorithm 13: Redundancy-Relation

```
Function Redundancy-Relation(e, i, C) { // compute Δ_e x_i for a_j, with f(j) = e
{
  Data: (j_1, ..., j_t) = e = f(j) for some j, i ∈ {1, ..., n}, C, where n = 2^m, and C is a dictionary used
        to cache the already computed redundancy relations
  Result: the set of indices of the components of x in Δ_e x_i

  if ( t = 0 )
  {
    return (∅, C)
  }
  if ( (e, i) is in C ) // check if the pair (e, i) was already computed
  {
    // return the cached value C(e, i) = C((j_1, ..., j_t), i) and the cache C
    return (C(e, i), C)
  }
  if ( t = 1 ) // if t == 1, then return Δ_{j_1} x_i = x_i + x_{φ(i, j_1)} and C
  {
    i' ⟵ Change-Bit(i, j_1)
    R ⟵ {i, i'}                                        // set of indices i and i'
    return (R, C)
  }
  else // if t > 1, then compute Δ_e x_i by recursion
  {
    e' ⟵ (j_1, ..., j_{t-1})
    i' ⟵ Change-Bit(i, j_t)
    (R_1, C) ⟵ Redundancy-Relation(e', i, C)                    // compute Δ_{e'} x_i
    C(e', i) ⟵ R_1                                  // update cache with Δ_{e'} x_i
    (R_2, C) ⟵ Redundancy-Relation(e', i', C)                   // compute Δ_{e'} x_{i'}
    R ⟵ R_1 ∪ R_2
    C(e', i') ⟵ R_2                                 // update cache with Δ_{e'} x_{i'}
    C(e, i) ⟵ R                                     // update cache with Δ_e x_i
    // return the set of indices of the components of x in Δ_e x_i and the cache C
    return (R, C)
  }
}
```

$\{0, \ldots, 15\}$ from I_n.

Algorithm 14: Redundancy-Relations

```
Function Redundancy-Relations(j, m, Y-set, C) { // compute the set of all redundancy
  relations for a_j
{
  Data: j ∈ {1, ..., k − 1}, m ∈ ℕ, Y-set, C, where k = 2^(m−1), and C is a dictionary used to cache the
        already computed redundancy relations
  Result: {Δ_e x_i} for i = 0, ..., n − 1, e and the updated cache C

  n ⟵ 2^m
  R ⟵ empty list
  I_n ⟵ {0, ..., n − 1}
  e ⟵ Row-To-Comb(j, m, Y-set)
  while ( I_n is not empty ) {
  {
    i ⟵ lowest element in I_n
    (R_c, C) ⟵ Redundancy-Relation(e, i, C)
    append R_c to R
    I_n ⟵ I_n \ R_c                        // remove the set of indices in R_c from I_n
  }
  return (R, e, C)
}
```

In algorithm 15, we computed all the redundancy relations for each a_j, where $j = j_{\text{start}}, \dots, j_{\text{end}}$, and we store them in a dictionary indexed by the coefficient degree of a_j. The reason for this choice is that, by definition, the first $k - \frac{1}{2}\binom{m}{m/2}$ rows of the generator matrix G are fixed, meaning that two different generator matrices will have the same first $k - \frac{1}{2}\binom{m}{m/2}$ rows. Hence, to optimize the process, we compute a_j, for $0 \leq j < k - \frac{1}{2}\binom{m}{m/2}$, and save the cache and the redundancy relations in a file. Since the last $\frac{1}{2}\binom{m}{m/2}$ rows of the generator matrix are random and depend on the complement-free set Y, during the key-generation phase, we can compute the remaining redundancy relations, specifically the redundancy relations of a_j, for $k - \frac{1}{2}\binom{m}{m/2} \leq j < k$. In order to save the results in a file, we run the algorithm with $j_{\text{start}} = 0$ and $j_{\text{end}} = k - \frac{1}{2}\binom{m}{m/2} - 1$, and during the key-generation phase, we run the algorithm with $j_{\text{start}} = k - \frac{1}{2}\binom{m}{m/2}$ and $j_{\text{end}} = k - 1$.

Algorithm 15: Redundancy-Relations-Set

```
Function Redundancy-Relations-Set(m, Y-set, C, j_start, j_end) { // compute the set of all
  redundancy relations for all a_j, for j = j_start, ..., j_end
{
  Data: C is the cache, m ∈ ℕ, Y-set, j_start ∈ ℕ, j_end ∈ ℕ, with 0 ≤ j_start < j_end < k, and Y-set is the
        complement-free set from Complement-Free-Set
  Result: { Δ_e x_i } for each a_j, with i = 0, ..., 2^m − 1, j = j_start, ..., j_end
  R ⟵ empty dictionary
  for (j = j_start to j_end) {
  {
    (R_c, e, C) ⟵ Redundancy-Relations(j, m, Y-set, C)
    r ⟵ Length(e)                                  // r is the degree of the coefficient a_j
    if (R has not key r)
    {
      R(r) ⟵ empty list
    }
    insert (j, R_c) into R(r)                        // insert (j, R_c) into R with key r
  }
  return R
}
```

In algorithm 16, we computed the complement-free set Y. Specifically, we list all the element of $\binom{\{1,\dots,m\}}{m/2}$ and sort them according to the rule defined in equation (13.8). We refer to this list as the X-set. It is clear that the first half of the X-set is the complement of the second half in reverse order. In particular, if A is the first half of the X-set and B is the second half of the X-set in reverse order, then A_i is the complement of B_i, where A_i and B_i are the i-th element of A and B, respectively. Therefore, we can compute the complement-free set Y by randomly choosing an element from A or B. In order to add some randomness, we shuffle the two lists in the same way.

In algorithm 17, we define a pseudocode to compute the basis vector $\mathbf{v_i}$ by using the exponentiation by squaring, as previously noted in remark 13.1. Specifically, if $\mathbf{r} = (\mathbf{0}|\mathbf{1})$, with $\mathbf{0}, \mathbf{1} \in \mathbb{F}_2{}^{2^i}$, then $\mathbf{v_i} = \mathbf{r}^{\frac{n}{2^{i+1}}}$. For instance, if $n = 8$ and $\mathbf{r} = (0|1)$, then $i = 0$ and $\mathbf{v_1} = \mathbf{r}^{\frac{8}{2}} = (0|1)^4 = (0,1,0,1,0,1,0,1)$.

In algorithm 18, we computed the first $k - \frac{1}{2}\binom{m}{m/2}$ rows of the generator matrix G. In particular, we compute the first $m+1$ rows using algorithm 17. We then compute the

Algorithm 16: Complement-Free-Set

```
Function Complement-Free-Set(m) { // build the partial generator matrix a the HL-code of
 parameter m
{
  Data: m ∈ ℕ, where m | 2
  Result: The complement-free Y-set
  Y-set ⟵ empty list
  M ⟵ {0, ..., m − 1}
  // compute all m/2-combination of elements in M
  X-set ⟵ sorted list of e ∈ (M choose m/2)
  // if we take all e in (M choose m/2) such that e_1 ≤ e_2, then the second half of X-set is the
     complement of its first half
  t ⟵ (m choose m/4)                                        // half of the size of X-set
  A ⟵ first t elements of X-set
  B ⟵ last t elements of X-set
  B ⟵ reverse B                      // B = (e_{t+1}, ..., e_{2t}) ↦ (e_{2t}, ..., e_{t+1})
  A ⟵ shuffle A
  // shuffle B such that B_j is always the complement of A_j
  B ⟵ shuffle B as A
  for (j = 1 to t) {
  {
    r ⟵ random binary number                                  // r ∈ {0, 1} random
    if (r = 0)
    {
      append A_j to Y-set                 // append the j-th element of A to Y-set
    }
    else // r == 1
    {
      append B_j to Y-set                 // append the j-th element of B to Y-set
    }
  }
  return Y-set
}
```

Algorithm 17: Build-V-Vector

```
Function Build-V-Vector(i, n) {
{
  Data: i ∈ ℕ, n ∈ ℕ, with n | 2^{i+1}
  Result: (0|1|...|0|1) = v ∈ F_2^n, where 0, 1 ∈ F_2^{2^i}
  s ⟵ n / 2^{i+1}
  v' ⟵ empty vector
  z ⟵ 0 ∈ F_2^{2^i}
  u ⟵ 1 ∈ F_2^{2^i}
  v ⟵ (z|u)
  while (s > 1) {
  {
    if (s is odd)
    {
      v' ⟵ (v'|v)                                      // concatenate v' and v
      s ⟵ s − 1
    }
    v ⟵ (v|v)                                      // concatenate v with itself
    s ⟵ s/2
  }
  v ⟵ (v|v')                                          // v = (0|1)^{n/2^{i+1}}
  return v
}
```

remaining $k - \frac{1}{2}\binom{m}{m/2} - (m+1)$ rows by computing all the pairwise products of the vectors $\mathbf{v_i}$, for $i = 1, \ldots, m$, taking all the combinations of indices $\binom{\{1,\ldots,m\}}{j}$ (sorted according to the rule defined in equation (13.8)), for $j = 2, \ldots, \frac{m}{2} - 1$. As these rows do not change, we save this matrix into a file. During the key generation phase, we will read the matrix from the saved file to complete the remaining rows of the

generator matrix.

Algorithm 18: Build-HL-Partial-Generator-Matrix

```
Function Build-HL-Partial-Gen-Matrix(m) { // build the partial generator matrix a the
  HL-code of parameter m
{
  Data: m ∈ ℕ, where m | 2
  Result: The partial generator matrix G ∈ F_2^{k×n} of an HL-code, where n = 2^m, and k = 2^{m−1}
  n ⟵ 2^m
  k ⟵ 2^{m−1}
  G ⟵ empty binary matrix of dimension k × n
  G_0 ⟵ 1 ∈ F_2^n
  for (i = 1 to m) {
  {
    G_i ⟵ Build-V-Vector(i − 1, n)          // set the i-th row of G as (0|1)^{n/2^i}
  }
  t ⟵ m + 1
  M ⟵ {0, . . . , m − 1}
  // compute all the pairwise products of the rows at j of G, with j = 1, . . . , m
  for (i = 2 to m/2 − 1) {
  {
    // get all the ordered combinations in (M choose i)
    forall (e in (M choose i))do
    {
      G_t ⟵ ∏_{j∈e} G_j                      // pairwise products of rows
      t ⟵ t + 1;
    }
  }
  return G
}
```

In algorithm 19, we computed the complete generator matrix of the HL-code by using the complement-free set Y computed in algorithm 16, and the partial generator matrix that was computed using the algorithm 18.

Algorithm 19: Build-HL-Generator-Matrix

```
Function Build-HL-Gen-Matrix(m, G, Y-set) { // build the full generator matrix a the HL-code
  of parameter m
{
  Data: m ∈ ℕ, G ∈ F_2^{k×n}, where m | 2, n = 2^m, k = 2^{m−1}, G is the partial generator matrix from
        Build-HL-Partial-Gen-Matrix, and Y-set is the complement-free set from
        Complement-Free-Set
  Result: The full generator matrix G ∈ F_2^{k×n} of an HL-code
  // compute the index of the first row of G to insert the pairwise product from Y-set
  i ⟵ Σ_{j=0}^{m/2−1} (m choose j) + 1
  forall (e in Y-set)do
  {
    G_i ⟵ ∏_{j∈e} G_j                        // pairwise products of rows
    i ⟵ i + 1;
  }
  return G
}
```

In order to decode the word $\mathbf{x} = \mathbf{a} \cdot G + \mathbf{e}$, we need to apply a multilevel decoding starting with the coefficient a_j of highest-degree. In algorithm 20, we decoded the word $\mathbf{x}$ for the coefficient of degree r, which was computed using algorithm 15. We

also set the appropriate a_j to the vector $\mathbf{a}'$, which, in the end of algorithm 15, will be equal to the weighted vector $\mathbf{a}$ in $\mathbf{x}$ one is looking for.

Note that, it is not necessary to count exactly how many redundancy relations for a_j have outcome equal to 1. Specifically, let s be the number of redundancy relations for a_j, if the number of ones (or the number of zeros) is greater than $\frac{s}{2}$, then the value of a_j is equal 1 or 0, respectively.

Algorithm 20: Decode-Level-R

```
Function Decode-Level-R(R, r, x, G) {
{
  Data: R, x ∈ F2^n, G ∈ F2^(k×n), where x is the vector to decode, G is the generator matrix of the
        HL-code, and R is the dictionary of the redundancy relations for the coefficients a_j of degree r
        as in Redundancy-Relations-Set
  Result: a′ and v
  k ⟵ get the number of rows of G
  n ⟵ get the number of columns of G
  a′ ⟵ 0 ∈ F2^k
  v ⟵ 0 ∈ F2^n
  forall ( (j, R_j) in R(r) )do
  {
    t ⟵ 0
    u ⟵ 0                                                    // counter for the 1
    z ⟵ 0                                                    // counter for the 0
    s ⟵ Length(R_j)                       // number of rendundancy relations for a_j
    // count how many e ∈ R_j are equal to 1, and how many are equal to 0
    while ( u ≤ s/2 and z ≤ s/2 ) {
    {
      // h is sum the components of x according to the t-th redundancy relation R_j^(t) of a_j
      h ⟵ Σ_{e∈R_j^(t)} Σ_{i∈e} x_i
      if ( h = 1 )
      {
        u ⟵ u + 1
      }
      else
      {
        z ⟵ z + 1
      }
      t ⟵ t + 1
    }
    if ( u = z )
    {
      // it is not possible to determine the proper value of a_j
      return Error!
    }
    elif ( u > z ) // a_j = 1
    {
      v ⟵ v + G_j                                                        // v_j = G_j
      a′_j ⟵ 1                                     // set the j-th element of a′ as 1
    }
    else
    {
      // a_j = 0
    }
  }
  return v, a′
}
```

Finally, in algorithm 21, we fully decode the word $\mathbf{x} = \mathbf{a} \cdot G + \mathbf{e}$ by calling the algorithm 20 starting with the highest-degree coefficients. Specifically, we first reverse sort the keys of the dictionary of redundancy relations computed in algorithm 15. Since the keys of the dictionary are the degrees of the coefficients, the first element in

the list L is the highest degree, and the last element of L is the lowest degree. Next, we call algorithm 20 for each degree in L, subtracting the resulting word from the previous one. In particular, we call algorithm 20 with the word $\mathbf{v}$, which is

- $\mathbf{x}$ before the first call of algorithm 20;
- $\mathbf{x} - \mathbf{x}'$ after the first call of algorithm 20, where $\mathbf{x}'$ is the result of algorithm 20, that is, $\mathbf{x}' = \sum a_i \mathbf{v_i}$ for each coefficient a_i of highest degree.

