SOFTWARE IMPLEMENTATIONS AND APPLICATIONS OF ELLIPTIC CURVE CRYPTOGRAPHY

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Cyber Security

by

KIRILL KULTINOV
B.S.C.S., Wright State University, 2017

2019
Wright State University

Wright State University
GRADUATE SCHOOL

May 1, 2019

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Kirill Kultinov ENTITLED Software Implementations and Applications of Elliptic Curve Cryptography BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Cyber Security.

_____

Meilin Liu, Ph.D.
Thesis Director

_____

Mateen Rizki, Ph.D.
Chair, Department of Computer Science
and Engineering

Committee on
Final Examination

_____

Meilin Liu, Ph.D.

_____

Junjie Zhang, Ph.D.

_____

Keke Chen, Ph.D.

_____

Barry Milligan, Ph.D.
Interim Dean of the Graduate School

ABSTRACT

KULTINOV, KIRILL. M.S.C.S., Department of Computer Science and Engineering, Wright State University, 2019. *SOFTWARE IMPLEMENTATIONS AND APPLICATIONS OF ELLIPTIC CURVE CRYPTOGRAPHY*.

Elliptic Curve Cryptography (ECC) is a public-key cryptography system. Elliptic Curve Cryptography (ECC) can achieve the same level of security as the public-key cryptography system, RSA, with a much smaller key size. It is a promising public key cryptography system with regard to time efficiency and resource utilization.

This thesis focuses on the software implementations of ECC over finite field GF(p) with two distinct implementations of the Big Integer classes using character arrays, and bit sets in C++ programming language. Our implementation works on the ECC curves of the form $y^2 = x^3 + ax + b \ (mod\ p)$. The point addition operation and the scalar multiplication are implemented on a real SEC (Standards for Efficient Cryptography) ECC curve over a prime field with two different implementations. The Elliptic Curve Diffie-Hellman key exchange, the ElGamal encryption/decryption system, and the Elliptic Curve Digital Signature Algorithm (ECDSA) on a real SEC ECC curve with two different implementations of the big integer classes are tested, and validated. The performances of the two different implementations are compared and analyzed.

Contents

List of Figures

List of Tables

Acknowledgment

I want to thank my advisor Dr. Meilin Liu teaching cryptography class at Wright State University, which is one of my favorite areas of study. I also thank Dr. Meilin Liu for her patient and accurate advising throughout the course of this project.

I would also like to thank my family and, especially, my parents for their unlimited support, help, and encouragement.

Finally, I would like to thank my friends who motivated me during the course of my graduate study.

Chapter 1: Introduction

1.1 Motivation

Data security is very crucial for almost any system nowadays [19]. Cryptography is a mathematical tool that is used in the software and hardware system to provide security services, and defend data and information in storage and in transmission against unauthorized access or tampering, facilitate key exchange between two communication parties. Cryptography plays an important role in many applications.

In the early days of cryptography, the symmetric key cryptographic systems [7] are used to encrypt and decrypt messages. Public-key cryptography systems [2], Diffie-Hellman key exchange system, and RSA, are developed in 1976, 1977 respectively, which are more secure compared to the symmetric encryption systems, since public-key systems are based on the number theory, and are asymmetric, involving the use of two separate keys, the public key, and the private key.

Nowadays, public key cryptography is very crucial because data integrity and confidentially depends on it. Public key cryptography has to provide forward secrecy, which means that information secure in present must be also secure in the future [14]. RSA is the most popular public key cryptography algorithm. Its security is based on the difficulty of factoring large numbers [10]. As computational capabilities of the computers increase, RSA is not able to provide sufficient forward secrecy without exponentially increasing key sizes. Because of the computational overhead of RSA systems with large key sizes,

Elliptic curve cryptography (ECC), a public-key cryptography system based on algebra, became more and more popular for developing Public-key based cryptography system. Elliptic Curve Cryptography (ECC) can achieve the same level of security as the public-key cryptography system, RSA, with a much smaller key size. The key size comparisons to achieve the same level of security by RSA and ECC are shown in table 1.1. ECC is a promising public key cryptography system with regard to time efficiency and resource utilization. ECC was developed in 1985 by Neal Koblitz and Victor Miller and has come into use since 2005 [8]. The logic of ECC is completely different from any other cryptographic algorithm and depends on the difficulties of solving discrete logarithm problem over point additions and multiplications on elliptic curves. ECC is growing its popularity and has been applied in many systems and protocols.

Table 1.1: Comparable key sizes in Terms of Computational Effort for Cryptanalysis.

Symmetric key size in bits	**RSA key size in bits**	**ECC key size in bits**
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

One of the most popular applications of ECC is the key exchange between two communication parties. ECC can be used in a variant of Diffie-Hellman key exchange called Elliptic Curve Diffie-Hellman. 97% of popular websites have some support of elliptic curve Diffie-Hellman. More precisely, these websites are using Elliptic Curve Diffie-hellman Ephemeral Elliptic Curve Digital Signature Algorithm (EDHE_ECDSA) for key exchange while establishing HTTPS connections [8]. ECDSA is widely applied in blockchain technology [13]. ECC can also be applied in DNSSEC protocol, which is a secured version of DNS that protects DNS servers from DDoS attacks [8]. It is possible to implement DNSSEC using RSA as a signature algorithm. However, it makes servers vulnerable to different possible attacks [8]. Instead, ECDSA (Elliptic Curve Digital Signature Algo-

rithm) algorithm can be applied on DNS servers to protect them from amplification attacks without any packet fragmentation and other complications [23].

Mobile devices are a part of every aspect of people's lives. These devices have networking capabilities and attackers can exploit different vulnerabilities. Securing mobile devices is very important. However, public key cryptography algorithms are too computationally expensive due to computing capabilities and memory constraints of these devices. Fortunately, ECC is suitable to be used in two-factor authentication. For example, a lightweight protocol proposed by a team of researchers uses elliptic curves and is resistant to many different attacks such as man-in-the-middle attacks and replay attacks [15]. In addition to that, ECC can also used in one time password (OTP) scheme based on Lamport's OTP algorithm and for IoT devices utilizing ECDH [8]. Finally, ECC is used in protecting smart grids and securing communication channels of autonomous cars [6, 9].

This thesis focuses on the software implementation of ECC over finite field GF(p) using character arrays, and bit sets in C++ programming language. Our implementation works on the ECC curves of the form $y^2 = x^3 + ax + b \ (mod\ p)$. The point addition operation and the scalar multiplication are implemented on a real SEC (Standards for Efficient Cryptography) ECC curve over a prime field with two different implementations. The ElGamal encryption/decryption system and Elliptic Curve Digital Signature Algorithm (ECDSA) on a real SEC ECC curve implemented with two different implementations are implemented, and validated. The performance of the two different implementations are compared and analyzed.

1.2 Overview

Chapter 1 illustrates why it is important to implement ECC public-key systems on real NIST (National Institute of Standard and Technology) ECC curves over a prime field using character arrays, and bit sets in C++ programming language, and also provides the outline

and the organization of the thesis.

Chapter 2 provides background information used in this thesis. It includes basic number theory concepts, including modular arithmetic and properties of groups, rings, fields. Finally,The ECC crypto-system is illustrated in details, and point addition, point doubling operations are introduced.

Chapter 3 describes the detailed implementation of the ECC public-key systems on real NIST (National Institute of Standard and Technology) ECC curves over a prime field using two distinct implementations of the Big Integer objects, i.e., character arrays, and bit sets. We illustrate how each component of ECC system is designed and also introduced the optimization techniques we used to make our implementations more efficient.

Chapter 4 reports the experimental results of our implementations of ECC in C++ on a Linux Ubuntu OS. We also compare the timing performances of the basic operations, point addition, point doubling operations, of our implementations of Big Integer objects used in the ECC systems with two distinct implementations, the character arrays, and the bit sets. In addition, we implement and test the Diffie-Hellman key exchange, the ElGamal encryption/decryption system, and the Elliptic Curve Digital Signature Algorithm (ECDSA) on a real SEC ECC curve.

Chapter 5 summarizes our thesis work and discusses the future work of this thesis.

Chapter 2: Background

In this chapter, we will introduce basic concepts used in this master thesis.

2.1 Mathematical Background

Number theory and algebra play a very important role in cryptography [20]. Cryptography algorithms are based on number theory concepts that allows making these algorithms secure against different attacks. Logic of ECC is different from other public key cryptography algorithms and can be challenging to understand. In this chapter, we will introduce basic concepts including modular operations, groups, finite fields, ECC, point addition, and scalar multiplication on ECC curve, and the applications of ECC, including ElGamal systems.