After the loop, the vector $\mathbf{v}$ will be equal to $a_0\mathbf{v_0} + \mathbf{e}$. In order to decode a_0, we count the number of bits in $\mathbf{v}$ that are equal to 1. If the number of ones is greater then $\frac{n}{2}$, then a_0 is equal to 1, while if the number of ones is lower than $\frac{n}{2}$, then a_0 is equal to 0.

Algorithm 21: Decode

```
Function Decode(x, G, R) {
{
  Data: a·G + e = x ∈ F2^n, G ∈ F2^(k×n), R, where x is the vector to decode, G is the generator matrix of
        the HL-code, and R is the dictionary of the redundancy relations for the coefficients a_j of
        degree r as in Redundancy-Relations-Set
  Result: a·G
  k ⟵ get the number of rows of G
  n ⟵ get the number of columns of G
  a ⟵ 0 ∈ F2^k
  v ⟵ x
  L ⟵ reverse sort the keys in R
  forall ( r in L )do
  {
    (v', a') ⟵ Decode-Level-R(R, r, v, G)
    v ⟵ v − v'                                              // v − ∑ a_j v_j
    a ⟵ a + a'
  }
  // v = a_0 v_0 + e
  u ⟵ count how many components of v are equal to 1
  if ( u = n/2 )
  {
    // it is not possible to determine the proper value of a_0
    return Error!
  }
  elif ( u > n/2 ) // a_0 = 1
  {
    v ⟵ v − G_0                                             // v_0 = G_0
    a_0 ⟵ 1                            // set the first element of a as 1
  }
  else
  {
    // a_0 = 0
  }
  // now v = e and x − v = a·G
  return x − v, a
}
```

In summary, in order to optimize the overall process, we have divided our algorithm into two steps. First, we compute the partial generator matrix using the algorithm 18, and we save it to a file. Additionally, we calculate the redundancy relations for a_j, with $1 \leq j < k - \frac{1}{2}\binom{m}{m/2}$, and save them along with the cache to a file. Once the partial generator matrix and redundancy relations have been saved, we use the McEliece protocol to encrypt/decrypt a message. Specifically, we define protocol 1, protocol 2, and protocol 3.

Protocol 1: Key-Gen

Input: The parameter $m \in \mathbb{N}$ of the HL-code

Result: Return the private-key and the public-key

1. Retrieve the partial generator matrix for m from the file, and get its dimension $k \times n$;
2. Compute a random complement-free set Y using algorithm 16, and determine the last rows of the generator matrix G using algorithm 19;
3. Retrieve the redundancy relations for a HL-code of parameter m from the file, and compute the remaining redundancy relations using algorithm 15 with $j_{\text{start}} = k - \frac{1}{2}\binom{m}{m/2}$, and $j_{\text{end}} = k - 1$. We refer to R as the complete dictionary of all the redundancy relations;
4. Compute an invertible matrix $S \in {\mathbb{F}_2}^{k \times k}$, and its inverse S^{-1};
5. Compute a permutation $\rho \in Sym(\{1, \dots, n\})$, and its inverse ρ^{-1};
6. Return the private-key $(S^{-1}, \rho^{-1}, G, R)$ and the public-key $G' = \rho(S \cdot G)$, which consists of the ρ-permutation of the columns of $S \cdot G$.

Protocol 2: Encrypt

Input: The public-key $G' \in {\mathbb{F}_2}^{k \times n}$, the parameter m of the code, and a message $\mathbf{h} \in {\mathbb{F}_2}^k$

Result: Return the encrypted message

1. Compute the minimum distance $d = 2^{\frac{m}{2}}$, and $t = \left\lfloor \frac{d-1}{2} \right\rfloor$;
2. Compute a random word $\mathbf{e} \in {\mathbb{F}_2}^n$ such that $\text{wt}(\mathbf{e}) = t$;
3. Return the encrypted message $\mathbf{c} = \mathbf{h} \cdot G' + \mathbf{e}$.

Protocol 3: Decrypt

Input: The private-key $(S^{-1}, \rho^{-1}, G, R)$, the parameter m of the code, and the encrypted message $\mathbf{h} \cdot G' + \mathbf{e} = \mathbf{c} \in {\mathbb{F}_2}^n$

Result: Retrieve the message m from the encrypted message c

1. Decode the word $\rho^{-1}(\mathbf{c})$ using algorithm 21, and get the word $\mathbf{a} = \mathbf{h} \cdot S$;
2. Return the message $\mathbf{h} = \mathbf{a} \cdot S^{-1}$.

B Implementations and benchmarks

In this appendix, we show the implementation codes and the results of the algorithm described in appendix A using Python V3.6, and the SageMath library V9.0. Furthermore, we benchmarked our algorithms using the function **perf_counter** (which measures the WALL-time) of the library **time** of Python, and plotted the results using the libraries **plotly** and **matplotlib** of Python.

Lastly, the hardware specifications of the machine used for these implementations are provided in table B.1.

TABLE B.1: Hardware

SSD	SAMSUNG 850 EVO - 256 GB
RAM	12 GB - 1600 MHz
Processor	i5-5200U 2.2 GHz

B.1 ECDLP using the Log function

In this section, we show the results of our implementation of the pseudocodes in appendix A.1. Specifically, in script B.1, we show our implementation codes using the library SageMath, and the library **gmpy2** (a Python's wrapper of the library GMP) to implement the computation over $\mathbb{Q}$. Note that, in our implementation, although in (7.3) we defined the p-adic evaluation differently, we forced the p-adic evaluation of 0 to be equal to zero in order to easy handle the case $\mathrm{Exp}(0)$.

In fig. B.1 and fig. B.2, we plot the performance of the ECDLP algorithm discussed in subsec. 8.4.1. Specifically, we computed the discrete logarithm for the elliptic curve in short Weierstrass form given by the equation $y^2 = x^3 - x + \frac{1}{4}$ over $\mathbb{Z}/p^k\mathbb{Z}$ for $p = 3$, $p = 5$, $p = 17$, and $p = 37$. It is noteworthy that, for $p = 37$, there is a significant decrease in performance due to the high computational cost of the operation $n \otimes P$ over $\mathbb{Q}$ (as outlined in section 8.2). Nonetheless, as n increases (denoted as "logarithm" in the x-axis), there is an exponential slowdown, but for $p = 3$, this growth is much smaller since the order of the Jacobian of this curve is small.