2.1.1 Modular Arithmetic

Given any positive integer n, and any nonnegative integer a, dividing a by n will give us an integer quotient q, and a remainder r that meet the following condition [21]:

$$a = qn + r \quad 0 \leq r < n; q = \lfloor a/n \rfloor \tag{2.1}$$

Considering assumptions made above, the remainder of an arithmetic operation, where a is divided by n, is denoted by a mod n and the integer n is called modulus. For any integer

a, the division algorithm illustrated in equation 2.1 can be written as:

$$a = \lfloor a/n \rfloor \times n + (a \bmod n) \tag{2.2}$$

Examples of mod operation is shown below:

$$13 \bmod 7 = 6 \quad -13 \bmod 7 = 1$$

Congruent Modulo

Properties of congruent modulo are used in cryptography and play an important role. Congruence is denoted by $\equiv$ symbol and we say that two integers a and b are congruent modulo n if the following condition is satisfied [21]:

$$(a \bmod n) = (b \bmod n) \tag{2.3}$$

From the equation 2.3, we note that if $a \equiv 0 \pmod{n}$, then n is a divisor of a or $n \mid a$. In addition to that, congruences have the following properties:

1. $a \equiv b \pmod{n}$ if $n \mid (a - b)$
2. $a \equiv b \pmod{n} \implies b \equiv a \pmod{n}$
3. $a \equiv b \pmod{n} \wedge b \equiv c \pmod{n} \implies a \equiv c \pmod{n}$

Properties of Modular Arithmetic

Equation 2.1 shows that $\pmod{n}$ operation maps all integers into a set of values $\mathbb{Z}_n = \{0, 1, ..., n-1\}$. Indeed, if we divide a by n, the remainder satisfies the following condition $0 \leq r < n$. It means that we can perform arithmetic operations within the defined set [21], which are called modular arithmetic and have the following properties:

1. $[(a \bmod n) + (b \bmod n)] \bmod n = (a + b) \bmod n$

2. $[(a \bmod n) - (b \bmod n)] \bmod n = (a - b) \bmod n$

3. $[(a \bmod n) \times (b \bmod n)] \bmod n = (a \times b) \bmod n$

4. $a^k \equiv b^k \pmod{n}$ if k is an integer and $k \geq 0$

2.1.2 Multiplicative Inverse

Multiplicative inverse is an important modular operation used in many cryptography algorithms [24]. The existence of a multiplicative inverse is one of the axioms that makes a set of elements a field.

Suppose a and x are elements of $\mathbb{Z}_p$ and p is a prime number, then x is a multiplicative inverse of a, i.e., $x = 1/a \pmod{p}$ if the following equation is satisfied.

$$ax \pmod{p} = 1 \tag{2.4}$$

The Extended Euclidean Algorithm

The greatest common divisor of a and b is the largest integer that divides both a and b with a zero remainder. The greatest common divisor of two integers is denoted by $gcd(a, b)$ [21]. The Euclidean algorithm can be used to compute the greatest common divisor of two integers. The algorithm states that for any nonnegative integer a and a positive integer b, we can find the greatest common divisor of these two integers using the following equation:

$$gcd(a, b) = gcd(b, a \bmod b) \;\; if \;\; a > b \tag{2.5}$$

The Euclidean Algorithm can be used to compute the greatest common divisor of two integers. However, it is not suitable to dealing with large numbers. In cryptography, the Extended Euclidean Algorithm is used instead. The algorithm not only computes the gcd

of two integers a and b, but also calculates two additional integers x and y that satisfy the following equation:

$$ax + by = gcd(a, b) \tag{2.6}$$

In the equation above, the integer x is a modular multiplicative inverse of a modulo b and y is a modular multiplicative inverse of b modulo a.

Cryptography algorithms primarily deal with elements of finite fields $GF(p)$. In this case, the Extended Euclidean Algorithm can also be used for computing multiplicative inverses of a and b. If a and b are relatively prime ($gcd(a, b) = 1$), then b has a multiplicative inverse modulo a. Also, if $gcd(a, b) = 1$, we have $ax + by = 1$ [21]. Now, applying properties of modular arithmetic, the equation 2.6 becomes:

$$[(ax \bmod a) + (by \bmod a)] \bmod a = 1 \bmod a$$

$$by \bmod a = 1$$

From this, if $by \bmod a = 1$, then we have $y = b^{-1} \bmod a$, which is identical to equation 2.4, where y is the multiplicative inverse of b. Thus, the Extended Euclidean algorithm can be used to compute multiplicative inverse efficiently.

2.1.3 Group, Ring, Field

Group

A group, denoted by $\{G, \bullet\}$, is a set of elements with a binary operator $\bullet$, in which the relationship among all elements in the set satisfies the following properties:

1. *Closure:* If a and $b \in G$, then $a \bullet b \in G$

2. *Associative:* $a \bullet (b \bullet c) = (a \bullet b) \bullet c$ for all $a, b, c \in G$

3. *Identity element:* There is an element e in G such that $a \bullet e = e \bullet a = a$ for all $a \in G$

4. *Inverse element:* For every $a \in G$, there is an element $a' \in G$ such that $a \bullet a' = a' \bullet a = e$

Groups can be finite and infinite. Finite groups have a finite number of elements, while infinite groups have infinite number of elements. In addition to that, a group is said to be abelian [21] if the following property exists:

- *Commutative:* $a \bullet b = b \bullet a$ for all a and $b \in G$

Ring

A ring is a set of elements, denoted by $\{R, +, \times\}$, with two binary operator $+$ and $\times$, such that for all $a, b, c \in R$ the following axioms are satisfied:

1. *Abelian group:* R is an abelian group with respect to addition with the identity element denoted by 0 and the inverse of a denoted by $-a$.

2. *Closure under multiplication:* if a and $b \in R$, then $ab \in R$

3. *Associativity of multiplication:* $a\,(bc) = (ab)\,c$ for every $a, b, c \in R$

4. *Distributive laws:* In R, multiplication is distributive with respect to addition as shown in the following:

    - $a\,(b + c) = ab + ac$ for all $a, b, c \in R$
    - $(a + b)\,c = ac + bc$ for all $a, b, c \in R$

A ring can also be commutative if the following condition is satisfied:

- *Commutativity of multiplication:* $ac = ca$ for all $a, c \in R$

In order to understand properties of a field, it is necessary to describe an integral domain. An integral domain is a commutative ring R, under which two addition properties are satisfied:

1. *Multiplicative identity:* an element 1 in R exists that satisfies $a1 = 1a = a$ for all $a \in R$

2. *No 0 divisors:* if $a, b \in R \wedge ab = 0 \implies a = 0 \vee b = 0$

Field

A field is a set of elements, denoted by $\{F, +, \times\}$, with two binary operators $+$ and $\times$, in which for all $a, b, c \in F$, the following conditions are satisfied:

1. *Integral domain:* F is an integral domain. More precisely, all axioms for an abelian group are satisfied and properties of an integral domain are satisfied.

2. *Multiplicative inverse:* For every $a \in F$, there is an element a^{-1} such that $aa^{-1} = (a^{-1})\, a = 1$

Finite Fields $GF(p)$

Finite fields of order p^n (order is the number of elements in a given field) are called Galois Fields and denoted by $GF(p^n)$. In cryptography, finite fields of the form $GF(p)$ are preferred due to security measures and their arithmetic properties.

For a given prime p, Galois Field of order p is a set of integers $\mathbb{Z}_p = \{0, 1, ..., p-1\}$ along with operations modulo p. In addition to properties of a field described earlier, $GF(p)$ has an additional unique property:

- *Multiplicative inverse* w^{-1}: for every $w \in \mathbb{Z}_p \wedge w \neq 0$ there is a $z \in \mathbb{Z}_p$ that satisfies $w \times z \equiv 1 \pmod{p}$

As mentioned in sections 2.1.2, we can use the Extended Euclidean algorithm in order to calculate w^{-1} when dealing with large numbers during the implementation process of ECC.

2.2 ECC Concepts

Elliptic Curve cryptography are based on equations of elliptic curves and computations over the points that belong to a given curve. In this section, we introduce concepts of ECC used in cryptography. First, we explain properties and operations of Elliptic curves over real numbers because important details can be easily shown with the use of geometry. Next, we describe elliptic curves over $GF(p)$ that are used in ECC.

2.2.1 Main idea of ECC

Imagine a large but finite set E that consists of points on the plane (x_i, y_i) drawn from the elliptic curve. We can define a group addition operator, denoted by $+$, for given two points P and Q in the set E. The group operator allows us to calculate a third point $R \in E$ such that $P + Q = R$.

Given a point $G \in E$, we are interested in calculating $G + G + G + ... + G$ using the group operator. More precisely, having an arbitrary number $k \notin G$, we use notation $k \times G$ to represent a repeated addition of point G to itself k times (operator $+$ invoked $k-1$ times). The idea of ECC is that it is hard to recover k from $k \times G$ because an attacker needs to try all possible combinations of repeated addition: $G+G$, $G+G+G$, $G+G+G+...+G$ [11]. This problem is called discrete logarithm problem and is the base for the security of ECC algorithm.

2.2.2 ECC Over Real Numbers

Elliptic curves do not have anything in common with ellipses [11]. Instead, they are described using cubic equations that are used for calculating the circumference of an ellipse [21]. Generally, those curves have the form known as Weiestrass equation

$$y^2 + axy + by = x^3 + cx^2 + dx + e \tag{2.7}$$

where parameters a, b, c, d are real numbers. For cryptography purposes, the equation of the following form is used instead:

$$y^2 = x^3 + ax + b \tag{2.8}$$

The equation above is for a field of real numbers, where coefficients a and b and the variables x and y belong to the field of real numbers. Moreover, examples and concepts provided in this section use fields of characteristic 0. Characteristic of a field is the smallest number of additions of multiplicative identity element to itself that gives the additive identity element [3]. Characteristic 0 means that it does not matter how many times we add the multiplicative identity to itself, we never get the additive identity element.

Figure 2.1 and Figure 2.2 show examples of elliptic curves drawn from equation with different parameters a and b in equation 2.8:

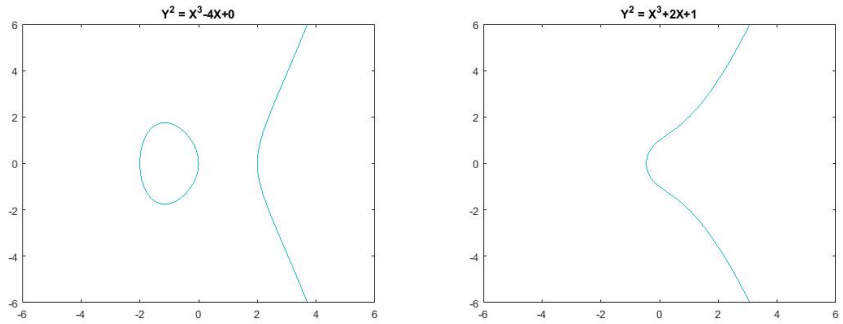

Figure 2.1: Examples of Elliptic Curves

Elliptic curves can be singular or non-singular. Figure 2.1 shows an example non-singular elliptic curve. You can see that the curve is smooth. Non-smooth curves are singular as shown in Figure 2.2. Smooth curves satisfy the discriminant condition of a polynomial $f(x) = x^3 + ax + b$:

$$4a^3 + 27b^2 \neq 0 \tag{2.9}$$

The elliptic curve described in equation 2.8 is a cubic polynomial. It means it has three distinct roots r_1, r_2, and r_3. The discriminant is described by a formula:

$$D_3 = \prod_{i<j}^{3} (r_i - r_j)^2 \tag{2.10}$$

which is equivalent to:

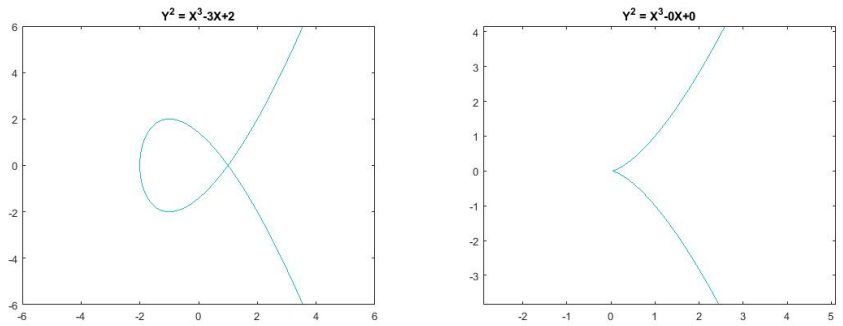

Figure 2.2: Examples of Singular Curves

$$D_3 = (r_1 - r_2)^2 (r_1 - r_3)^2 (r_2 - r_3)^2 \tag{2.11}$$

If the discriminant is zero, then two or more roots have coalesced, which makes the curve non-smooth [11]. Singular curves are not suitable for cryptography purposes because they are easy to crack. Thus, we are only interested in non-singular curves, which means curves used in ECC algorithms must have non-zero discriminant.

Group operator for ECC

For an elliptic curve defined in the equation 2.8, we have a set of points that belong to the curve denoted by a set $E(a, b)$ along with a special distinguished point at infinity, denoted as O. $E(a, b)$ is abelian group [11, 21] under a special addition operator, denoted by $+$. The addition operator does not have anything in common with a traditional addition operator used in algebra and is described as follows.

Point addition

Suppose we want to add a point P to another point Q, we take the following steps:

1. Draw a straight line joining points P and Q
2. Find the intersection of the joining line and the elliptic curve to get a third point R
3. Reflect the point R along the x-axis. It gives us a point denoted by $-R$. The reflection is possible because the equation 2.8 can be re-written as $y = \sqrt{x^3 + ax + b}$, which means the curve is symmetrical with respect to the x-axis.

Point addition on a curve over real numbers can be shown geometrically (See Figure 2.3).

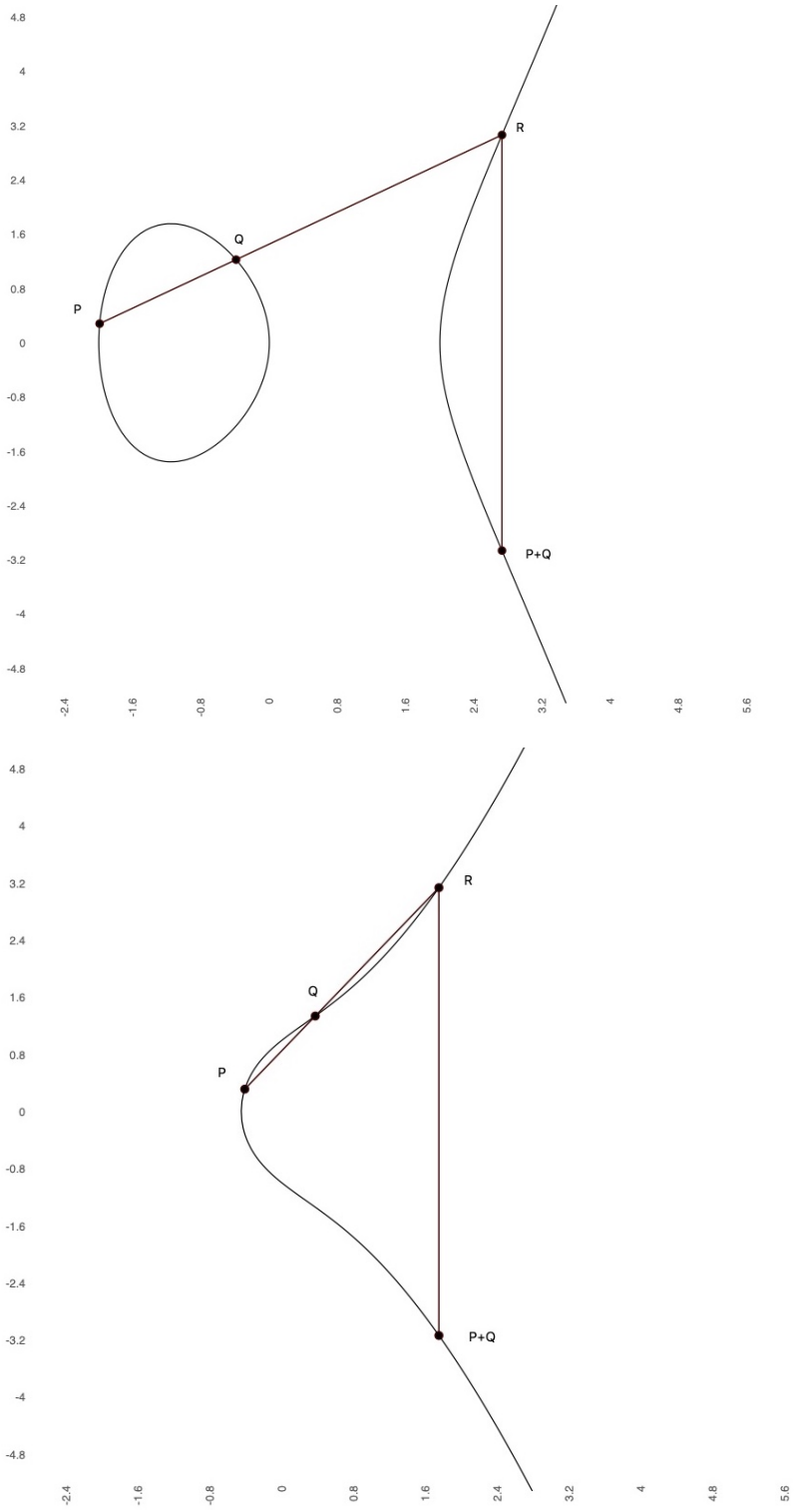

Figure 2.3: Point Addition on Elliptic Curves

Thus, adding two points gives us $P + Q = -R$. An exception to this procedure is when the intersection of the joining line of points P and Q with the elliptic curve does not exist. In this case, we say it is at the distinguished point at infinity. It is only possible when the line joining P and Q is parallel to y-axis. Point at infinity allows us to stipulate the following properties:

- $P + O = P$. Adding point P to a point at infinity will require us drawing a line parallel to y-axis. We find another point crossing the curve, which is the mirror reflection of P with regard to x-axis. Thus, reflecting that other point $-P$ along x-axis gives us point P. Also, $-P$ is an additive inverse of P under $+$ group operator.
- if P is the mirror reflection of Q with regard to x-axis, then $P = -Q$ and
- $O + O = O$ and $O = -O$

The point addition procedure described above can be algebraically expressed. Again, suppose we want to add two arbitrary points P and Q on an elliptic curve $E(a, b)$. This process results in having three intersections with the curve such that:

$$P + Q = -R \tag{2.12}$$

which must satisfy that $P + Q + R = O$

The straight line passing through points P and Q is described with an equation :

$$y = \alpha x + \beta \tag{2.13}$$

with the slope:

$$\alpha = \frac{y_Q - y_P}{x_Q - x_P} \tag{2.14}$$

Since both points P and $Q \in E(a, b)$ and the line of the equation 2.13 joins these points, we can write equation 2.8 as:

$$(\alpha x + \beta)^2 = x^3 + ax + b \tag{2.15}$$

Moreover, if the equation 2.15 has three roots, then there are three intersection points between the straight line described with equation 2.13 and the elliptic curve described with equation 2.8. Two of these roots are x-coordinates of points P and Q, denoted as x_P and x_Q respectively. The third root x_R (x-coordinate of intersection between the line and the curve) can be found using the property of a monic polynomial. Monic polynomials have the coefficient of the highest power of x equal to 1 [25]. It means that equation 2.15 is a monic polynomial, which can also be re-written as:

$$x^3 - \alpha^2 x^2 + (a - 2\alpha\beta)\, x + \left(b - \beta^2\right) = 0 \tag{2.16}$$