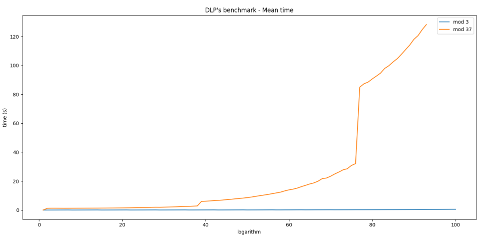

FIGURE B.1: Benchmark for $p = 3$ and $p = 37$

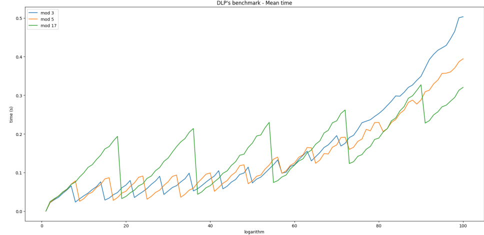

FIGURE B.2: Benchmark for $p = 3$, $p = 5$ and $p = 17$

FIGURE B.3: Benchmark of the ECDLP for the elliptic curve given by the equation $y^2 = x^3 - x + \frac{1}{4}$ over $\mathbb{Z}/p^k\mathbb{Z}$

```
import gmpy2


# this function computes the p-adic evaluation of an integer
def p_adic_eval(a, p):
    # a -> an integer
    # p -> a prime number
    if a == 0:
        return 0 # if a = 0, then return 0
    i = 0
    while a % (p ** (i + 1)) == 0: # if p^(i + 1) divides a, then increment i
        i += 1
    return i


# this function computes the p-adic evaluation of a rational number
def p_adic_eval_fraction(a, p):
    # a -> a rational number
    # p -> a prime number
    return p_adic_eval(a.numerator, p) - p_adic_eval(a.denominator, p)


# this function transforms an integer "a" in (+a) or (-a)
def mod_transform(a, q):
    # a -> an integer
    # q -> an integer
    if a <= q // 2: # if a <= q / 2
        return a
    else: # if a > q / 2
        return - (q - a)


# this function transforms a point P, whose projective coordinates are over the rationals,
# into an equivalent point Q reduced modulo p^k
def reduce_point(P, p, k):
    # P -> point belonging to an elliptic curve W
    # p -> an odd prime number
    # k -> a positive integer

    q = p ** k
    # g = GCD(Z_P, X_P, Y_P)
    g = gmpy2.gcd(gmpy2.gcd(P[0].numerator, P[1].numerator), P[2].numerator)
    (Z, X, Y) = (P[0] / g, P[1] / g, P[2] / g)
    z = (Z.numerator * gmpy2.invert(Z.denominator, q)) % q # z = Z (mod p^k)
    x = (X.numerator * gmpy2.invert(X.denominator, q)) % q # x = X (mod p^k)
    y = (Y.numerator * gmpy2.invert(Y.denominator, q)) % q # y = Y (mod p^k)
    return (z, x, mod_transform(y, q))



# this function computes the s-th coefficient of the series expansion of wp(z)
def get_exp_coefficient(g2, g3, s, coefficients):
    # s -> integer
    # g2, g3 -> elliptic invariants defining an elliptic curve in short Weierstrass form
    # given by the equation (1/2 wp'(z))^2 = wp(z)^3 - g2 * wp(z) - g3
    # coefficients -> list of the already computed coefficients of the series expansion

    if s < 2:
        return gmpy2.mpq(0)
    elif s == 2:
        return gmpy2.mpq(g2, 20) # g2 / 20
    elif s == 3:
        return gmpy2.mpq(g3, 28) # g3 / 28
```

```
64      c = gmpy2.mpq(3, (2 * s + 1) * (s - 3)) # c = 3 / ((2 * s + 1) * (s - 3))
65      return c * sum([2 * coefficients[m] * coefficients[s - m] if m != s - m else
            coefficients[m] * coefficients[s - m] for m in range(2, s - 2 + 1) if m <= s - m])
66
67
68  # this functions computes the series expansion of wp(z) for an elliptic
69  # curve in short Weierstrass form W
70  def weierstrass_p(W, p, k, z):
71      # W -> Elliptic curve in short Weierstrass form over Z / p^k z
72      # z -> integer such that p divides z
73
74      a, b = W
75      g2, g3 = - 4 * a, - 4 * b
76      z = gmpy2.mpq(z)
77      wp_z = z # z^3 * wp(z) = z (mod p)
78      s, e, e_1 = 2, 0, max(1, p_adic_eval_fraction(z, p))
79      # list of the coefficients
80      # set coefficients[0] as 0
81      # set coefficients[1] as 0
82      coefficients = [gmpy2.mpq(0), gmpy2.mpq(0)]
83      while e < k:
84          c_s = get_exp_coefficient(g2, g3, s, coefficients)
85          coefficients.append(c_s)
86          r = p_adic_eval_fraction(c_s, p) # max j such that p^j divides c_s
87          e_2 = 2 * s - 2 # exponent of the s-th term of the series expansion of wp(z)
88          e = e_1 * e_2 + r # p-adic evaluation of the s-th term of the series expansion of wp
                (z)
89          if e < k: # c_s z^(e_2) is not zero modulo p^k
90              wp_z += c_s * (z ** e_2) # wp(z) = wp(z) + c_s * z^(2s - 2)
91          s += 1
92      return wp_z # wp(z)
93
94
95  # this functions computes a point belonging to Im(Exp_W), where W is an elliptic
96  # curve in short Weierstrass form
97  def exp(W, p, k, z):
98      # W -> Elliptic curve in short Weierstrass form over Z / p^k z
99      # z -> integer such that p divides z
100
101     a, b = W
102     g2, g3 = - 4 * a, - 4 * b
103
104     z = gmpy2.mpq(z)
105     z3 = z ** 3
106     # z^3 * wp(z) = z (mod p)
107     # z^3 * wp'(z) = -2 (mod p)
108     wp_z, der_wp_z = z, gmpy2.mpq(-2)
109     s, e, e_1 = 2, 0, max(1, p_adic_eval_fraction(z, p))
110     # list of the coefficients
111     # set coefficients[0] as 0
112     # set coefficients[1] as 0
113     coefficients = [gmpy2.mpq(0), gmpy2.mpq(0)]
114     while e < k:
115         c_s = get_exp_coefficient(g2, g3, s, coefficients)
116         coefficients.append(c_s)
117         r = p_adic_eval_fraction(c_s, p) # max j such that p^j divides c_s
118         e_2 = 2 * s - 2 # exponent of the s-th term of the series expansion of wp(z)
119         e = e_1 * ((e_2 - 1) + 3) + r # p-adic evaluation of the s-th term of the series
                expansion of (z^3 * wp'(z))
120         if e < k: # c_s z^3 z^(e_2 - 1) is not zero modulo p^k
121             wp_z += c_s * (z ** (e_2 + 3)) # wp(z) = wp(z) + c_s * z^3 * z^(2s - 2)
122             der_wp_z += e_2 * c_s * (z ** (e_2 + 3 - 1)) # wp'(z) = wp'(z) + (2s - 2) * c_s
                    * z^3 * z^(2s - 2 - 1)
123         s += 1
124     P = (z3, wp_z, der_wp_z / 2) # P = [z^3 : z^3 * wp(z), z^3/2 wp'(z)]
125     return reduce_point(P, p, k)
126
127
128 # this functions checks if the projective point P is equivalent to the projective point Q
129 def are_equals(P, Q, mod=None):
130     # P -> point belonging to an elliptic curve W
131     # Q -> point belonging to an elliptic curve W
132
133     zp, xp, yp = P
134     zq, xq, yq = Q
135     if mod is None:
136         return xp * zq == xq * zp and yp * zq == yq * zp and xp * yq == xq * yp
137     else:
138         return (xp * zq - xq * zp) % mod == (yp * zq - yq * zp) % mod == (xp * yq - xq * yp
                ) % mod == 0
139
140
141 # this functions computes the inverse of the function "exp"
142 def log(W, p, k, P):
143     # W -> Elliptic curve in short Weierstrass form over Z / p^k z
144     # P -> point belonging to Im(Exp_W)
```

```
145
146     a, b = W
147     q = p ** k # q = p^k
148     g2, g3 = - 4 * a, - 4 * b
149     Z, X, Y = P
150     if k <= 4 or Z == 0:
151         XY = gmpy2.mpq(-X) / Y # XY = - X / Y
152         z = (XY.numerator * gmpy2.invert(XY.denominator, q)) % q # z = (- X / Y) mod (p ^ k)
153         return gmpy2.mpq(z)
154     p5 = p ** 5 # p5 = p ^ 5
155     z = gmpy2.mpq(X, Y) * (-1 + gmpy2.mpq(g2, 20) * gmpy2.mpq(Z ** 2, X ** 2)) # z = (X / Y
          ) * (-1 + (g2/20) * (Z / X)^2)
156     z = gmpy2.mpq((z.numerator * gmpy2.invert(z.denominator, p5)) % p5) # z = z (mod p^5)
157     if k == 5:
158         return z
159     c_set = set([z]) # init c_set as a singleton with z (mod p^5)
160     for i in range(6, k + 1):
161         curr_c_set = set([])
162         p_i_prev = p ** (i - 1) # p_i_prev = p ^ (i - 1)
163         p_i_curr = p * p_i_prev # p_i_curr = p ^ i
164         for z in c_set:
165             # computes the conjugates of z in c_set
166             for j in range(p):
167                 z_curr = gmpy2.mpq(z + j * p_i_prev) # z_curr = z + j * p^(i - 1)
168                 wp_z_curr = weierstrass_p(W, p, k, z_curr) # wp(z_curr) (mod p^k) = L / M
169                 # check if wp(z_curr) is equivalent to X / Z modulo p^i, that is,
170                 # check if L * Z - M * X = 0 (mod p^i)
171                 if (Z * wp_z_curr.numerator - X * wp_z_curr.denominator) % p_i_curr == 0:
172                     if are_equals(P, exp(W, p, k, z_curr), mod=q): # if P = exp(z_curr) mod p
                          ^k
173                         return z_curr
174                     else: # z_curr is a conjugate but not the full logarithm of P
175                         curr_c_set.add(z_curr)
176         c_set = curr_c_set.copy() # c_set = curr_c_set
177     for z in c_set:
178         if z < q and are_equals(P, exp(W, p, k, z), mod=q):
179             return z
180     raise Exception("The logarithm of {} not exists modulo {}^{}".format(P, p, k))
181
182
183 # this functions computes the projective opposite of a point P belonging to an
184 # elliptic curve in short Weierstrass form W
185 def projective_opposite(P):
186     zp, xp, yp = P
187     return (zp, xp, -yp)
188
189
190 # this function checks if P is the point at infinity
191 def is_infinity_point(P):
192     if P[0] == P[1] == 0 and P[2] != 0:
193         return True
194     return False
195
196
197 # this functions computes the projective sum P + Q for an
198 # elliptic curve in short Weierstrass form W
199 def projective_add(W, P, Q, O):
200     # W -> Elliptic curve in short Weierstrass form over rationals
201     # P -> point belonging to W
202     # Q -> point belonging to W
203     # O -> the point at infinity
204
205     if is_infinity_point(P): # if P = Omega
206         return Q
207     if is_infinity_point(Q): # if Q = Omega
208         return P
209
210     a4, a6 = W
211
212     zp, xp, yp = P
213     zq, xq, yq = Q
214
215     zpzq = zp * zq
216     xpzq, xqzp = xp * zq, xq * zp
217     ypzq, yqzp = yp * zq, yq * zp
218
219     if xpzq == xqzp and ypzq == - yqzp: # if P = -Q
220         return O
221
222     if xpzq == xqzp and ypzq == yqzp: # if P = Q -> doubling formulas
223         E = 3 * xp * xp + a4 * zp * zp
224         D = zp * (2 * yp)
225     else: # if P is different from Q -> add formulas
226         E = yqzp - ypzq
227         D = xqzp - xpzq
228
```

```
229     D2 = D ** 2
230     H = zpzq * E * E - D2 * (xpzq + xqzp)
231     F = zq * D2 * (xp * E - D * yp) - H * E
232     z, x, y = zpzq * D2 * D, H * D, F
233     return (z, x, y)
234
235
236 # this functions computes n * P for an elliptic curve in short Weierstrass form W
237 def projective_mul(W, P, n):
238     # W -> Elliptic curve in short Weierstrass form over rationals
239     # P -> point belonging to W
240     # n -> integer
241
242     # if n < 0, then set P as its opposite according to the group law of W
243     P = P if n >= 0 else projective_opposite(P)
244     one = (gmpy2.mpq(0), gmpy2.mpq(0), gmpy2.mpq(1)) # one = Omega
245     Q = (gmpy2.mpq(0), gmpy2.mpq(0), gmpy2.mpq(1)) # Q = Omega
246     n = abs(int(n)) # n = |n|
247     for d in bin(n)[2:]: # d = binary representation of n
248         Q = projective_add(W, Q, Q, one) # Q = Q + Q
249         if d == "1":
250             Q = projective_add(W, Q, P, one) # Q = Q + P
251     return Q
252
253
254 # this functions transforms a point P, whose projective coordinates are rationals,
255 # in an equivalent point Q, whose projective coordinates are integers
256 def normalize(P):
257     # compute the LCM of the denominators of the projective coordinates of P
258     t = gmpy2.lcm(gmpy2.lcm(P[0].denominator, P[1].denominator), P[2].denominator)
259     Q = (P[0] * t, P[1] * t, P[2] * t)
260     # compute the GCD of the numerators of the projective coordinates of Q
261     t = gmpy2.gcd(gmpy2.gcd(Q[0].numerator, Q[1].numerator), Q[2].numerator)
262     Q = (Q[0] / t, Q[1] / t, Q[2] / t)
263     return Q
264
265
266 # this functions computes the discrete logarithm of a point Q = n * P, for some integer n,
267 # belonging to an elliptic curve in short Weierstrass form W over the rationals
268 def ecdlp(W, P, Q, p, order):
269     # W -> Elliptic curve in short Weierstrass form over rationals
270     # P -> point such that (P - Omega) is a generator of Jacobian(W)
271     # Q -> point such that (Q - Omega) = h * (P - Omega)
272     # order -> order of Jacobian(W) over Z / p Z
273
274     if are_equals(P, Q): return 1
275     F = GF(p)
276     W_a4, W_a6 = gmpy2.mpq(str(W.a4())), gmpy2.mpq(str(W.a6()))
277
278     # compute the value "a" such that h = a (mod order)
279     a = 0
280     P_mod = P.change_ring(F) # P (mod p)
281     try:
282         Q_mod = Q.change_ring(F) # Q (mod p)
283     except ZeroDivisionError:
284         # the discrete logarithm of Q is a multiple of order
285         # Q = j * order * P, for some integer j
286         curve = P.parent() # get curve W from P
287         omega = curve(0) # point at infinity
288         Q_mod = omega.change_ring(F) # set Mod(Q) as the point at infinity over GF(p)
289     while a < order and Q_mod != a * P_mod:
290         a += 1
291
292     mP_list = normalize([gmpy2.mpq(str(e)) for e in (1,) + (-P).xy()]) # -P
293     P_list = normalize([gmpy2.mpq(str(e)) for e in (1,) + P.xy()]) # P
294     Q_list = normalize([gmpy2.mpq(str(e)) for e in (1,) + Q.xy()]) # Q
295
296     O = (gmpy2.mpq(0), gmpy2.mpq(0), gmpy2.mpq(1)) # O = Omega
297     aP = projective_mul((W_a4, W_a6), mP_list, a) # aP = - a * P
298     R = projective_add((W_a4, W_a6), Q_list, aP, O) # R = Q - a * P
299     S = projective_mul((W_a4, W_a6), P_list, order) # S = order * P
300
301     h, i = a, 2
302     while h * P != Q:
303         F_i = Zmod(p ** i) # Z / p^i Z
304         # a4 -> W_a4 (mod p^i)
305         # a6 -> W_a6 (mod p^i)
306         a4, a6 = gmpy2.mpq(str(F_i(W_a4))), gmpy2.mpq(str(F_i(W_a6)))
307         R_mod = tuple([gmpy2.mpq(str(F_i(e))) for e in R]) # R (mod p^i)
308         S_mod = tuple([gmpy2.mpq(str(F_i(e))) for e in S]) # S (mod p^i)
309         l1 = log((a4, a6), p, i, S_mod) / p # l1 = (Log(S) (mod p^i)) / p = (Log(order * P)
                (mod p^i)) / p
310         l2 = log((a4, a6), p, i, R_mod) / p # l2 = (Log(R) (mod p^i)) / p = (Log(Q - a * P)
                (mod p^i)) / p
311         if l1 != 0:
312             m = i - 1 # solution modulo p^(i - 1)
```

```
313             F_m = Zmod(p ** m)
314             l2l1 = l_2 / l1 # l2 / l1
315             if F_m(l2l1.denominator).is_unit(): # check if l2 / l1 is invertible modulo p^(i
                    - 1)
316                 r = int(F_m(l2 / l1)) # r = (l2 / l1) (mod p^(i - 1))
317                 h = int(a + r * order) # h = a + r * order
318         i += 1
319     return h
320
321
322 # this function checks if the implemented funcion "ecdlp" works for
323 # the elliptic curve in short Weierstrass form given by the equation
324 # y^2 = x^3 - x + 1/4
325 def test_ecdlp():
326     p = 3
327     h = 57
328     a4, a6 = (gmpy2.mpq(-1), gmpy2.mpq(1, 4))
329     E = EllipticCurve(QQ, (a4, a6)) # y^2 = x^3 - x + 1/4
330     curve_order_mod_p = 7 # card(J(E) over GF(p)) = 7
331     P = E.point((2, 5/2)) # (2, 5/2)
332     Q = h * P
333     computed_h = ecdlp(E, P, Q, p, curve_order_mod_p)
334     print("Real h -> {}".format(h))
335     print("Computed h -> {}".format(computed_h))
336
337
338 if __name__ == "__main__":
339     test_ecdlp()
```

Script B.1: Python code that implements the algorithms in appendix A.1

B.2 AG Goppa codes for Edwards curves

In this section, we show our implementation code in script B.2 of the algorithms in appendix A.2. Additionally, the function `test_goppa` compute the generator matrix, and the parity-check matrix that we used in section 11.3.

```
1  # this function computes a basis for the Riemann-Roch space L(D) over
2  # GF(p), where D = P + (t - 1) O, with O = (0, 1)
3  def edwards_rr_basis(P, p, t):
4      # P -> point belonging to an Edwards curve E
5      # p -> odd prime defining the charactestic of the ground field of E
6      # t -> integer defining the degree of the divisor D = P + (t - 1) * O
7      # used to compute the basis of L(D)
8
9      R.<X, Y> = PolynomialRing(GF(p)) # GF(p)[X, Y]
10     (a, b) = P
11     F_0 = R(1)
12     F_2 = 1 / (Y - 1)
13     F_3 = ((Y + 1) / X) * F_2
14     if P == (0, 1): # if P = O
15         F = [F_0, F_2, F_3]
16         t += 1
17     elif P == (0, -1): # if P = O'
18         F_1 = 1 / X
19         F = [F_0, F_1, F_2, F_3]
20     elif P == (1, 0): # if P = H
21         F_1 = ((X + 1) * (Y + 1)) / (X * Y)
22         F = [F_0, F_1, F_2, F_3]
23     elif P == (-1, 0): # if P = H'
24         F_1 = ((X - 1) * (Y + 1)) / (X * Y)
25         F = [F_0, F_1, F_2, F_3]
26     else: # if P is different from O, O', H, and H'
27         F_1 = (X * (Y + b)) / ((X - a) * (Y - 1))
28         F = [F_0, F_1, F_2, F_3]
29     while len(F) < t:
30         i = len(F)
31         # h = floor(i / 2)
32         # m = i (mod 2)
33         h, m = divmod(i, 2)
34         if m == 0: # i = 2h
35             F_i = F_2 * F[2*h - 2]
36         else: # i = 2h + 1
37             F_i = ((Y + 1) / X) * F[2 * h]
38         F.append(F_i)
39     return F
40
41
```

```
42 # this function computes a generator matrix for an algebraic-geometric
43 # Goppa code for an Edwards curve E
44 def goppa_edwards_gen_matrix(P, F, p):
45     # P -> list of point belonging to an Edwards curve E defined over GF(p)
46     # F -> list of basis functions belonging to the Riemann-Roch space L(D),
47     # where D = Q + m O, with Q and O = (0, 1) belonging to E
48     # p -> characteristic of the ground field of E
49     n, m = len(F), len(P)
50     G = matrix.zero(GF(p), n, m)
51     for i in range(n):
52         for j in range(m):
53             x, y = P[j]
54             G[i, j] = F[i](x, y)
55     return G.echelon_form() # compute the standard form of G
56
57
58 def test_goppa():
59     t = 5 # degree of the divisor D used to compute the basis of L(D)
60     p = 17 # characteristic of the ground field of the Edwards curve
61     P = (2, 15) # point belonging to the Edwards curve E defined by the equation x^2 + y^2 =
          1 + 10 x^2 y^2
62     F = edwards_rr_basis(P, p, t) # compute the basis of L(P + (t - 1) O), where O = (0, 1)
63     # Q -> list of points belonging to the Edwards curve E
64     Q = [(5, 8), (5, 9), (6, 3), (6, 14), (8, 5), (8, 12), (9, 5)]
65     # compute the generator matrix (in standard form) of the algebraic-geometric Goppa code
66     # for the Edwards curve E by using the list of points in "Q",
67     # and the functions in "F"
68     G = goppa_edwards_gen_matrix(Q, F, p)
69     r = len(Q) - t
70     # compute the parity-check matrix of the code
71     # matrix.identity(r) -> identity matrix of order "r"
72     # -G[:, -r:].T -> takes the opposite of the transpose of the last "r" columns of "G"
73     H = (-G[:, -r:].T).augment(matrix.identity(r))
74     print("Basis functions -> {}".format(F))
75     print("\nGenerator matrix ->")
76     print(G)
77     print("\nParity-check matrix ->")
78     print(H)
79
80
81 if __name__ == "__main__":
82     test_goppa()
```