The property of monic polynomials states that the sum of all roots equals to the coefficient of the second highest power of x [11]. In other words, the following equation holds:

$$x_P + x_Q + x_R = \alpha^2 \tag{2.17}$$

from which we can find the third root x_R:

$$x_R = \alpha^2 - x_P - x_Q \tag{2.18}$$

We know that the third point R must be on the straight line passing through points P and Q as well. So, we can express y-coordinate of point R as:

$$y_R = \alpha x_R + \beta \tag{2.19}$$

Combining equation 2.19 and equation 2.13 of a straight line for points P and R, we get:

$$y_R = \alpha x_R + (y_P - \alpha x_P) = \alpha (x_R - x_P) + y_P$$

Summarizing the equations derived above, we can calculate coordinates (x_R, y_R) of point R using the following two equations:

$$\begin{aligned} x_R &= \alpha^2 - x_P - x_Q \\ y_R &= \alpha (x_R - x_P) + y_P \end{aligned} \tag{2.20}$$

From the equation 2.12, we know that the result of addition of two points is the reflection of point R along x-axis. So, the final step of point addition is to write equations 2.20 as:

$$\begin{aligned} x_{P+Q} &= \alpha^2 - x_P - x_Q \\ y_{P+Q} &= \alpha (x_P - x_R) - y_P \end{aligned} \tag{2.21}$$

Point doubling

As stated earlier, ECC is based on adding a point to itself k times in order to obtain another point $k \times G$. Adding point to itself is called point doubling and expressed as $P + P = 2P$. Essentially, point doubling is very similar to adding points P and Q. Adding point to itself means that the other point Q approaches point P until they represent the same point on the curve. So, we can compute $2P$ using the following steps:

1. draw a tangent line at P
2. find the intersection of the tangent line with the elliptic curve to obtain point R

3. reflect the point of intersection along the x-axis

Figure 2.4 shows an example of point doubling. An exception to this is when the tangent line is parallel to y-axis. In this case, we say $P + P = O$. This also means that the additive inverse of such point is the point itself.

Point doubling can also be easily expressed algebraically. Similarly to point addition, we calculate a slope. However, we obtain the slope of a single point P at (x, y) by differentiating both sides of equation 2.13:

$$2y\frac{dy}{dx} = 3x^2 + a \tag{2.22}$$

and expressing the slope as :

$$\alpha = \frac{3x_P^2 + a}{2y_P} \tag{2.23}$$

Next, we use equation 2.15 in order to describe three roots of the polynomial. It is important to notice that two roots of this equation will be identical because point Q approaches P. So, the equation 2.17 becomes:

$$x_P + x_P + x_R = \alpha^2 \tag{2.24}$$

from which we can find the x-coordinate of point R:

$$x_R = \alpha^2 - 2x_P \tag{2.25}$$

Almost Identical to point addition, we know that the third point R is on the straight line of the equation 2.19. Using this equations, we can find y-coordinate of the point R:

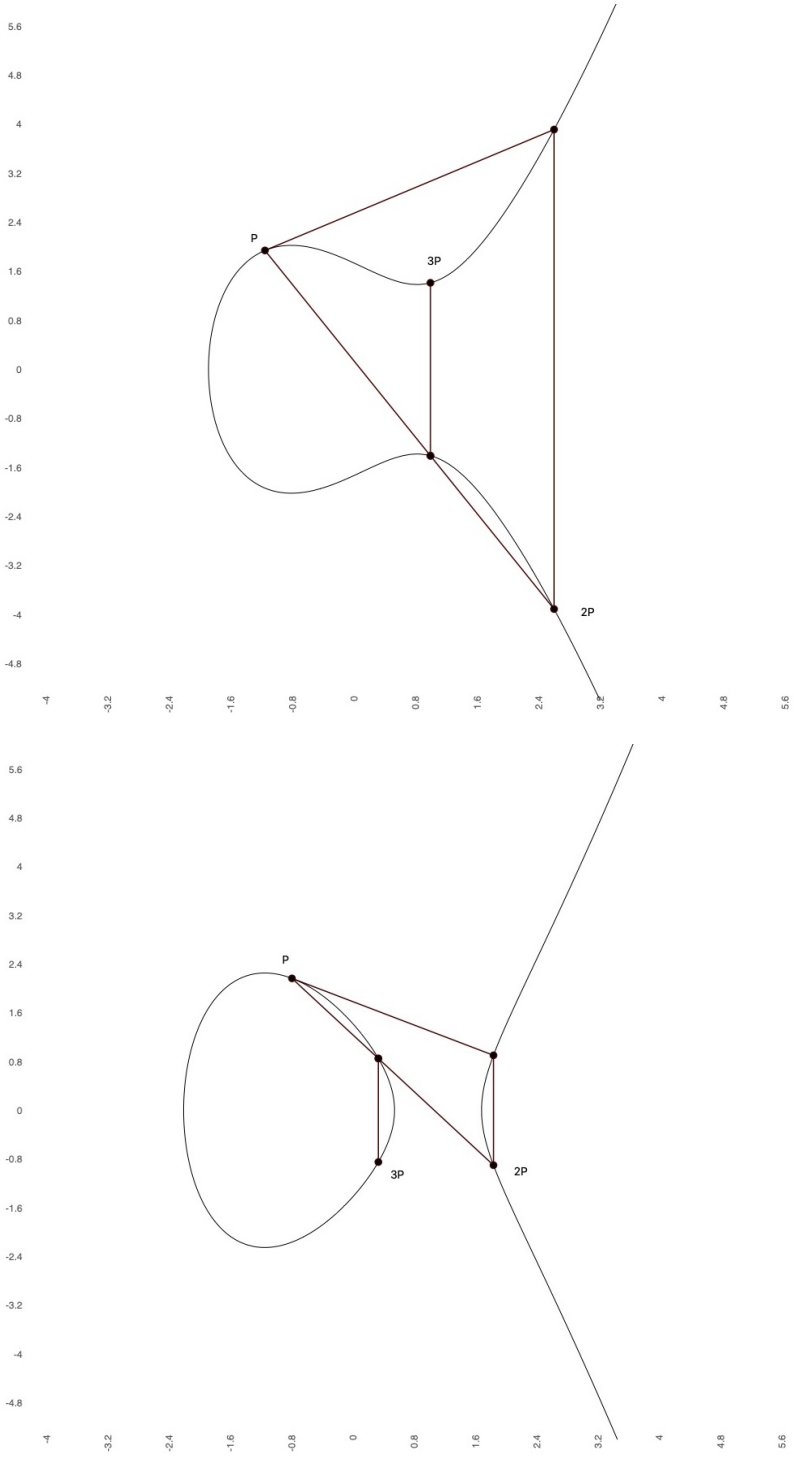

Figure 2.4: Point Doubling on Elliptic Curves

$$y_R = \alpha\left(x_R - x_P\right) + y_P \tag{2.26}$$

The final step is to reflect the point R with respect to x-axis in order to get the point $2P$:

$$\begin{aligned} x_{2P} &= \alpha^2 - 2x_P \\ y_{2P} &= \alpha\left(x_P - x_R\right) - y_P \end{aligned} \tag{2.27}$$

2.2.3 ECC Over $GF(p)$

Elliptic curves over real numbers are not suitable for cryptography. Instead, prime numbers are preferred due to error-free arithmetic provided by prime fields Z_p. ECC over $GF(p)$ work only with values that are in the set $\{0, 1, ..., p-1\}$. It means that parameters a and b along with variables x and y are in the set $GF(p)$. Finally, all operations are carried out modulo p. The elliptic curve described with equation 2.8 becomes:

$$y^2 \equiv x^3 + ax + b \pmod{p} \tag{2.28}$$

where the condition 2.9 is also satisfied in the form:

$$\left(4a^3 + 27b^2\right) \neq 0 \pmod{p} \tag{2.29}$$

A set of points (x, y) on the elliptic curve over $GF(p)$ are denoted by $E_p\left(a, b\right)$ along with a distinguished point at infinity O. These points no longer constitute a curve but a set of discrete points on the plane [11], which means it is impossible to show point addition and point doubling geometrically. However, all algebraic expressions and properties hold under

modulo p operation. The major distinction is how slopes are calculated for point addition and doubling. As for point addition, we can find the slope of a line passing through points P and Q by modifying equations 2.14:

$$\alpha = (y_Q - y_P)\,(x_Q - x_P)^{-1} \pmod{p} \tag{2.30}$$

where $(x_Q - x_P)^{-1}$ is a multiplicative inverse $(\mathrm{mod}\ p)$. Similarly, the slope of a tangent line for point doubling shown in the equations 2.23 becomes:

$$\alpha = \left(3x_P^2 + a\right)(2y_P)^{-1} \pmod{p} \tag{2.31}$$

Finally, the set $E_p\,(a, b)$ is a group with an addition $+$ group operator. The prime number p represent the characteristic of the field Z_p. Prime finite fields with $p \leq 3$ are not safe for cryptography purposes [11].

2.3 Point Encoding

In most cryptographic systems, we need to map a plaintext into a value that can be used by a given cryptography algorithm [22]. ECC is not an exception to this procedure. More precisely, for ECC, we need to convert a message into a point on the elliptic curve. Then we perform point operations described earlier in order to obtain a ciphertext.

The process of converting a message to a point on the elliptic curve has one major problem. There is no deterministic algorithm for writing down points on a curve over $GF(p)$ [22]. However, we can use the Koblitz algorithm [16], which allows us to find an appropriate point on the elliptic curve with a very small probability of an error. Suppose, we are using elliptic curve described in equation 2.28. The plaintext m, represented as a number, will be embedded into the x-coordinate of a point with some additional bits at the end. We cannot use the message m as the x-coordinate because it gives us only 50% of

probability that a square modulo p is equal to $m^2 + am + b$.