Script B.2: Python code that implements the algorithms in appendix A.2

B.3 AG Goppa codes for hyperelliptic curves

In this section, we show our implementation code in script B.3 of the algorithms in appendix A.3. Additionally, the function `test_goppa` compute the generator matrix, and the parity-check matrix for an algebraic-geometric Goppa code for an hyperelliptic curve $\mathcal{H}$ of genus $g \geq s = 11$ over GF(101).

```
1 import itertools
2 import functools
3 import operator
4
5
6 set_random_seed(1871) # set the seed for the pseudo-random number generator used by Sage
7
8
9 # this function computes a random pair (x_r, y_r) such that u(x_r) is different
10 # from zero, x_r is not in the set "evals"
11 def get_random_point(F, ux, evals):
12     # F -> field used to generate a random pair (x_r, y_r)
13     # ux -> polynomial u(x) used to check if u(x_r) is different from zero
14     # evals -> set of abscissas such that x_r not belongs to "evals"
15     x_r = F.random_element()
16     while ux(x_r) == 0 or x_r in evals: # if u(x_r) = 0 or x_r is an element of "evals",
          then generate a new "x_r"
17         x_r = F.random_element()
18     y_r = F.random_element()
19     while y_r == 0: # reject any pair such that y_r = 0
20         y_r = F.random_element()
21     return (x_r, y_r)
22
23
24 # this function checks if an algebraic-geometric Goppa code for the set of
25 # functions "funcs", and the set of "points" is MDS
```

```
26 def check_MDS(F, funcs, points):
27     # F -> GF(p)
28     # funcs -> set of functions belonging to a basis of the Riemann-Roch space
29     # L(D) for an hyperelliptic curve H over F
30     # points -> set of points belonging to the curve H
31 
32     n, m = len(funcs), len(points)
33     G = matrix.zero(F, n, m) # matrix of zeros over the field F, whose dimensions are n x m
34 
35     # computes the generator matrix by using the points passed to this function
36     try:
37         for i in range(n):
38             for j in range(m):
39                 x_r, y_r = points[j]
40                 G[i, j] = funcs[i](x=x_r, y=y_r)
41 
42         # check if all the minors of order "n" of G are full rank
43         ranks = {}
44         for cols in itertools.combinations(range(G.ncols()), n):
45             GG = G.matrix_from_columns(cols)
46             r = GG.rank()
47             ranks[r] = ranks.get(r, []) + [cols]
48         return len(list(ranks)) == 1 and n in ranks
49     except ZeroDivisionError:
50         return False
51 
52 
53 # this function computes a basis of the Riemann-Roch space L(D), where
54 # D = ((u(x), v(x)) + (n + g - 1) Omega) is a divisor belonging to the Jacobian of
55 # an hyperelliptic curve H, whose genus "g" is greater or equal to "s", over GF(p).
56 # Additionally, this function determines "m" random points in order to compute
57 # the generator matrix for an algebraic-geometric Goppa code for the curve H
58 def goppa_hec_generator_matrix(n, m, p, s):
59     # m -> number of random points to determine
60     # n -> dimension of the basis of the Riemann-Roch space
61     # p -> characteristic of the ground field of the hyperelliptic curve H of genus "g"
62     # given by the equation (v(x))^2 = f(x) + k(x) u(x), for some k(x) in GF(p)[x]
63     # s -> integer such that g >= s
64 
65     F = GF(p)
66     R.<x, y> = PolynomialRing(F) # GF(p)[x, y]
67     GFx = PolynomialRing(F, 'x') # GF(p)[x]
68 
69     if s == 0:
70         ux, vx = GFx(1), GFx(0)
71     else:
72         l = randint(0, s - 1) # l is a random integer in [0, s - 1]
73         ux = GFx(1)
74         x_var = GFx.gen() # x of GF(p)[x]
75         for i in range(s):
76             x_i = F.random_element()
77             ux *= (x_var - x_i)
78         vx = GFx.random_element(l) # random polynomial of degree l in GF(p)[x]
79         # if v(x_i) = 0 for multiple abscissas x_i,
80         # then (u(x), v(x)) is not a Mumford representation of a divisor
81         while gcd(gcd(ux, ux.derivative()), vx) != 1:
82             vx = GFx.random_element(l)
83 
84     # compute the minimal genus for the hyperelliptic curve
85     # with these parameters
86     g = s
87     while (2 * g + 2) - s - (m // 2) - (m % 2) < 0:
88         g += 1
89 
90     degfx = 2 * g + 1 # degree of the polynomial f(x)
91     degD = n + g - 1 # deg(D)
92 
93     h = (degD - s) // 2
94     k = (degD - (degfx - s)) // 2
95 
96     psi = (y + vx) / ux # psi = (y + v(x)) / u(x)
97     funcs = [x^i for i in range(0, h + 1)] + [psi * x^j for j in range(0, k + 1)]
98 
99     points = []
100     evals = set([0])
101     counter, MAX_TRIES = 0, 10
102     while len(points) < m:
103         x_r, y_r = get_random_point(F, ux, evals)
104         if m % 2 == 1 and len(points) + 1 == m:
105             R = points + [(x_r, y_r)]
106         else:
107             R = points + [(x_r, y_r), (x_r, - y_r)]
108         if len(points) >= n - 2 and not check_MDS(F, funcs, R):
109             counter += 1
110         else:
111             counter = 0
```

```
112             points = R[:] # points = copy(R)
113             evals.add(x_r) # evals = evals U {x_r}
114         if counter >= MAX_TRIES:
115             points = []
116             evals = set([0])
117
118     # computes the generator matrix for the AG Goppa code
119     G = matrix.zero(F, n, m)
120     for i in range(n):
121         for j in range(m):
122             x_r, y_r = points[j]
123             G[i, j] = funcs[i](x=x_r, y=y_r)
124     return G.echelon_form()
125
126
127 def test_goppa():
128     # OUTPUT OF THIS FUNCTION
129     # Generator matrix ->
130     # [ 1 0 0 0 0 23 54 2 90 87]
131     # [ 0 1 0 0 0 78 84 35 98 0]
132     # [ 0 0 1 0 0 81 83 32 65 5]
133     # [ 0 0 0 1 0 20 21 72 88 47]
134     # [ 0 0 0 0 1 1 62 62 64 64]
135     #
136     # Parity-check matrix ->
137     # [ 78 23 20 81 100 1 0 0 0 0]
138     # [ 47 17 18 80 39 0 1 0 0 0]
139     # [ 99 66 69 29 39 0 0 1 0 0]
140     # [ 11 3 36 13 37 0 0 0 1 0]
141     # [ 14 0 96 54 37 0 0 0 0 1]
142
143     s = 11
144     p = 101
145     n, m = 5, 10
146     G = goppa_hec_generator_matrix(n, m, p, s)
147     r = m - n
148     H = (-G[:, -r:].T).augment(matrix.identity(r))
149     print("Generator matrix ->")
150     print(G)
151     print("\nParity-check matrix ->")
152     print(H)
153
154
155 if __name__ == "__main__":
156     test_goppa()
```

Script B.3: Python code that implements the algorithms in appendix A.3

B.4 HL-codes

In this section, we show the implementations codes in script B.4, and the benchmark results of our implementation with respect to the algorithms and protocols in appendix A.4. Specifically, we ran the protocols 2 and 3 one hundred times on random codewords, and measured the WALL time needed to encrypt and decrypt those codewords, respectively.

Since we implemented our algorithms in Python, we optimize the code representing each redundancy relation as an integer of $n = 2^m$ bits. More precisely, for a given word $\mathbf{x} = \mathbf{a} \cdot G + \mathbf{e} = (x_0, x_1, \ldots, x_{n-1})$, we associated each redundancy relation $\underset{e}{\Delta}\, x_i = \sum x_j$ to an integer such that its j-th most significant bit (MSB) is equal to 1. For instance, if $n = 16$ and the redundancy relation is $x_0 + x_1 + x_4 + x_5$ (that is, the first redundancy relation of a_6 in equation (13.18)), then $x_0 + x_1 + x_4 + x_5 = (1,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0)_2 = 52224$. By using this representation, we pre-computed the bit-arrays for each component of $\mathbf{x}$ and used them to optimize algorithm 14 and algorithm 20. In particular, in the algorithm 14, we composed the redundancy relation as a bitwise-OR ($|$) of the corresponding bit-arrays. For example, $x_0 + x_1$ is equivalent to the bitwise-OR of the bit-array of x_0 and the bit-array of x_1, which are $(1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)_2 = 32768$

and $(0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)_2 = 16384$ respectively, that is, $x_0 + x_1 = 32768|16384 = 49152$. In this way, the set of redundancy relations for each a_j is a set of integers. Moreover, in algorithm 20, we also represented the word $\mathbf{x}$ as an integer and the evaluation of each redundancy relation is equivalent to counting the number of bits equal to 1 in $\left(\underset{e}{\Delta}\, x_i\right)$ & $\mathbf{x}$, where (&) is the bitwise-AND between the two integer representations. For instance, in order to evaluate the redundancy relation $x_0 + x_1 + x_4 + x_5$, we count the number of bits equal to 1 in the integer 52224 & $\mathbf{x}$. In particular, since $1 + 1 = 0$ in binary arithmetic, it follows that the outcome of the redundancy relation is equal to 1 if and only if the number of ones among of the involved components is odd. Therefore, if the number of ones in $\left(\underset{e}{\Delta}\, x_i\right)$ & $\mathbf{x}$ is odd, then the outcome of $\underset{e}{\Delta}\, x_i$ is equal to 1.

In fig. B.4 and fig. B.5, we plot the encryption and the decoding times (in seconds) respectively, using our algorithms for a random HL-code with parameter $m = 10$. Additionally, in fig. B.7 and fig. B.8, we plot the encryption and the decoding times (in seconds) respectively, using our algorithms for a random HL-code with parameter $m = 12$.

Finally, in table B.2, we show the mean times for the encryption and the decryption, as well as the time for the key-generation phase. It is worth recalling that in protocol 1, we retrieve the partial generator matrix (computed using algorithm 18) from the disk. In particular, for $m = 12$, we have that protocol 1 required 6.59 seconds, with 3.2 seconds spent on reading the matrix from the disk.