Instead, we pick an integer K, which describes a failure rate $1/2^K$. The plaintext message will satisfy the following condition:

$$(m+1)\,K < p \tag{2.32}$$

This restricts the message to be in the following range of values:

$$0 \leq m \leq \frac{p-K}{K} \tag{2.33}$$

The x-coordinate of a point, which contains the encoded plaintext, is described using the following equation:

$$x = mK + j \tag{2.34}$$

where j is in the range $0 \leq j < K$. Then, we iterate through all possible values of j and compute $x^3 + ax + b$ until we find a square root of $x^3 + ax + b \pmod{p}$. This will represent the y-coordinate of the point. If we are not able to find a square root for all possible values of j, then the given message cannot be mapped to a point on a given elliptic curve. Values derived from equation 2.34 and the square root y give us a point $P_m = (x, y)$, which can be used in encryption. In order to get the plaintext m from the point, we use the following equation:

$$m = [x/K] \tag{2.35}$$

Suppose, we have a message $m = 9$ and an elliptic curve over $GF(p)$ represented by $y^2 \equiv x^3 + 2x + 7 \pmod{p}$. If a failure rate of $1/2^{15}$ is sufficient for us, then value of K is equal to 15. Using equation 2.33, we find that the size is restricted to $0 \leq m \leq 10$. From equation 2.34, the x-coordinate will be represented as $x = 9 * 15 + j = 135 + j$. Now, we

iterate through all possible values of $x = \{135, 136, ..., 149\}$ until we find a square root of $x^3 + ax + b \pmod{p}$. For $x = 139$, we get $x^3 + ax + b \equiv 9 \pmod{p}$ or $9 \equiv 3^2 \pmod{p}$. So, we represent the message as point on a curve $P_m = (139, 3)$. The point can be decoded by using equation 2.35, which results in $m = [139/15] = 9$.

2.4 Cryptographic Schemes

Repeated additions are not used directly for encryption described by $m \times G$ [11]. Instead, the concept of point multiplication is applied in many different cryptographic schemes and algorithms. In this section, we introduce the Elliptic Curve Diffie-Hellman key exchange scheme, the ElGamal cryptosystem, and the ECDSA.

2.4.1 Elliptic Curve Diffie-Hellman

Elliptic Curve Diffie-Hellman (ECDH) secret key exchange scheme can be used for establishing a shared secret key between two parties. Suppose, Alice and Bob want to create a secure communication channel. Assume, they choose ECC parameters p, a, and b for an elliptic curve of equation 2.28 and a base point $B \in E_p(a, b)$.

Alice selects an integer X_A as her private key. She also calculates a point $Y_A = X_A \times G$, which will be her public key. At the same time, Bob selects a private key represented as an integer X_B and calculates a point on the curve $Y_B = X_B \times G$. Public keys Y_A and Y_B are shared between the two participants. Now, Alice and Bob can calculate the shared secret key. Alice uses the following equation to obtain the secret key:

$$K = X_A \times Y_B \tag{2.36}$$

while Bob obtains the secret key by:

$$K = X_B \times Y_A \tag{2.37}$$

where K is a point on the elliptic curve in $E_p\,(a, b)$. Equations 2.36 and 2.37 result in the same value of K, i.e., $K = X_A \times X_B \times G$, since $Y_B = X_B \times G$, $Y_A = X_A \times G$, and the elliptic curve group $E_p\,(a, b)$ has the associativity property.

2.4.2 ElGamal cryptosystem

ElGamal cryptosystem is an asymmetric key encryption algorithm based on Diffie-Hellman key exchange. The ElGamal system based on RSA is widely used [22]. It can also be implemented over ECC. The elliptic curve version of ElGamal operates on the points of a given curve over $GF(p)$ and involves repeated point addition operations rather than exponentiations used in RSA [17].

Alex and Bob agree on an elliptic curve and a base point $B \in E_p\,(a, b)$. Alice selects a random large integer $a = \{1, 2, ..., p-1\}$ as her private key. Bob also selects a random large integer $b = \{1, 2, ..., p-1\}$ as his private key. Next, the public key (p, B, G) is calculated, where $G_A = a \times B$ for Alice and $G_B = b \times B$ for Bob. Suppose, Alice wants to send a message m to Bob. The message is first encoded into a point P_m. Then, Alice represents the ciphertext P_c as a pair of points on the curve:

$$P_c = [(a \times B)\,, (P_m + a \times G_B)] \tag{2.38}$$

and sends it to Bob, where B and G_B are obtained from Bob's public key (p, B, G_B).

Bob can decrypt the message by computing the product of the first point from P_c and his private key $b \times (a \times B)$. Then, Bob subtracts this product from the second point of P_c:

$$(P_m + a \times G_B) - \ [b \times (a \times B)\ ] = P_m + a\,(b \times B) - b\,(a \times B) = P_m \tag{2.39}$$

Finally, Bob can decode the original message from the point P_m using equation 2.35.

2.4.3 Elliptic Curve Digital Signature Algorithm

The Elliptic Curve Digital Signature Algorithm is a variant of the Digital Signature Algorithm (DSA) that uses elliptic curves. The ECDSA algorithm is implemented in DNSSEC protocol and blockchain technology to provide sufficient level of security in terms of authenticity. Similar to the ECDH and the Elgamal cryptosystem, both parties have to agree on an elliptic curve equation, a base point B, and a prime integer n, which is the order of B, such that $n \times B = O$.

Suppose, Alice wants to send a message along with the digital signature to Bob. Alice's private key is an integer a that is in range $\{1, 2, ..., p-1\}$. The public key $G_A = a \times B$ is obtained using scalar multiplication, where B is the base point of the selected curve. Alice needs to perform a series of steps to generate a signature of a message m as follows:

1. Calculate $e = HASH\,(m)$ using hashing algorithm
2. Obtain value z by extracting L_n leftmost bits of e, where L_n is the bit length of the group order n
3. Select a cryptographically secure random integer k that is in the range $\{1, 2, ..., n-1\}$
4. Calculate a point $(x_1, y_1) = k \times B$ using point multiplication operation
5. Calculate $r = x_1 \bmod n$. If $r = 0$, go back to step 3
6. Calculate $s = k^{-1}\,(z + ra) \bmod n$. If $s = 0$, then go back to step 3

The generated signature is the pair of values r and s denoted by (r, s). Alice sends it together with the message m to Bob. Bob can verify the received signature using the following steps:

1. Check that both values r and s are in the range $\{1, 2, ..., n-1\}$. If at least one number does not satisfy this condition, then the signature is invalid

2. Calculate $e = HASH\,(m)$ using hashing algorithm identical to the one used by Alice during signature generation process

3. Identical to the signature generation process, obtain value z by extracting L_n leftmost bits of e, where L_n is the bit length of the group order n

4. Calculate the multiplicative inverse of s

5. Obtain values $u_1 = zs^{-1} \bmod n$ and $u_2 = rs^{-1} \bmod n$

6. Calculate a point $(x_1, y_1) = u_1 \times B + u_2 \times G_A$. If $(x_1, y_1) = O$, then the signature is invalid

7. If $r \equiv x_1 \pmod p$, then the signature is valid. Otherwise, the signature is invalid

2.5 Jacobian Projective Coordinates

As illustrated in the previous sections, when points are represented in affine coordinates, the point operations on the elliptic curve use arithmetic additions, subtractions, multiplications, squaring, and also compute the modulo multiplicative inverse, since point addition and doubling operations require the calculation the value of the slope. Since, we work on elliptic curves over $GF(p)$, a multiplicative inverse must be calculated as shown in equations 2.30 and 2.31. Calculating multiplicative inverse is computationally expensive, compared to other arithmetic operations. Multiplicative inverse must be calculated multiple

times because point multiplication involves multiple point addition and multiplication operations. Since computing the modulo multiplicative inverse is computationally expensive, it is practical to represent the elliptic curve points in projective coordinates. One of the efficient coordinates for the elliptic curves over $GF(p)$, is the Jacobian projective coordinate system.

Jacobian coordinates could be used to improve the performance of ECC algorithms because it allows us to reduce the number of multiplicative inverse calculations on big integers [18].

A given point in Affine coordinates (x, y) has a representation in Jacobian coordinates of the following form (X, Y, Z). For example, a point P with Affine coordinates (x_P, y_P) can be represented in Jacobian coordinates $(x_P, y_P, 1)$. Oppositely, a point represented in Jacobian coordinates (X, Y, Z) can be converted back to Affine coordinates using equations:

- $x = \frac{X}{Z^2}$
- $y = \frac{Y}{Z^3}$

The point at infinity corresponds to (1, 1, 0), while the negative of (X, Y, Z) is $(X, -Y, Z)$.

Suppose we want to add a point P with coordinates (X_P, Y_P, Z_P) and another distinct point Q with coordinates (X_Q, Y_Q, Z_Q). First, we define variables A, B, C, and D described by equations:

- $A = X_P * Z_Q^2$
- $B = Y_P * Z_Q^3$
- $C = X_Q * Z_P^2 - A$
- $D = Y_Q + Z_P^3 - B$

Now, the coordinates (X_R, Y_R, Z_R) representing the result of point addition $R = P + Q$ can be obtained by:

- $X_R = -C^3 - 2A * C^2 + D^2$
- $Y_R = -B * C^3 + D(A * C^2 - x_R)$
- $Z_R = Z_P * Z_Q * C$

If we want to perform point doubling operation on a point represented in Jacobian coordinates, where $P + P = 2P = R$, we need to calculate three variables A, B, and C using the following equations:

- $A = 4X_P * Y_P^2$
- $B = 3X_P^2 + a * Z_P^4$
- $C = -2A + B^2$

then coordinates of the point R are obtained using the following equations:

- $X_R = C$
- $Y_R = -8Y_P^4 + B(A - C)$
- $Z_R = 2Y_P * Z_P$

Chapter 3: ECC Implementation

Implementations of the ECC require an understanding of the main components of the ECC from the software engineering prospective. We identify 4 main components of any security system implemented using ECC. We present the hierarchy of these components in a pyramid-like view in order to underline the dependence of all layers from each other. Figure 3.1 shows these components.

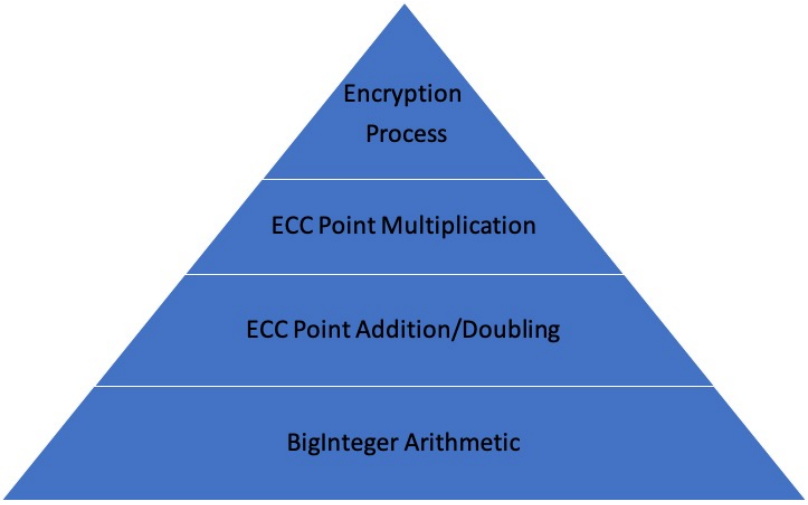

Figure 3.1: ECC Components Pyramid

Encryption algorithms that utilize properties of ECC are based on scalar multiplication. Scalar multiplication is a combination of point addition and point doubling techniques, which requires a way to handle big integers because standard primitive data types are not able to handle values larger than 64 bits. Moreover, big integer arithmetic is needed for representing a plain text message as a point on an elliptic curve. Thus, big integers are

the base for every arithmetic operation and point operation of ECC.

In this chapter, we first introduce algorithms and data structures used in our customed Big Integer class. Then, we illustrate the implementations of elliptic curve point addition, doubling, and multiplication with two distinct implementations of the Big Integer objects using character arrays, and bit sets. Next, we illustrate the working mechanism of the ECDH (Elliptic Curve Key Exchange), the ElGamal encryption /decryption algorithms, the ECDSA and their implementations. We also justify our design choices and considerations during the implementations of ECC. We demonstrate our implementations using a real real SEC (Standards for Efficient Cryptography) ECC curve over a prime field, i.e., the secp192r1 curve whose parameters are presented in table 3.1 [1]. However, our implementation works on any valid elliptic curve over $GF(p)$.

Table 3.1:

Parameter	**Value**
prime number p	FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFFFF FFFFFFFF
a	FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFFFF FFFFFFFC
b	64210519 E59C80E7 0FA7E9AB 72243049 FEB8DEEC C146B9B1
base point G	04 188DA80E B03090F6 7CBF20EB 43A18800 F4FF0AFD 82FF1012 07192B95 FFC8DA78 631011ED 6B24CDD5 73F977A1 1E794811

3.1 Big Integer Class Implementation

The computations over Big integers are the basis for all arithmetic operations in public cryptography. We design our own Big Integer class for exploration and potential performance improvement purposes. Also, we achieve flexibility with the implementations of Big Integer classes for the elliptic curves, which can be supplied by a user. In our master thesis research, the Big Integer class are implemented using character arrays, and bit sets. The first Big Integer class uses array of characters to represent each digit of a large number.

The second Big Integer class stores binary representation of integers in an array of Boolean values, i.e., bit sets.

The main question to be answered before implementing the big integer class is what is the best suitable data structure for representing a big integer of any size. We could consider linked list as an option. However, accessing elements in a linked list has $O(n)$ complexity. In addition to this, linked lists introduce performance overhead because each node needs to store the pointer to the next node. Vectors provided in the standard C++ library are built using dynamic arrays [4]. This data structure is easy to use and provides rich functionality. However, we do not know how the size of the dynamic array will be changed internally during run time. We want to have control over the memory usage in our implementations. Arrays seem to be the best option for our implementation because working with primitive data types is faster in terms of performance.

The second question we want to answer is what is the most suitable data type for the array to hold. We represent each digit of a big integer as a number in the range 0-9, where each digit is stored in a separate index of an array. Using integer data type is not efficient in terms of memory consumption because each int value takes 32 bits. The best option for us is to choose $char$ data type to represent each digit of big integers [5]. Ideally, we can represent big integers as an array of $long$ values instead of character arrays. For example, it is possible to represent 320 bits integer as an array of $long$ values with size 5.

In addition, we need to consider the order in which the digits are stored in an array. Naturally, the arithmetic operations always require accessing least significant bit (LSB) first [5]. That is why, we store the digits in LSB format. In this case, only printing of these values will require starting from the last index on the array. However, this approach eliminates the flexibility of our solution. Our implementation allows the usage of any elliptic curves by simply modifying parameters of an elliptic curve equation 2.28 only.

Character array is a suitable choice for our goals because it gives flexibility to work with the ECC parameters of any size and is considered to be relatively efficient. The sec-

ond implementation of the Big Integer class using bit sets is explored because arithmetic operations such as addition and multiplication can be implemented without any data dependencies. Moreover, division and exponentiation can be implemented using algorithms that require less data manipulations. Each bit of an integer is stored as a Boolean value in a separate index of the array. Since each $bool$ value takes 8 bits of memory, we attempt to make a trade-off between speed and memory in the favor of speed using arrays of Boolean values. Identically to the character array version, we store numbers in LSB format.

Both implementations of Big Integer class support all arithmetic operations. In this section, we describe the detailed implementation and illustrate how the mathematical operations (addition, subtraction, multiplication, division, modulus, and modulo exponentiation) work with pseudocode algorithms. In addition to this, we implement comparisons and shift operators for each version of the Big Integer class. Thus, these big integer classes can be used in other application areas other than cryptography.

3.1.1 Big Integer Addition

Addition operation on big integers is one of the most important and basic operations needed in public cryptography. We overload $+$ addition operator in order to add two big integers. The operator takes two Big Integer objects, performs addition and returns result of the addition as a Big Integer object.

There are several cases to be considered when performing the addition operation. More precisely, we need to consider sizes and signs of operands. Overloaded addition function performs a check based on several conditions. First, we check if two numbers have identical sign. When signs are identical, we can directly perform addition operation on either negative or positive numbers. We use wrapper function $add()$ in order to provide a logical separation between the addition operator and possible cases as illustrated above. If both numbers have identical signs, we compare lengths of these numbers and make a function call to $add()$ function placing the longer number as the first parameter and a shorter

number as a second parameter.

Algorithm 1: The pseudocode of *add()* function using char array.

```
   Input: num1, num2
   Output: result
 1 carry = 0;
 2 tempSum = 0;
 3 i = 0;
 4 while i < num1.size do
 5     if i < num2.size then
 6         tempSum = num1[i] + num2[i] + carry;
 7         tempSum += carry;
 8         carry = 0;
 9         if tempSum > 9 then
10             result[i] = tempSum - 10;
11             update carry;
12         else
               result[i] = tempSum;
           end
13     else
14         tempSum = num1[i];
15         tempSum += carry;
16         carry = 0;
17         if tempSum > 9 then
18             result[i] = tempSum - 10;
19             update carry;
20         else
               result[i] = tempSum;
           end
       end
21     i++ ;
   end
22 return result;
```

Algorithm 1 shows the pseudocode algorithm for adding two big integers represented as a character array. The algorithm is linear and requires iterating through all elements of the arrays. The inputs are two character arrays stored in variables $num1$ and $num2$ respectively. We use a loop to iterate through each digit of the first number. Since, the second number can be shorter than the first one, we check boundaries of the second number.

If a value of the counter i is within the length of the array, we add two digits and the carry in value together and store it in a variable that holds this sum. Next, we check if the the temporary sum is greater then 9. If this condition is true, then we need to subtract the value of the base, which is 10, and update the carry value for the next digit in the array, because each digit can only be in the range $\{0...9\}$. Otherwise, we just place the value of the sum into the result array at the current index i. If we run out of digits in the second operand $num2$, we need to fill out the rest of the array stored in $num1$ variable.

As for binary version, we follow the same logic. We override the $+$ operator and use wrapper $add()$ function if both operands have the same sign. However, the advantage of working with binary number is that we can eliminate using $if - else$ statements. There should be a performance advantage because $if - else$ statements slow down loops, depending on the compiler. We achieve that by using full adder algorithm for adding two numbers.