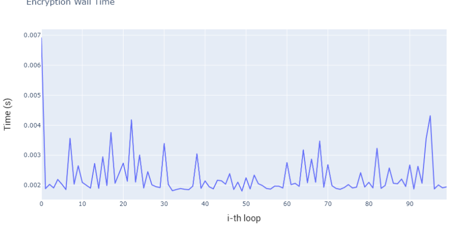

FIGURE B.4: Encryption times for an HL-code of parameter $m = 10$

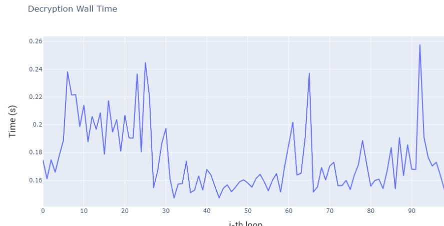

FIGURE B.5: Decryption times for an HL-code of parameter $m = 10$

FIGURE B.6: WALL-time for an HL-code of parameter $m = 10$

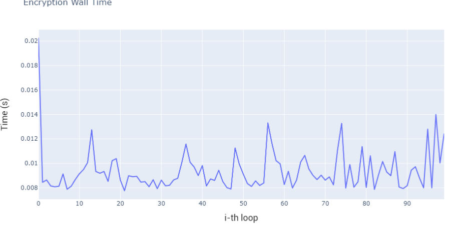

FIGURE B.7: Encryption times for an HL-code of parameter $m = 12$

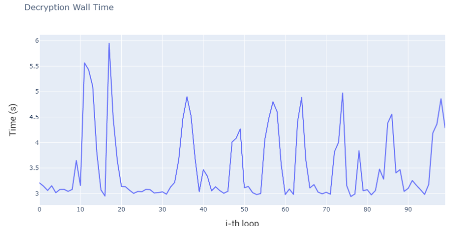

FIGURE B.8: Decryption times for an HL-code of parameter $m = 12$

FIGURE B.9: WALL-time for an HL-code of parameter $m = 12$

Table B.2: Mean time for the protocol

m	Key-Gen (s)	Mean encryption time (s)	Mean decryption time (s)
10	0.3297	0.0023	0.1758
12	6.5884	0.0094	3.5341

```
1  import os
2  import time
3  import pickle
4  import shutil
5  import operator
6  import itertools
7  import functools
8  import numpy as np
9  from sage.coding.decoder import Decoder
10 from sage.matrix.constructor import random_unimodular_matrix
11 from plotly.graph_objs import Layout, Figure, Bar, Scattergl # For OpenGL
12
13
14 DEBUG = False
15 ENABLE_FILE_LOG = False
16 LOG_FILE = "./infosetdec.txt"
17
18
19 # this function computes the current wall time
20 def get_time(perf=False):
21     return time.process_time() if not perf else time.perf_counter()
22
23
24 # this function prints the text into a log file or to console
25 def print_log(text):
26     if ENABLE_FILE_LOG:
27         with open(LOG_FILE, "a") as file:
28             file.write("{}\n".format(text))
29     else:
30         print(text)
31
32
33 # this function computes the standard form of a generator matrix
34 # for an error-correcting code
35 def canonical_form(M, d):
36     LCode = codes.encoders.LinearCodeSystematicEncoder(LinearCode(M, d=d))
37     P = Permutation(LCode.systematic_permutation()).to_matrix()
38     M = LCode.generator_matrix() * P
39     return M, LCode.systematic_permutation()
40
41
42
43 # class used to save data on disk
44 class HLCodeRelationData(object):
45     def __init__(self, data):
46         self.data = data
47     def get_data(self):
48         return self.data
49
50
51 # this class handles the redundancy relations for a HL-code
52 class HLCodeRelation(object):
53     def __init__(self, m):
54         # m -> parameter of the HL-code
55         self.m = m
56         # n = 2^m
57         # k = 2^(m - 1)
58         self.n, self.k = 1 << m, 1 << (m - 1)
59         # bitmasks for every integer in [0, n - 1]
60         self.bit_mask = {i: 1 << (self.n - 1 - i) for i in range(self.n)}
61         self.bit_mask_all = (1 << self.n) - 1
62     @staticmethod
63     # this function gets the saved redundacy relations from the disk
64     def get_database(DIR, m):
65         start = get_time()
66         filepath = os.path.join(DIR, "{}".format(m))
67         if not os.path.exists(filepath):
68             D = HLCodeRelation(m)
69             D.build_partial_relations()
70             return (D.positions, D.comb_relations)
71         with open(filepath, "rb") as f:
72             D = pickle.load(f)
73         data = D.get_data()
```

```
74          end = get_time() - start
75          print_log("Got database in {}s".format(end))
76          return data
77      # this function computes a subset of the redundancy relations and save them on the disk
78      def build_database(self, DIR):
79          if DIR is not None and not os.path.exists(DIR):
80              os.mkdir(DIR)
81          self.build_partial_relations()
82          D = HLCodeRelationData((self.positions, self.comb_relations))
83          with open(os.path.join(DIR, "{}".format(self.m)), "wb") as f:
84              pickle.dump(D, f, protocol=-1)
85      # this function computes the complete set of redundacy relations
86      def build_relations(self, Y_set, DIR=None):
87          start = get_time()
88          self.relations = [{} for _ in range(1, self.m // 2)]
89          if DIR is None:
90              self.build_partial_relations()
91          else: # get relations from the disk
92              self.positions, self.comb_relations = HLCodeRelation.get_database(DIR, self.m)
93
94          l = int(1)
95          for e in sorted(self.comb_relations, key=len):
96              m = int(len(e) - 1)
97              self.relations[m][l] = list(v for k, v in self.comb_relations[e].items() if k in
                        self.positions[e])
98              l += 1
99
100         y_start = get_time()
101         curr_relations = {}
102         # compute the redundancy relations related to the Y-set
103         for indices in Y_set:
104             self.__compute_relation_comb(indices)
105             curr_relations[l] = list(v for k, v in self.comb_relations[indices].items() if k
                        in self.positions[indices])
106             l += 1
107         self.relations.append(curr_relations)
108         self.relations = self.relations[::-1]
109         end = get_time()
110         y_end = end - y_start
111         end = end - start
112         print_log("Built partial relations for Y-set in {}s".format(y_end))
113         print_log("Built relations in {}s".format(end))
114     # this function computes a subset of the redundancy relations
115     def build_partial_relations(self):
116         start = get_time()
117         self.positions = {}
118         self.comb_relations = {}
119         for m in range(1, self.m // 2):
120             for indices in itertools.combinations(range(1, self.m + 1), m):
121                 int_indices = tuple(int(e) for e in indices)
122                 self.__compute_relation_comb(int_indices)
123         end = get_time() - start
124         print_log("Built partial relations in {}s".format(end))
125     @functools.lru_cache(maxsize=None)
126     # this function toggles the k-th LSB of idx
127     def __get_k_mod(self, idx, k):
128         i_k = (idx >> (k - 1)) & 1
129         add_part = 1 << (k - 1)
130         return idx + (- add_part if i_k == 1 else add_part)
131     # this function computes the redundancy relations for a combination of rows
132     # of the generator matrix
133     def __compute_relation_comb(self, comb):
134         var_set = 0
135         self.positions[comb] = set([])
136         self.comb_relations[comb] = {}
137         if DEBUG: print_log("a_{} = ".format(comb))
138         n = self.bit_mask_all
139         while n != 0:
140             j = int(self.n - int(n).bit_length())
141             a = int(self.__compute_relation_idx(comb, j))
142             n ^^= a
143             self.comb_relations[comb][j] = a
144             self.positions[comb].add(j)
145             if DEBUG:
146                 s = " + ".join("x_{}".format(e) for e in sorted(a))
147                 print_log(" {}".format(s))
148     # this function computes the operator DELTA(idx) for "comb",
149     # where DELTA is the Reed's DELTA operator
150     def __compute_relation_idx(self, comb, idx):
151         if comb in self.comb_relations:
152             if idx in self.comb_relations[comb]:
153                 return self.comb_relations[comb][idx]
154         if len(comb) == 0: return 0
155         elif len(comb) == 1:
156             new_idx = self.__get_k_mod(idx, comb[0])
157             if DEBUG:
```

```
158                 print_log("\u0394 {} ({}) = {}".format(comb, idx, set([idx, new_idx])))
159             return self.bit_mask[idx] | self.bit_mask[new_idx]
160         else:
161             t_comb = comb[:-1]
162             if t_comb not in self.comb_relations:
163                 self.comb_relations[t_comb] = {}
164             new_idx = self.__get_k_mod(idx, comb[-1])
165             if DEBUG:
166                 print_log("\u0394 {} ({}) = \u0394 {} ({}) + \u0394 {} ({})".format(comb, idx,
                        comb[:-1], idx, comb[:-1], new_idx))
167             a = self.__compute_relation_idx(t_comb, idx)
168             self.comb_relations[t_comb][idx] = a
169             b = self.__compute_relation_idx(t_comb, new_idx)
170             self.comb_relations[t_comb][new_idx] = b
171             if DEBUG:
172                 print_log("\u0394 {} ({}) = {} + {}".format(comb, idx, a, b))
173             return a | b
174 
175 
176 # this class handles the decoding for a HL-code
177 class HLCodeDecoder(Decoder):
178     def __init__(self, code, Y_set, m, DIR=None):
179         # code -> the code class
180         # Y_set -> the Y-set used for this code
181         # m -> the parameter of the HL-code
182         # DIR -> directory path used to locate the redundacy relations already computed
183         super(HLCodeDecoder, self).__init__(
184             code, code.ambient_space(), code.encoder().__class__.__name__)
185         self.m = m
186         self.Y_set = Y_set
187         start = get_time()
188         self.G, self.H = self.__parity_check_matrix()
189         end = get_time() - start
190         print_log("\nParity-check matrix in {}s".format(end))
191         self.F = self.G.base_ring()
192         self.k, self.n = self.G.nrows(), self.G.ncols()
193         self.zero_vector_k = vector(self.F, self.k)
194         self.zero_vector_n = vector(self.F, self.n)
195         self.zero_syndrome = vector(self.F, self.n - self.k)
196 
197         if DIR is not None and not os.path.exists(DIR):
198             os.mkdir(DIR)
199         self.DIR = DIR
200 
201         start = get_time()
202         self.__compute_relations()
203         end = get_time() - start
204         print_log("\nSetup in {}s".format(end))
205     def _repr_(self):
206         return "HL-code decoder for the %s" % self.code()
207     def _latex_(self):
208         return "\textnormal{HL-code decoder for the } %s" % self.code()
209     def __eq__(self, other):
210         return isinstance((other, HLCodeDecoder) and self.code() == other.code())
211     def decoding_radius(self):
212         return (self.code()._minimum_distance - 1) // 2
213     # this funcion computes the parity-check matrix for a HL-code
214     # by determining the standard form for the generator matrix G
215     def __parity_check_matrix(self):
216         if hasattr(self, "H"): return self.G, self.H
217         G = self.code().generator_matrix()
218         k, n = G.nrows(), G.ncols()
219         if max(k, n) > 256: return G, None
220         G, P = canonical_form(G, self.code()._minimum_distance)
221         H = (-G[:, -(n-k):].T).augment(matrix.identity(G.base_ring(), n - k))
222         if DEBUG: assert G * H.T == matrix.zero(G.base_ring(), k, n - k)
223         if DEBUG: assert self.code().generator_matrix() * (H * Permutation(P).inverse().
                to_matrix()).T == matrix.zero(G.base_ring(), k, n - k)
224         return self.code().generator_matrix(), H * Permutation(P).inverse().to_matrix()
225     # this function computes the complete set of redundancy relations for this code
226     def __compute_relations(self):
227         rel = HLCodeRelation(self.m)
228         rel.build_relations(self.Y_set, DIR=self.DIR)
229         self.relations = rel.relations
230     # this function determines the errors for a "word", and for a
231     # set of redundancy relations passed to this functions.
232     # additionally, this function computes the proper decoding at this stage
233     def __error_from_relations(self, word, relations):
234         start = get_time()
235         curr_a = copy(self.zero_vector_k)
236         error_vector = copy(self.zero_vector_n)
237         for i in relations:
238             curr_relations = relations[i]
239             # count the number of redundancy relations equals to one for "word"
240             one_count, zero_count = 0, 0
241             j, mid_size = 0, len(curr_relations) // 2
```

```
242             while zero_count <= mid_size and one_count <= mid_size:
243                 one_count += bin(word & curr_relations[j])[2:].count("1") % 2
244                 zero_count = j + 1 - one_count
245                 j += 1
246             if one_count == zero_count:
247                 raise Exception("Unable to decode")
248             if one_count > zero_count:
249                 error_vector += self.G.row(i)
250                 curr_a[i] = 1
251         end = get_time() - start
252         if DEBUG:
253             print_log("\nComputed errors in {}s".format(end))
254         return error_vector, curr_a
255     # this function decodes a word "y" by applying a multilevel decoding
256     def decode_to_code(self, y):
257         # y = x + e = a G + e
258         # a = (a_1, , ...., a_k)
259         if self.H is not None and self.H * y == self.zero_syndrome: return y
260
261         a_vector = copy(self.zero_vector_k) # the word to decode
262         # multilevel decoding
263         error_vector = copy(y)
264         for relation in self.relations:
265             prev_error = copy(ZZ(list(error_vector)[::-1], base=2))
266             # curr_e -> error vector for the set "relation"
267             # curr_a -> decoded word for the set "relation"
268             curr_e, curr_a = self.__error_from_relations(prev_error, relation)
269             error_vector -= curr_e
270             a_vector += curr_a
271
272         # determine the value of the first bit of the decoded word
273         j, mid_size = 0, len(error_vector) // 2
274         one_count, zero_count = 0, 0
275         while zero_count <= mid_size and one_count <= mid_size:
276             one_count += int(error_vector[j])
277             zero_count = j + 1 - one_count
278             j += 1
279
280         if one_count == zero_count:
281             raise Exception("Unable to decode")
282         if one_count > zero_count:
283             error_vector -= self.G.row(0)
284             a_vector[0] = 1
285
286         codeword = y - error_vector
287         if DEBUG:
288             print_log("Error vector -> {}".format(error_vector))