Algorithm 2: The pseudocode of *add()* function using bool array.

```
   Input: num1, num2
   Output: result
1  carry = 0;
2  tempSum = 0;
3  i = 0;
4  while i < sumSize do
5  |   result[i] = (num1[i] XOR num2[i]) XOR carry;
6  |   carry = ((first[i] & second[i]) OR (first[i] & carry)) OR (second[i] & carry);
7  |   i++ ;
   end
8  if carry > 0 then
   |   result[sumSize - 4] = 1;
   end
9  return result;
```

Algorithm 2 describes the logic for adding two binary numbers represented as arrays that hold Boolean values. Inputs to the function are two arrays $num1$ and $num2$ of the same size. We use a loop to iterate through all elements of arrays. Inside the loop, we use

a full adder algorithm, which utilizes bitwise operators. Suppose we want to add numbers A and B. Full adder algorithm consists of two operations. First operation calculates the sum $sum = A \wedge B \wedge C_{in}$, where C_{in} is the carry in. Second operation calculates carry out value $C_{out} = (A \ \& \ B) \mid (A \ \& \ C_{in}) \mid (B \ \& \ C_{in})$. Thus, for each bit of the number we are able to compute output of a particular bit and update the carry. Finally, after the loop is finished we check if there is a carry value that must be placed in front of the result.

3.1.2 Big Integer Subtraction

Our implementations of Big Integer class support subtraction operation. We overload subtraction operator $-$. However, it was mentioned above that addition may involve numbers that have different signs. That is why, we can convert subtraction to addition. More precisely, operation $A - B$ is converted to $A + (-B)$. That results in calling overloaded $+$ operator. Internally, $if-else$ conditions are used to call either $add()$ or $subtract()$ wrapper function inside the addition operator function.

Algorithm 3: The pseudocode of *subtract()* function.

```
   Input: num1, num2
   Output: result
 1 carry = 0;
 2 tempSum = 0;
 3 i = 0;
 4 while i < sumSize do
 5 |   tempSum[i] = num1[i] - num2[i] - carry;
 6 |   carry = 0;
 7 |   if tempSum < 0 then
 8 |   |   tempSum += (base);
 9 |   |   carry = 1;
   |   end
10 |   result[i] = tempSum;
11 |   i++ ;
   end
12 return result;
```

Algorithm 3 subtracts a big integer $num2$ from another big integer $num1$. Since overloaded $+$ operator passes a greater operand as a first parameter to $subtract()$ function, we iterate through every element of an array and calculate the difference between elements of $num1$ and $num2$ at index i and subtracting the carry. Next we check if the result of the subtraction is a negative number. If condition on line 7 is evaluated to true, we subtract a value of the base we are working with (2 if working with binary number and 10 in the implementation that uses the decimal numbers). Finally, we set the carry out value to 1 in order to indicate that a number was borrowed from the next digit in the array. The last step is to put the calculated value of temporary sum to the result at the specified index. This algorithm is applicable to both versions of our Big Integer classes because the only difference is the value of the base we work with. The returned result is normalized inside the overloaded $+$ operator function.

3.1.3 Big Integer Multiplication

Multiplication operation on two big integers is performed using overloaded $*$ operator. Multiplication is relatively straight forward compared to previously presented arithmetic operations. We do not need to use conditional statements to cover all possible cases regarding sizes and signs of the operators. However multiplication operation is the heaviest on memory consumption because we need to construct an array of size $m \times n$ where m and n are sizes of the first and second big integers respectively. Also, there are two extreme cases to be considered. The first case is when one of the operands is equal to 0. In this case, the result will be 0. Another case is when one of the operands is one, then the result will be equal to another operand. All remaining cases will require the implementation of the multiplication logic similar to when multiplication is performed manually.

Algorithm 4 shows the pseudocode for multiplying two Big Integer objects. Inputs to the function are two numbers represented as arrays of characters. The output of the function

Algorithm 4: The pseudocode of *multiply()* function using char array.

```
   Input: num1, num2
   Output: result
1  carry = 0;
2  tempSum = 0;
3  n1 = num1.size;
4  n2 = num2.size;
5  i = 0;
6  result = new char[num1.size + num2.size];
7  while i <= n1 − 1 do
8  |   carry = 0;
9  |   digit1 = num1[i];
10 |   j = 0;
11 |   while j <= n2 − 1 do
12 |   |   digit2 = num2[j];
13 |   |   sum = digit1*digit2 + result[i + j] + carry;
14 |   |   carry = sum / 10;
15 |   |   result[i + j] = sum % 10;
16 |   |   j++;
   |   end
17 |   if carry > 0 then
18 |   |   result[i + j] += carry;
   |   end
19 |   i++ ;
   end
20 normalize result ;
21 return result;
```

is the multiplication result. Before the multiplication process, we initialize an array called $result$, whose size is the sum of the size of the two operands. At most we can have an array of length $2n$, where n is the length of two input arrays of same size. Multiplication process is essentially multiplying each digit of the first number by the entire number and accumulating the result of multiplication in the result. More precisely, we take one digit at a time using the outer for loop created on line 7. The inner for loop is used to multiply this digit by every digit of the second number. The intermediate multiplication result of two digits is stored in a variable called sum and added together with the carry value. On lines 14 and 15, we calculate a carry out value by dividing the sum by 10 because we work with

base 10 numbers and place the result modulo 10 at index $i + j$. Index j is used to keep track of the proper index of the result. Finally, when we finish multiplying a number by a digit, we check if there is a carry out value and place it at the corresponding index position. After multiplication is finished, the result is normalized by removing leading zeros in front of the MSB of the number.

Algorithm 5 presents the pseudocode for multiplication operation on arrays of Boolean values, which is slightly different. Two Inputs have the same size. However, We initialize the intermediate multiplication result matrix represented as 2D array first.

Algorithm 5: The pseudocode of *multiply()* function using bool array.

```
   Input: num1, num2
   Output: result
1  carry = 0;
2  tempSum = 0;
3  i = 0;
4  matrix = new bool[size][size*2];
5  while i < rows do
6  |  if num2[i] == 1 then
7  |  |  copy elements of the first array to the matrix at row of index i with a shift of i
   |  |    bits;
   |  end
8  |  i++;
   end
9  i = 1;
10 result = bool[size*2 + 4];
11 while i < rows do
12 |  add row at index i to the result;
   end
13 normalize result;
14 return result;
```

Algorithm 5 shows a declaration of a multiplication matrix on line 4. The size of the matrix is $m \times 2m$ size since both arrays are normalized and size of the result cannot exceed $2m$. Number m represents number of rows in the matrix, while $2m$ is the number of columns. First, we fill out the matrix multiplying each digit of the second number by the first number. Multiplication on binary numbers has an advantage because we can have

only two possible cases:

1. $A \times 1 = A$

2. $A \times 0 = 0$

It means that if we multiply a number A by 1, then the result is one and 0 otherwise. In our case, line 6 checks if the bit of the second number at index i is one. If it true, then we copy contents of the first array into the row at index i with a shift of i bits. If the bit of the second number at index i is 0, then we do not do anything because the result of multiplication is zero and all values inside arrays in C++ are initialized to 0 by default. We cannot add multiple binaries number simultaneously. Thus, we need to add two numbers at a time and update the value of the result. This procedure is handled using the $while$ loop at lines 11 and 12.

3.1.4 Big Integer Division and Modulo Operations

Division and modulo operations are closely related to each other. Just as any other mathematical operation of our Big Integer class, operators $/$ and $\%$ are overloaded. Suppose we want to divide number A by another number B. We need to consider the extreme cases:

1. If B is zero, then division operator returns an error.

2. If A is zero, then the result is 0.

3. If $A < B$, the result of division is equal to A.

In all other cases we need to perform division operation by manipulating digits of both numbers. We could use repeated subtraction but this would be very inefficient. Instead, we implement long division algorithm inside a $divide()$ wrapper function that is called inside the overloaded $/$ operator.

Algorithm 6: The pseudocode of *divide()* function using char array.

```
   Input: dividend, divisor, type
   Output: quotient || remainder
1  reminder = 0;
2  quotient = 0;
3  i = dividend.size-1;
4  compute look up table;
5  while i >= 0 do
6  |  remainder[0] = dividend[i];
7  |  quotient << 1;
8  |  if remainder >= divisor then
9  |  |  quotient[0] = get temp quotient from look up table;
10 |  |  remainder = remainder - (temp quotient*divisor);
   |  end
11 |  remainder << 1;
12 |  i=i-1;
   end
13 if if type = 1 then
14 |  return remainder;
   end
15 else
16 |  return quotient;
   end
```

Algorithm 6 describes the logic of long division of two big integers using character arrays. In every case, the dividend is greater than the divisor. An advantage of this algorithm is that it calculates a quotient and a remainder simultaneously. That is why, the algorithm can be utilized in modulo operation. Initially, both quotient and remainder are initialized to 0 and the arrays, which represent the quotient and the remainder have the same size as the dividend. In each round of the while loop, we push digits of the quotient to the remainder one by one starting at MSD and shifting digits of the quotient to the left. It is done until the statement on line 8 is evaluated to true. which means that the temporary value of the remainder is greater than the divisor. Now we need to figure out what number should be added to the quotient by using precomputed table that contains values in $\{0 \times divisor, 1 \times divisor, ..., 9 \times divisor\}$ range. Next, we update the value of the remainder by subtracting the result of multiplication shown on line 10. Essentially, this algorithm

is identical to long division algorithm performed manually. Finally, during each iteration of the loop we shift digits of the remainder to the left in order to place the next digit of the quotient in the remainder. The function returns either the quotient or the remainder, based on the $type$ variable. In case of division, the type is 0 and the quotient is returned.

Long division technique for binary numbers is shown in algorithm 7, which is almost identical to algorithm 6. However, we do not need to precompute the lookup table and use it to find a right digit for the quotient because we have only two possible values 0 and 1; So, we directly subtract divisor from the remainder on line 8 and put 1 in the quotient array at index i.