289             if self.H is not None:
290                 assert self.H * codeword == self.zero_syndrome
291         return codeword, a_vector
292
293
294 HLCodeDecoder._decoder_type = {"hard-decision", "unique"}
295
296
297 # this class is used to save the partial data on disk
298 class ReedMullerData(object):
299     def __init__(self, data):
300         self.data = data
301     def get_data(self):
302         return self.data
303
304
305 # this function permutes a vector "v" according to a
306 # permutation P
307 def permute_vector(P, v):
308     r = copy(v)
309     for i, j in enumerate(P):
310         r[i] = v[j - 1]
311     return r
312
313
314 # this function computes the pairwise-product of two vectors
315 def pairwise_prod(a, b):
316     # a -> [a_0, a_1, ..., a_n]
317     # b -> [b_0, b_1, ..., b_n]
318     # returns [a_0 b_0, a_1 b_1, ..., a_n b_n]
319     return a.pairwise_product(b)
320
321
322 # this function computes the generator matrix for a Reed-Muller code of parameter m
323 def build_reed_muller_code(m):
324     if m % 2 != 0:
325         raise Exception("m have to be even")
326
327     main_start = get_time()
```

```
328     F2 = GF(2)
329
330     mid_m = m // 2
331     # n = 2^m
332     # k = 2^(m - 1)
333     # d = 2^(m / 2)
334     n, k, d = 1 << m, 1 << (m - 1), 1 << mid_m
335
336     print_log("Building a [{}, {}, {}] linear code...".format(n, k, d))
337
338     M = matrix.zero(F2, k, n)
339     row_count = 0
340
341     print_log("Computing first {} rows...".format(m + 1))
342     start = get_time()
343     # compute first (m + 1) rows of the generator matrix
344     zero_vec = vector(F2, [0] * n)
345     M.set_row(row_count, vector(F2, [1] * n))
346     row_count += 1
347     for l in range(m):
348         i = int(1 << l)
349         row = copy(zero_vec)
350         for j in range(i, n, 2*i):
351             row[j:j + i] = [1] * i
352         M.set_row(row_count, row)
353         row_count += 1
354     end = get_time() - start
355     print_log("End in {}s".format(end))
356
357     # compute the remaining rows of the generator matrix
358     # as a pairwise product of subsets the first (m + 1) rows
359     print_log("\nComputing products...")
360     start = get_time()
361     for i in range(2, mid_m):
362         if DEBUG: print_log("i -> {}, size -> {}".format(i, binomial(m, i)))
363         for indices in itertools.combinations(range(1, m + 1), i):
364             if DEBUG: print_log(indices)
365             v = vector(F2, M.row(indices[0]))
366             for idx in indices[1:]:
367                 v = v.pairwise_product(M.row(idx))
368             M.set_row(row_count, v)
369             row_count += 1
370     end = get_time() - start
371     print_log("\nEnd in {}s".format(end))
372
373     end = get_time() - main_start
374     print_log("\nEnding in {}s\n".format(end))
375     # returns the generator matrix, the number of the rows computed,
376     # and the minimum distance of the code
377     return M, row_count, d
378
379
380 # this function computes a random complement-free set of parameter "m"
381 def gen_complement_free_set(m):
382     X_set = []
383     for indices in itertools.combinations(range(m), m // 2):
384         v = [0] * m
385         for i in indices: v[i] = 1
386         X_set += [(tuple([i + 1 for i in indices]), v)]
387     t = len(X_set) // 2
388     A, B = X_set[:t], X_set[t:][::-1]
389
390     # shuffle A and B in the same way
391     seed = random()
392     set_random_seed(seed)
393     shuffle(A)
394     set_random_seed(seed)
395     shuffle(B)
396
397     Y_set, Y_set_indices = [], []
398     for i in range(t):
399         if randint(0, 1) == 0:
400             indices, v = A[i]
401         else:
402             indices, v = B[i]
403         Y_set += ["".join(map(str, v))]
404         Y_set_indices += [indices]
405     return Y_set, Y_set_indices
406
407
408 # build an HL-code with n = 2^m, k = 2^(m - 1), d = 2^(m / 2)
409 def build_hl_code(m, DIR=None):
410     main_start = get_time()
411
412     mid_m = m // 2
413     if DIR is None:
```

```
414         # compute the first "row_count" rows from a Reed-Muller code
415         M, row_count, d = build_reed_muller_code(m)
416     else:
417         # load the partial generator matrix from the disk
418         filepath = os.path.join(DIR, "rm_{}".format(m))
419         if not os.path.exists(filepath):
420             M, row_count, d = build_reed_muller_code(m)
421         else:
422             with open(filepath, "rb") as f:
423                 D = pickle.load(f)
424             M, row_count, d = D.get_data()
425
426     n, k = M.ncols(), M.nrows()
427
428     print_log("\nComputing complement-free Y set...")
429     start = get_time()
430     Y_set, Y_set_indices = gen_complement_free_set(m)
431     end = get_time() - start
432     print_log("Y set size -> {}".format(len(Y_set)))
433     print_log("End in {}s".format(end))
434
435     if DEBUG:
436         print_log("\nY set:")
437         for a in Y_set:
438             print_log(a)
439
440     # compute the remaining rows of the generator matrix by using the Y-set
441     print_log("\nComputing last rows...")
442     start = get_time()
443     for indices in Y_set_indices:
444         res = reduce(pairwise_prod, [M.row(idx) for idx in indices])
445         M.set_row(row_count, res)
446         row_count += 1
447     end = get_time() - start
448     print_log("End in {}s".format(end))
449
450     end = get_time() - main_start
451     print_log("\nEnding in {}s\n".format(end))
452     # returns the generator matrix, the Y-set as a set of indices,
453     # and the minimum distance of the code
454     return M, Y_set_indices, d
455
456
457 # this function computes the generator matrices for the Reed-Muller codes
458 # of parameter m, where m = m_start, m_start + 1, m_end,
459 # the redundancy relations for these rows, and save these data on the disk
460 def build_datasets(DIR, m_start, m_end):
461     if DIR is not None and not os.path.exists(DIR):
462         os.mkdir(DIR)
463     for m in range(m_start, m_end + 1, 2):
464         print("\nBuilding datasets for m = {}...".format(m))
465         with open(os.path.join(DIR, "rm_{}".format(m)), "wb") as f:
466             pickle.dump(ReedMullerData(build_reed_muller_code(m)), f)
467         HL = HLCodeRelation(m)
468         HL.build_database(DIR)
469
470
471 # this function decodes ntests random words by using a HL-code of parameter "m"
472 def decoding_test(m, ntests, DIR=None, DATADIR=None):
473     if DATADIR is not None and not os.path.exists(DATADIR):
474         os.mkdir(DATADIR)
475
476     wall_start = get_time(perf=True)
477     cpu_start = get_time()
478
479     F2 = GF(2)
480     G, Y_set, d = build_hl_code(m, DIR=DIR)
481     k, n = G.nrows(), G.ncols()
482     t = (d - 1) // 2
483     matrix_space = sage.matrix.matrix_space.MatrixSpace(F2, k)
484
485     curr_start = get_time()
486     perms = Permutations(n) # permutations of the indices in [1, n]
487     while True:
488         # compute a random invertible matrix
489         S = matrix_space.random_element()
490         if S.is_invertible(): break
491     end = get_time() - curr_start
492     print("S generated in {}s".format(end))
493     curr_start = get_time()
494     P = perms.random_element() # generate a random permutation in [1, n]
495     end = get_time() - curr_start
496     print("P generated in {}s".format(end))
497     curr_start = get_time()
498     inv_P = P.inverse() # compute the inverse of the permutation P
```

```
499       end = get_time() - curr_start
500       print("P inverse in {}s".format(end))
501       curr_start = get_time()
502       inv_S = S.inverse() # compute the inverse of the matrix S
503       end = get_time() - curr_start
504       print("S inverse in {}s".format(end))
505       # compute the matrix G' = S * G * P used to encript a vector
506       curr_start = get_time()
507       GG = S * G
508       GG.permute_columns(P) # permute the columns of (S * G) according to P
509       end = get_time() - curr_start
510       print("S G P computed in {}s".format(end))
511
512       LCode = LinearCode(G, d=d)
513
514       # initialize the class to add a random error vector of weight "t"
515       channel = channels.StaticErrorRateChannel(LCode.ambient_space(), t)
516       DEC = HLCodeDecoder(LCode, Y_set, m, DIR=DIR)
517
518       cpu_end = get_time()
519       wall_end = get_time(perf=True)
520       init_cpu_time = cpu_end - cpu_start
521       init_wall_time = wall_end - wall_start
522
523       enc_wall_times, dec_wall_times = [], []
524       enc_cpu_times, dec_cpu_times = [], []
525       decoded_message = None
526
527       print("\nParameters -> {}".format((n, k, d, t)))
528       for j in range(ntests):
529           print("Test #{}".format(j + 1), end="\r")
530           message = random_vector(F2, k) # random vector over GF(2) of length k
531
532           wall_start = get_time(perf=True)
533           cpu_start = get_time()
534           codeword = message * GG # codeword = m S G P
535           ciphertext = channel(codeword) # ciphertext = m S G P + e, for a random vector "e"
                  of weight "t"
536           cpu_end = get_time()
537           wall_end = get_time(perf=True)
538           enc_cpu_times.append(cpu_end - cpu_start)
539           enc_wall_times.append(wall_end - wall_start)
540
541           wall_start = get_time(perf=True)
542           cpu_start = get_time()
543           c_inv_P = permute_vector(inv_P, ciphertext) # c_inv_P = ciphertext * P^(-1) = m S G
                  + e P^(-1)
544           mSG, mS = DEC.decode_to_code(c_inv_P) # decode c_inv_P
545           decoded_message = mS * inv_S
546           cpu_end = get_time()
547           wall_end = get_time(perf=True)
548           dec_cpu_times.append(cpu_end - cpu_start)
549           dec_wall_times.append(wall_end - wall_start)
550           if message != decoded_message:
551               raise Exception("Unable to decode")
552
553       mean_enc_wall_time, mean_dec_wall_time = np.mean(enc_wall_times), np.mean(
              dec_wall_times)
554       mean_enc_cpu_time, mean_dec_cpu_time = np.mean(enc_cpu_times), np.mean(dec_cpu_times)
555       print("\nParameters -> {}".format((n, k, d)))
556       print("Init time (WALL) -> {}s".format(init_wall_time))
557       print("Init time (CPU) -> {}s".format(init_cpu_time))
558       print("Enc time (WALL) -> {}s".format(mean_enc_wall_time))
559       print("Enc time (CPU) -> {}s".format(mean_enc_cpu_time))
560       print("Dec time (WALL) -> {}s".format(mean_dec_wall_time))
561       print("Dec time (CPU) -> {}s\n".format(mean_dec_cpu_time))
562
563       # save results if a file *.csv
564       if DATADIR:
565           filepath = os.path.join(DATADIR, "{}.txt".format(m))
566           filepath_mean = os.path.join(DATADIR, "{}_mean.txt".format(m))
567           with open(filepath_mean, "w") as f:
568               f.write("Init time (WALL); Init time (CPU); Encryption time (WALL); Encryption
                      time (CPU); Decryption time (WALL); Decryption time (CPU)\n")
569               f.write("{}; {}; {}; {}; {}; {}\n".format(init_wall_time, init_cpu_time,
                      mean_enc_wall_time, mean_enc_cpu_time, mean_dec_wall_time,
                      mean_dec_cpu_time))
570           with open(filepath, "w") as f:
571               f.write("Encryption time (WALL); Encryption time (CPU); Decryption time (WALL);
                      Decryption time (CPU)\n")
572           with open(filepath, "a") as f:
573               for (enc_wall, enc_cpu, dec_wall, dec_cpu) in zip(enc_wall_times, enc_cpu_times,
                      dec_wall_times, dec_cpu_times):
574                   f.write("{}; {}; {}; {}\n".format(enc_wall, enc_cpu, dec_wall, dec_cpu))
575       return plot_data(enc_wall_times, dec_wall_times)
```

```
576
577
578 # this function plots the wall times of coding and decoding
579 def plot_data(enc_wall_times, dec_wall_times):
580     x_axes = list(range(0, len(enc_wall_times)))
581     enc_data = Scattergl(x=x_axes, y=enc_wall_times, name="Encryption (WALL) Time (s)")
582     dec_data = Scattergl(x=x_axes, y=dec_wall_times, name="Decryption (WALL) Time (s)")
583
584     enc_layout = Layout(
585         showlegend=False,
586         autosize=True,
587         hovermode='closest',
588         title="Encryption Wall Time",
589         xaxis=dict(
590             title="i-th loop",
591             titlefont=dict(
592                 family='Roboto',
593                 size=18,
594                 color='#212121'
595             )),
596         yaxis=dict(
597             title='Time (s)',
598             titlefont=dict(
599                 family='Roboto',
600                 size=18,
601                 color='#212121'
602             )))
603
604     dec_layout = Layout(
605         showlegend=False,
606         autosize=True,
607         hovermode='closest',
608         title="Decryption Wall Time",
609         xaxis=dict(
610             title="i-th loop",
611             titlefont=dict(
612                 family='Roboto',
613                 size=18,
614                 color='#212121'
615             )),
616         yaxis=dict(
617             title='Time (s)',
618             titlefont=dict(
619                 family='Roboto',
620                 size=18,
621                 color='#212121'
622             )))
623
624     return [
625         Figure(data=enc_data, layout=enc_layout).show(renderer="notebook_connected"),
626         Figure(data=dec_data, layout=dec_layout).show(renderer="notebook_connected")]
627
628
629 DEBUG = False
630 set_random_seed(1587)
631 build_datasets("./hl_code", 4, 12)
632 decoding_test(12, 100, DIR="./hl_code", DATADIR="./hl_data")
```

SCRIPT B.4: Python code that implements the algorithms in appendix A.4

Bibliography

[1] Milton Abramowitz and Irene A. Stegun. “Weierstrass Elliptic and Related Functions”. In: *Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables*. Ed. by Milton Abramowitz and Irene A. Stegun. New York: Dover Publications, Inc., 1972. Chap. 18, pp. 627–671. ISBN: 978-0486612720 (cit. on p. 25).

[2] Jeff Achter. *On Computing the Rank of Elliptic Curves*. https://www.math.colostate.edu/~achter/math/brown.pdf. 1992-05 (cit. on p. 26).

[3] Tom Mike Apostol. In: *Modular functions and Dirichlet series in number theory*. 2nd ed. Graduate texts in mathematics 41. New York: Springer Verlang, 1996. Chap. 1.6 - 1.11, pp. 9 –14. ISBN: 978-0387971278 (cit. on p. 25).

[4] Diego F. Aranha et al. *A note on high-security general-purpose elliptic curves*. Cryptology ePrint Archive, Paper 2013/647. https://eprint.iacr.org/2013/647. 2013. URL: https://eprint.iacr.org/2013/647 (cit. on p. 11).

[5] Paulo S. L. M. Barreto et al. “Efficient pairing computation on supersingular Abelian varieties”. In: *Designs, Codes and Cryptography* 42.3 (2007-03), pp. 239–271. ISSN: 1573-7586. DOI: 10.1007/s10623-006-9033-6. URL: https://doi.org/10.1007/s10623-006-9033-6 (cit. on p. 46).

[6] E. Berlekamp, R. McEliece, and H. van Tilborg. “On the inherent intractability of certain coding problems (Corresp.)” In: *IEEE Transactions on Information Theory* 24.3 (1978), pp. 384–386. DOI: 10.1109/TIT.1978.1055873 (cit. on p. 50).

[7] Daniel J. Bernstein and Tanja Lange. “Failures in NIST’s ECC standards”. In: 2015. URL: https://cr.yp.to/newelliptic/nistecc-20160106.pdf (cit. on p. 11).

[8] Daniel J. Bernstein and Tanja Lange. “Faster Addition and Doubling on Elliptic Curves”. In: *Advances in Cryptology – ASIACRYPT 2007*. Ed. by Kaoru Kurosawa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 29–50. ISBN: 978-3-540-76900-2. DOI: 10.1007/978-3-540-76900-2_3 (cit. on pp. 20, 27, 29, 31, 32).

[9] Daniel J. Bernstein and Tanja Lange. “Inverted Edwards Coordinates”. In: *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*. Ed. by Serdar Boztaş and Hsiao-Feng (Francis) Lu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 20–27. ISBN: 978-3-540-77224-8. DOI: 10.1007/978-3-540-77224-8_4 (cit. on pp. 20, 27).

[10] Daniel J Bernstein and Tanja Lange. “Post-quantum cryptography”. In: *Nature* 549.7671 (2017), pp. 188–194. DOI: 10.1007/978-3-540-88702-7 (cit. on p. 3).

[11] Daniel J. Bernstein and Tanja Lange. *SafeCurves: choosing safe curves for elliptic-curve cryptography*. 2013. URL: http://safecurves.cr.yp.to (cit. on p. 11).

[12] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. “Attacking and Defending the McEliece Cryptosystem”. In: *Post-Quantum Cryptography*. Ed. by

Johannes Buchmann and Jintai Ding. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 31–46. ISBN: 978-3-540-88403-3. DOI: 10.1007/978-3-540-88403-3_3 (cit. on p. 50).

[13] Daniel J. Bernstein et al. *Elligator: Elliptic-curve points indistinguishable from uniform random strings.* Cryptology ePrint Archive, Paper 2013/325. 2013. DOI: 10.1145/2508859.2516734. URL: https://eprint.iacr.org/2013/325 (cit. on p. 11).

[14] Daniel J. Bernstein et al. "Twisted Edwards Curves". In: *Progress in Cryptology – AFRICACRYPT 2008.* Ed. by Serge Vaudenay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 389–405. ISBN: 978-3-540-68164-9. DOI: 10.1007/978-3-540-68164-9_26 (cit. on p. 35).

[15] Ian F. Blake, Gadiel Seroussi, and Nigel P. Smart. *Advances in Elliptic Curve Cryptography.* London Mathematical Society Lecture Note Series. Cambridge University Press, 2005. DOI: 10.1017/CBO9780511546570 (cit. on pp. 20, 45, 46).

[16] Fedor A. Bogomolov and Hang Fu. "Division polynomials and intersection of projective torsion points". In: *European Journal of Mathematics* 2.3 (2016-09), pp. 644–660. ISSN: 2199-6768. DOI: 10.1007/s40879-016-0111-7 (cit. on p. 26).

[17] Victor Buchstaber and Dmitry Leykin. "Hyperelliptic Addition Law". In: *Journal of Nonlinear Mathematical Physics* 12.Supplement 1 (2005), pp. 106–123. DOI: 10.2991/jnmp.2005.12.s1.10 (cit. on p. 37).

[18] David Cantor. "Computing in the Jacobian of a hyperelliptic curve". In: *Mathematics of Computation - American Mathematical Society* 48 (1987), pp. 95–101. DOI: 10.1090/S0025-5718-1987-0866101-0 (cit. on pp. 20, 37, 87).

[19] David G. Cantor. "On the analogue of the division polynomials for hyperelliptic curves." In: 1994.447 (1994), pp. 91–146. DOI: doi:10.1515/crll.1994.447.91. URL: https://doi.org/10.1515/crll.1994.447.91 (cit. on p. 26).

[20] Xavier Caruso. "Computations with p-adic numbers". In: *Les cours du CIRM* 5 (2017-01). DOI: 10.5802/ccirm.25 (cit. on p. 55).

[21] J. W. S. Cassels. *LMSST: 24 Lectures on Elliptic Curves.* London Mathematical Society Student Texts. Cambridge University Press, 1991. DOI: 10.1017/CBO9781139172530 (cit. on pp. 20, 55).

[22] Craig Costello and Kristin Lauter. "Group Law Computations on Jacobians of Hyperelliptic Curves". In: *Selected Areas in Cryptography.* Ed. by Ali Miri and Serge Vaudenay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 92–117. ISBN: 978-3-642-28496-0. DOI: 10.1007/978-3-642-28496-0_6 (cit. on p. 37).

[23] Alain Couvreur, Irene Márquez-Corbella, and Ruud Pellikaan. "A polynomial time attack against algebraic geometry code based public key cryptosystems". In: *2014 IEEE International Symposium on Information Theory.* 2014, pp. 1446–1450. DOI: 10.1109/ISIT.2014.6875072 (cit. on p. 50).

[24] Mario A. de Boer. "The generalized Hamming weights of some hyperelliptic codes". In: *Journal of Pure and Applied Algebra* 123.1 (1998), pp. 153–163. ISSN: 0022-4049. DOI: https://doi.org/10.1016/S0022-4049(97)00160-6. URL: https://www.sciencedirect.com/science/article/pii/S0022404997001606 (cit. on pp. 88, 90, 91).

[25] Hang Dinh, Cristopher Moore, and Alexander Russell. "McEliece and Niederreiter Cryptosystems That Resist Quantum Fourier Sampling Attacks". In: *Advances in Cryptology – CRYPTO 2011.* Ed. by Phillip Rogaway. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 761–779. ISBN: 978-3-642-22792-9. DOI: 10.1007/978-3-642-22792-9_43 (cit. on p. 50).

[26] MJ Dworkin et al. "Advanced Encryption Standard (AES); Federal Inf". In: *Process. Stds.(NIST FIPS)* 197 (2001). DOI: 10.6028/NIST.FIPS.197 (cit. on p. 3).

[27] Pál Dömösi, Carolin Hannusch, and Géza Horváth. "A Cryptographic System Based on a New Class of Binary Error-Correcting Codes". In: *Tatra Mountains Mathematical Publications* 73.1 (2019), pp. 83–96. DOI: doi:10.2478/tmmp-2019-0007. URL: https://doi.org/10.2478/tmmp-2019-0007 (cit. on p. 99).

[28] Harold Edwards. "A normal form for elliptic curves". In: *Bulletin of The American Mathematical Society - BULL AMER MATH SOC* 44 (2007-07), pp. 393–423. DOI: 10.1090/S0273-0979-07-01153-6 (cit. on pp. 20, 27).

[29] Giovanni Falcone and Giuseppe Filippone. *Mumford representation and Riemann Roch space of a divisor on a hyperelliptic curve*. 2023. arXiv: 2303.08441 [math.AG] (cit. on pp. 5, 86).

[30] Cédric Faure and Lorenz Minder. "Cryptanalysis of the McEliece cryptosystem over hyperelliptic codes". In: *ACCT 2008* (2008-01), pp. 99–107. URL: http://www.moi.math.bas.bg/acct2008/b17.pdf (cit. on p. 50).

[31] Giuseppe Filippone. "Exp function for Edwards curves over local fields". In: *Advances in Mathematics of Communications* (2023). ISSN: 1930-5346. DOI: 10.3934/amc.2023012. URL: https://www.aimsciences.org/article/doi/10.3934/amc.2023012 (cit. on pp. 5, 108).

[32] Giuseppe Filippone. *Goppa codes over Edwards curves*. 2023. arXiv: 2301.09309 [math.AG] (cit. on pp. 5, 78).

[33] Giuseppe Filippone. *On the Discrete Logarithm Problem for elliptic curves over local fields*. 2023. arXiv: 2304.14150 [math.AG] (cit. on pp. 5, 73).

[34] G. Frey, M. Muller, and H.-G. Ruck. "The Tate pairing and the discrete logarithm applied to elliptic curve cryptosystems". In: *IEEE Transactions on Information Theory* 45.5 (1999), pp. 1717–1719. DOI: 10.1109/18.771254 (cit. on p. 46).

[35] Pierrick Gaudry. "Index calculus for abelian varieties of small dimension and the elliptic curve discrete logarithm problem". In: *Journal of Symbolic Computation* 44.12 (2009). Gröbner Bases in Cryptography, Coding Theory, and Algebraic Combinatorics, pp. 1690–1702. ISSN: 0747-7171. DOI: https://doi.org/10.1016/j.jsc.2008.08.005 (cit. on p. 46).

[36] Victor Gayoso Martínez, Lorena González-Manzano, and Agustín Martín Muñoz. "Secure Elliptic Curves in Cryptography". In: *Computer and Network Security Essentials*. Ed. by Kevin Daimi. Cham: Springer International Publishing, 2018, pp. 283–298. ISBN: 978-3-319-58424-9. DOI: 10.1007/978-3-319-58424-9_16. URL: https://doi.org/10.1007/978-3-319-58424-9_16 (cit. on p. 46).

[37] Carolin Hannusch and Giuseppe Filippone. *Decoding algorithm for HL-codes and performance of the DHH-cryptosystem – a candidate for post-quantum cryptography*. 2023. arXiv: 2303.09820 [cs.CR] (cit. on p. 5).

[38] Carolin Hannusch and Piroska Lakatos. "Construction of self-dual binary 2^2k,2^2k-1,2^k-codes". In: *Algebra and Discrete Mathematics* 21.1 (2016), pp. 59–68. URL: https://admjournal.luguniv.edu.ua/index.php/adm/article/view/25/pdf (cit. on p. 99).

[39] F. Hess, N.P. Smart, and F. Vercauteren. "The Eta Pairing Revisited". In: *IEEE Transactions on Information Theory* 52.10 (2006), pp. 4595–4602. DOI: 10.1109/TIT.2006.881709 (cit. on p. 46).

[40] Toshiro Hiranouchi. "Local torsion primes and the class numbers associated to an elliptic curve over $\mathbb{Q}$." In: *arXiv: Number Theory* (2017). arXiv: 1703.08275v3 (cit. on p. 57).

[41] Huseyin Hisil and Craig Costello. "Jacobian Coordinates on Genus 2 Curves". In: *Journal of Cryptology* 30.2 (2017-04), pp. 572–600. ISSN: 1432-1378. DOI: 10.1007/s00145-016-9227-7 (cit. on p. 37).

[42] Yan Huang et al. "Quantum algorithm for solving hyperelliptic curve discrete logarithm problem". In: *Quantum Information Processing* 19.2 (2020-01), p. 62. ISSN: 1573-1332. DOI: 10.1007/s11128-019-2562-5 (cit. on p. 45).

[43] James E. Humphreys. *Linear Algebraic Groups*. 1st ed. Vol. 21. Graduate Texts in Mathematics. Springer-Verlag New York, 1975, pp. XVI, 248. DOI: 10.1007/978-1-4684-9443-3 (cit. on p. 12).

[44] Dale Husemöller. *Elliptic Curves*. Vol. 111. Graduate Texts in Mathematics. Springer-Verlag New York, 2004, pp. XXII, 490. DOI: 10.1007/b97292 (cit. on pp. 20, 22, 68).

[45] Michael Jacobson, Alfred Menezes, and Andreas Stein. "Solving elliptic curve discrete logarithm problems using Weil descent". In: *J. Ramanujan Math. Soc.* 16.3 (2001), pp. 231–260. ISSN: 0970-1249. URL: https://eprint.iacr.org/2001/041 (cit. on p. 46).

[46] Heeralal Janwa and Oscar Moreno. "McEliece Public Key Cryptosystems Using Algebraic-Geometric Codes". In: *Designs, Codes and Cryptography* 8.3 (1996-06), pp. 293–307. ISSN: 1573-7586. DOI: 10.1023/A:1027351723034 (cit. on p. 50).

[47] Anthony W. Knapp. *Elliptic Curves. (MN-40), Volume 40*. Princeton University Press, 1992. DOI: 10.2307/j.ctv346st5. (Visited on 01/16/2023) (cit. on p. 20).

[48] Neal Koblitz. "Elliptic curve cryptosystems". In: *Mathematics of Computation - American Mathematical Society* 48 (1987), pp. 203–209. DOI: 10.1090/S0025-5718-1987-0866109-5 (cit. on p. 45).

[49] Neal Koblitz. "Hyperelliptic cryptosystems". In: *Journal of Cryptology* 1.3 (1989-10), pp. 139–150. ISSN: 1432-1378. DOI: 10.1007/BF02252872. URL: https://doi.org/10.1007/BF02252872 (cit. on pp. 20, 38, 87).

[50] Neal Koblitz, Alfred Menezes, and Scott Vanstone. "The State of Elliptic Curve Cryptography". In: *Des. Codes Cryptography* 19 (2000-03), pp. 173–193. DOI: 10.1023/A:1008354106356 (cit. on p. 46).

[51] Michiel Kosters and René Pannekoek. "On the structure of elliptic curves over finite extensions of $\mathbb{Q}_p$ with additive reduction". In: (2017-03). DOI: 10.48550/ARXIV.1703.07888 (cit. on p. 57).

[52] Tanja Lange. "Edwards Curves". In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Boston, MA: Springer US, 2011, pp. 380–382. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_243 (cit. on p. 20).

[53] Tanja Lange. "Formulae for Arithmetic on Genus 2 Hyperelliptic Curves". In: *Applicable Algebra in Engineering, Communication and Computing* 15.5 (2005-02), pp. 295–328. ISSN: 1432-0622. DOI: 10.1007/s00200-004-0154-8 (cit. on pp. 37, 39).

[54] Frank Leitenberger. "About the group law for the Jacobi variety of a hyperelliptic curve." eng. In: *Beiträge zur Algebra und Geometrie* 46.1 (2005), pp. 125–130. URL: http://eudml.org/doc/229233 (cit. on p. 37).

[55] Franck Leprévost et al. "Generating anomalous elliptic curves". In: *Information Processing Letters* 93.5 (2005), pp. 225–230. ISSN: 0020-0190. DOI: https://doi.

org/10.1016/j.ipl.2004.11.008. URL: https://www.sciencedirect.com/science/article/pii/S0020019004003527 (cit. on p. 46).

[56] P. Lockhart. “On the Discriminant of a Hyperelliptic Curve”. In: *Transactions of the American Mathematical Society* 342.2 (1994), pp. 729–752. ISSN: 00029947. DOI: 10.2307/2154650. URL: http://www.jstor.org/stable/2154650 (visited on 01/17/2023) (cit. on p. 86).

[57] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes*. 2nd. North-holland Publishing Company, 1978 (cit. on pp. 94, 96).

[58] R. J. McEliece. “A Public-Key Cryptosystem Based On Algebraic Coding Theory”. In: *Deep Space Network Progress Report* 44 (1978-01), pp. 114–116. URL: https://ui.adsabs.harvard.edu/abs/1978DSNPR..44..114M (cit. on pp. 50, 100).

[59] A.J. Menezes, T. Okamoto, and S.A. Vanstone. “Reducing elliptic curve logarithms to logarithms in a finite field”. In: *IEEE Transactions on Information Theory* 39.5 (1993), pp. 1639–1646. DOI: 10.1109/18.259647 (cit. on p. 46).

[60] Victor S. Miller. “Use of Elliptic Curves in Cryptography”. In: *Advances in Cryptology — CRYPTO ’85 Proceedings*. Ed. by Hugh C. Williams. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 417–426. ISBN: 978-3-540-39799-1 (cit. on p. 45).

[61] P. L. Montgomery. “Speeding the Pollard and elliptic curve methods of factorization”. In: *Mathematics of Computation* 48 (1987), pp. 243–264 (cit. on pp. 20, 27).

[62] D. E. Muller. “Application of Boolean algebra to switching circuit design and to error detection”. In: *Transactions of the I.R.E. Professional Group on Electronic Computers* EC-3.3 (1954), pp. 6–12. DOI: 10.1109/IREPGELC.1954.6499441 (cit. on p. 94).

[63] David Mumford. *Abelian Varieties*. 2nd ed. Vol. 7. Tata Institute of Fundamental Research Studies in Mathematics. https://bookstore.ams.org/tifr-13. London: Oxford University Press: Tata Institute of Fundamental Research, 1974, p. 242 (cit. on p. 37).

[64] I. S. Reed. “A class of multiple-error-correcting codes and the decoding scheme”. In: *Transactions of the IRE Professional Group on Information Theory* 4.4 (1954), pp. 38–49. DOI: 10.1109/TIT.1954.1057465 (cit. on pp. 94, 103).

[65] R. L. Rivest, A. Shamir, and L. Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. New York, NY, USA, 1978-02. DOI: 10.1145/359340.359342. URL: https://doi.org/10.1145/359340.359342 (cit. on p. 3).

[66] Takakazu Satoh and Kiyomichi Araki. *Fermat Quotients and the Polynomial Time Discrete Log Algorithm for Anomalous Elliptic Curves*. 1998-06. DOI: 10.14992/00009878. URL: https://doi.org/10.14992/00009878 (cit. on p. 46).

[67] J. Scholten and F. Vercauteren. “An Introduction to Elliptic and Hyperelliptic Curve Cryptography and the NTRU Cryptosystem”. In: https://www.cse.iitk.ac.in/users/nitin/courses/WS2010-ref4.pdf. 2004 (cit. on p. 46).

[68] I. A. Semaev. “Evaluation of Discrete Logarithms in a Group of P-Torsion Points of an Elliptic Curve in Characteristic p”. In: *Math. Comput.* 67.221 (1998-01), 353–356. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-98-00887-4. URL: https://doi.org/10.1090/S0025-5718-98-00887-4 (cit. on p. 46).

[69] Jean-Pierre Serre. *Local fields*. Vol. 67. Graduate Texts in Mathematics. Translated from the French by Marvin Jay Greenberg. New York: Springer-Verlag, 1979, pp. viii+241. ISBN: 0-387-90424-7. DOI: https://doi.org/10.1007/978-1-4757-5673-9 (cit. on p. 54).

[70] Jean-Pierre Serre. “Singular Algebraic Curves”. In: *Algebraic Groups and Class Fields*. New York, NY: Springer New York, 1988, pp. 58–73. ISBN: 978-1-4612-1035-1. DOI: 10.1007/978-1-4612-1035-1_4 (cit. on p. 81).

[71] P.W. Shor. “Algorithms for quantum computation: discrete logarithms and factoring”. In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700 (cit. on p. 4).

[72] Joseph H. Silverman. “Lifting and Elliptic Curve Discrete Logarithms”. In: *Selected Areas in Cryptography*. Ed. by Roberto Maria Avanzi, Liam Keliher, and Francesco Sica. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 82–102. ISBN: 978-3-642-04159-4. DOI: 10.1007/978-3-642-04159-4_6. URL: https://doi.org/10.1007/978-3-642-04159-4_6 (cit. on p. 73).

[73] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*. 2nd ed. Vol. 106. Graduate Texts in Mathematics. New York: Springer Verlang, 2009. ISBN: 978-0-387-09493-9. DOI: 10.1007/978-0-387-09494-6 (cit. on pp. 20, 22, 26, 46, 57).

[74] N. P. Smart. “The Discrete Logarithm Problem on Elliptic Curves of Trace One”. In: *Journal of Cryptology* 12.3 (1999-06), pp. 193–196. ISSN: 1432-1378. DOI: 10.1007/s001459900052. URL: https://doi.org/10.1007/s001459900052 (cit. on p. 46).

[75] ChunMing Tang, MaoZhi Xu, and YanFeng Qi. “Cryptography on twisted Edwards curves over local fields”. In: *Science China Information Sciences* 58.1 (2015-01), pp. 1–15. ISSN: 1869-1919. DOI: 10.1007/s11432-014-5155-z. URL: https://doi.org/10.1007/s11432-014-5155-z (cit. on p. 73).

[76] Rosa Winter. *Elliptic curves over* $\mathbb{Q}_p$. Bachelor thesis, Mathematisch Instituut, Universiteit Leiden. https://www.universiteitleiden.nl/binaries/content/assets/science/mi/scripties/bachwinter.pdf. 2011-08 (cit. on p. 57).

[77] MaoZhi Xu et al. “Cryptography on elliptic curves over p-adic number fields”. In: *Science in China Series F: Information Sciences* 51.3 (2008-03), pp. 258–272. ISSN: 1862-2836. DOI: 10.1007/s11432-008-0014-4. URL: https://doi.org/10.1007/s11432-008-0014-4 (cit. on p. 73).

[78] Zhi Hong Yue and Mao Zhi Xu. “Hierarchical Management Scheme by Local Fields”. In: *Acta Mathematica Sinica* 27.1 (2010-12), pp. 155–168. ISSN: 1439-8516. DOI: https://doi.org/10.1007/s10114-011-9110-2. URL: https://actamath.cjoe.ac.cn/Jwk_sxxb_en/EN/10.1007/s10114-011-9110-2 (cit. on p. 73).

Index

j-invariant, 21
p-adic evaluation, 56
p-adic integers, 56
p-adic metric, 55
p-adic norm, 56
p-adic numbers, 55
p-adic order, 56

abelian group, 22
abelian variety, 20
Advanced Encryption Standard, 3
AES, 3
affine algebraic curve, 17
affine algebraic plane curve, 17
affine algebraic set, 17
affine space, 13
affine variety, 17
AG Goppa code, 52, 82
algebraic group, 20
algebraic normal form, 96
algebraic variety, 17
algebraic-geometric Goppa codes, 50
algebraically closed field, 17
ANF, 96
anomalous curve, 46
Archimedean, 54
asymmetric-key, 7
authenticated, 5

Baby-step giant-step, 26
backdoor, 11
binary Goppa codes, 50
birationally equivalent, 33
block ciphers, 8
Boolean algebra, 94
Boolean function space, 95
Boolean functions, 94
brute force attack, 9

canonical forms, 30
canonical height, 60, 61
Cantor-Koblitz algorithm, 37
chord-and-tangent rule, 22
ciphertext, 4, 7
ciphertext-only attack, 4
codeword, 49
Coding theory, 48
cofactor, 46
complement-free, 99
complete, 54
completion, 18
complex projective plane, 24
composition, 38
cryptanalysis, 11
cryptographic attack, 9
Cryptography, 3
Curve25519, 27, 66

decryption method, 7
degenerate cubic, 40
degree, 18
dehomogenization, 15
dehomogenize, 13, 15
Diffie-Hellman protocol, 46
discrete logarithm problem, 45
discriminant, 21
divisor, 18
DLP, 45

ECC, 45
ECDLP, 45
Ed25519, 27
EdDSA, 27, 35
Edwards curve, 27
Edwards-curve Digital Signature Algorithm, 35
effective divisor, 19
ElGamal encryption, 47
elliptic curve, 20
Elliptic Curve Cryptography, 45
elliptic curve discrete logarithm problem, 45
elliptic invariants, 24
encryption method, 7
Enigma machine, 3

error-correction codes, 48
errors, 49
exhaustive search, 9

factorization of the function, 15
finitely generated abelian group, 25
floating point operations per second, 10
FLOPS, 10
free abelian group, 18
free group, 18

Gauss-Jordan method, 49
Gaussian elimination, 83
generalized Weierstrass form, 21
generator matrix, 49
genus, 20
good reduction, 21
group law, 20

half-periods, 24
Hamming distance, 49
Hasse's bound, 26
Hasse's theorem, 26
HCC, 46
HCDLP, 46
HECDLP, 45
height, 60
homogeneous polynomial, 15
homogeneous rational function, 15
homogenization, 15
homogenize, 15
hyperelliptic curve, 36
hyperelliptic curve cryptography, 46
hyperelliptic curve discrete logarithm problem, 45
Hyperelliptic Curve DLP, 46

imaginary hyperelliptic curves, 36
improper point, 14
Index Calculus, 46
inverse limit, 56
irreducible, 17

Jacobian, 19

key agreement, 45
key expansion, 7

lattice, 24
least significant bit, 103
linear code, 49
local field, 54
logarithmic height, 60, 61
LSB, 103

maximal, 99
Maximum Distance Separable, 78
Maximum Distance Separable code, 50
MDS, 78, 84
MDS code, 50
million instructions per second, 10
minimum (Hamming) distance, 49
minterm, 97
MIPS, 10
mono-alphabetic, 4
Montgomery curve, 27
Mordell-Weil theorem, 25
morphism, 20
most significant bit, 137
MSB, 137
multiplicity, 18
Mumford representation, 37

National Institute of Standards and Technology, 3, 11
neutral element, 28
NIST, 3, 11
non-Archimedean, 54
non-singular, 20
non-smooth, 21, 27
non-supersingular, 22
normalized valuation, 55
NP-hard, 50
Néron–Tate height, 61

one-time pad, 4
order, 23
ordinary elliptic curves, 36
ordinary singular points, 28
OTP, 4

parity-check matrix, 49
perfect secrecy, 4
perfect secrecy cipher, 4
plaintext, 4, 7
plaintext attacks, 9
point at infinity, 14
pole, 16
Pollard's rho, 45, 46
poly-alphabetic, 4
post-quantum cryptography, 50
principal divisor, 19
principle of superposition, 95
private-key, 7

PRNG, 5
projective algebraic curve, 17
projective algebraic plane curve, 17
projective algebraic set, 17
projective line, 14
projective plane, 14
projective point, 13
projective resolution, 33
projective space, 13
projective variety, 17
proper point, 14
pseudo-random number generators, 5
public-key, 7
public-key encryption algorithm, 8
Puiseux series, 61

quantum computers, 3
qubits, 3

randomization, 50
rank, 25
real hyperelliptic curves, 36
reduced divisor, 36
reduction, 38
Reed-Muller codes, 94
Reed-Muller expansion, 96
regular function, 20
residue field, 54, 55
restriction, 18
ring of integers, 55
RM-codes, 94
RSA, 3

Schoof's algorithm, 26
Schoof–Elkies–Atkin's algorithm, 26
scytale, 3
secret key, 47
secure, 7, 10
security level, 10
security strength, 10
security tests, 8
semi-reduced divisor, 39
shared key, 7
shared-key encryption algorithm, 8
Shor's algorithm, 4, 50
short Weierstrass form, 21
singular, 21
smooth, 20
standard form, 49, 82
statistical analysis, 4
stream ciphers, 8
supersingular, 22
support, 18
symmetric-key, 7

torsion subgroup, 25
torsion-free generating set, 26
torus, 24
twisted Edwards curve, 34

uniformizer, 55
units, 55

Vernam cipher, 4

Weierstrass uniformizing map, 25
weight, 49

X25519, 27

zero, 16
zero degree divisor, 19
zero set, 17