Algorithm 7: The pseudocode of *divide()* function using bool array.

```
   Input: dividend, divisor, type
   Output: quotient || remainder
1  reminder = 0;
2  quotient = 0;
3  i = dividend.size-1;
4  while i >= 0 do
5  |  remainder << 1;
6  |  remainder[0] = dividend[i];
7  |  if remainder > divisor then
8  |  |  remainder = remainder - divisor;
9  |  |  quotient[i] = 1;
   |  end
   end
10 if if type = 1 then
11 |  return remainder;
   end
12 else
13 |  return quotient;
   end
```

Modulo operation is based on division operation, which is implemented inside the overloaded $\%$ operator and uses wrapper member functions presented earlier in this chapter. Thus, it is suitable for both versions of big integer classes. However, if numbers A and B are equal to each other, then the result will be zero. The modulo operation does not cover

the case when the second operand is negative since modulo operations using a negative divisor are not used in public key cryptography systems. All other cases are covered in algorithm 8.

Algorithm 8: The pseudocode of modulo operation.

```
   Input: num1, num2
   Output: result
1  if num1 < num2 then
2      if same signs then
3          return num1;
       end
4      else
           return num2 - num1;
       end
   end
5  else
6      if same signs then
7          result = num1.divide(num2, true);
       end
       else
8          result = num1 / num2;
9          result = num1 - result * num2;
       end
   end
10 return result;
```

Suppose we want to perform $A\%B$ operation. Absolute values of these numbers are the inputs to the modulo function shown in algorithm 8 and represented as $num1$ and $num2$ respectively. We first check if the first number is less than the second number. If these numbers have identical signs, then the result will be $num1$, for example $5 \bmod 8 = 5$. Otherwise, we subtract $num1$ from $num2$, for example $-5 \bmod 8 = 3$.

All the other possible cases are when the absolute value of the first number is greater than the second. We also check if numbers A and B have same signs. If they do, then we use $divide()$ function to get the remainder of the division operation. Otherwise, we get the quotient, multiply it by $num2$, and subtract this value from $num1$.

3.1.5 Big Integer Modulo Exponentiation

Exponentiation operation is used in many algorithms in ECC. Simply thinking, exponentiation is a process of repeated multiplication of a number by itself. However, this approach will overwhelm system resources when we work with large numbers. Instead, we apply repeated squaring algorithm that results in n multiplications at most, where n is the length of the exponent in bits.

Algorithm 9: The pseudocode of *pow()* function using char array.

```
   Input: num1, num2, p
   Output: result
1  result = 1;
2  i = 0;
3  num2Binary = DecimalToBinary(num2);
4  if num2=0 then
5  |  return 1;
   end
6  else
7  |  while i >= num2.size do
8  |  |  if num2Binary[i]=1 then
9  |  |  |  result = result * num1;
10 |  |  |  result = result % p;
   |  |  end
11 |  |  num1 = num1 * num1;
12 |  |  num1 = num1 % p;
13 |  |  i++;
   |  end
   end
14 return result;
```

Algorithm 9 describes the squaring algorithm used for calculating $A^B \bmod p$. The inputs to the function are $num1$ representing A, $num2$ representing B, and p representing a prime number, which is the order of a group we work with during ECC implementations. The function covers an extreme case when $num2 = 0$ on lines 4 and 5. Since, we cannot access bits of $num2$ directly, we do a conversion from base 10 to base 2 of the exponent. Next, we iterate through each bit of the binary number. Every iteration requires multiplying

number by itself and performing modulo operation because we want the result to be in the range $\{0...p-1\}$. If a bit at index i is 1, then we multiply result by $num1$ and round the value using modulo operation.

Implementations of Big Integers using array of Boolean values does not need the conversion from base 10 to base 2 since all numbers are in the binary format already. In general, we also overload ^ operator that uses the same technique but without lines 10 and 12 because modulo operation is not needed in these cases.

3.2 ECC Point Operations

Arithmetic operations on points of elliptic curves are essential for any applications of ECC. Scalar multiplication (Multiplying a scalar with a point), also known as repeated point addition, is the key element that provides forward secrecy. Scalar multiplication consists of point addition and point doubling. For the purposes of point operations on elliptic curves, we implement a Point class that has two private members, which represent the x and y coordinates of a point represented as Big Integer objects. We implement basic setters and getters to access those coordinates. Most importantly, we implement $add()$, $double()$, and $multiply()$ public member functions. In this section, we describe algorithms of ECC point operations along with any additional algorithms required to support these operations.

3.2.1 Point Addition

Algebraic expressions for point addition on elliptic curves are presented in equation 2.21. These equations can easily be coded in C++ since we have Big Integer classes that support all arithmetic operations.

Algorithm 10 describes the procedure of adding point $p1$ to a point $p2$ on an elliptic curve. Equation 2.21 requires the value of the slope to be calculated. Lines 3 and 4

Algorithm 10: The pseudocode of adding two points on a curve

```
   Input: p1, p2
   Output: result
1  if p1 = p2 then
2  |   return double p1;
   end
3  dY = p2.y - p1.y;
4  dX = p2.x -p1.x;
5  if dX is negative then
6  |   flip signs of dX and xY;
   end
7  dX = gcdExtended(dX, p);
8  slope = dY*dX%p;
9  result.x = (slope.pow(2, p) - p1.x - p2.x) % p;
10 result.y = slope * (p1.x - result.x) - p1.y;
11 return result;
```

show how to compute dY and dX that represent the differences of y and x coordinates respectively. Next, we check if the difference of x coordinate is negative. If it is the case, then we move the negative sign from denominator to numerator by flipping signs of dX and dY. Since all operations on elliptic curves over $GF(p)$ are carried out with modulo p operations, we need to calculate the multiplicative inverse of dX using the Extended Euclidean Algorithm described below. The equation for the slope 2.14 uses the division of dY and dX, since we are dealing with $GF(p)$, we convert the division to multiplication using multiplicative inverse. Finally, we code equations 2.21 directly in order to obtain the coordinates of the result point.

Algorithm 11 describes the pseudocode implementation of the Extended Euclidean Algorithm in C++. Recall from chapter 2, the algorithm calculates values x and y in the equation 2.6. The algorithm iterates until the value of the second number b is equal to 0. We declare helper variables x, $lastx$, $temp$, and $tempa$ for keeping track of previous values of the variables. Inside the while loop, we calculate quotient and remainder values by using division and modulo operations, then replacing values of x and $lastx$ while performing calculations with the new values of x. The same is true for values of y and $lasty$. This is

Algorithm 11: The pseudocode of the Extended Euclidean Algorithm

```
   Input: a, b
   Output: multiplicative inverse of a
1  x = 0 y=1 lastx = 1 lasty = 0;
2  temp = 0 temp2 = a temp1 = b;
3  while b != 0 do
4  |   q = a/b;
5  |   r = a%b;
6  |   a = b;
7  |   b = r;
8  |   temp = x;
9  |   x = lastx - q * x;
10 |   lastx = temp;
11 |   temp = y;
12 |   y = lasty - q*y;
13 |   lasty = temp;
   end
   return lastx;
```

similar to finding GCD but working backwards. Properties of the relatively prime numbers make these replacements to put the value of the multiplicative inverse of a in $lastx$ variable and the value of multiplicative inverse of b in $lasty$ variable. In algorithm 10, we pass values dX and p to the Extended Euclidean Algorithm. Thus, the algorithm returns the $lastx$ variable because we are interested in the multiplicative inverse of dX.

3.2.2 Point Doubling

Point doubling is another important point operation required for the implementations of ECC. Algebraic expression for point doubling is shown in equation 2.23. We implement the algorithm as a public member function of the Point class.

Algorithm 12 describes the implementation logic for point doubling. The input of this member function is a point $p1$, the output is the result $2 \times p1$. First of all, we need to calculate the slope using equation 2.23. The calculation of the numerator is shown on line 1. Next, we calculate the denominator using the statement on line 2. Next, we

Algorithm 12: The pseudocode of point doubling

```
   Input: p1
   Output: result
1  numerator = 3*p1.x.pow(2,p) + a;
2  denominator = 2*p1.y;
3  denominator = gcdExtended(denominator, p);
4  slope = numerator*denominator%p;
5  result.x = slope.pow(2,p) - 2*x% p;
6  result.y = slope * (p1.x - result.x) - p1.y;
7  return result;
```

find its multiplicative inverse using algorithm 11 because division in algebraic expressions for point operations in $GF(p)$ is replaced with multiplication. On line 4, we use modulo operation to keep the value of the slope within the range $\{0...p-1\}$. Finally, we are able to obtain x and y coordinates of the point $2 \times p1$.

3.2.3 Point Multiplication

Point multiplication is the main operation used in ECC systems. The multiplication is also called a scalar multiplication and can be represented as repeated point addition, i.e., points on an elliptic curve are multiplied by a number. Suppose we want to calculate $3P$. The multiplication can be represented as a series of additions $P + P + P$. To calculate $3P$, we can also perform point doubling, since $P + P = 2P$. This leaves us with $2P + P$. Since these two points are different, we need to add a point $2P$ to another point P, using point addition technique described in chapter 2.

Algorithm 13 shows the pseudocode for multiplying arbitrary point P on an elliptic curve over $GF(p)$ by a scalar multiplier m represented as a Big Integer object. The point multiplication is implemented using double-and-add method. First, the result point is initialized to the point at infinity. Since, there is no way for us to represent infinity values, we say that x and y coordinates of a point at infinity are equal to 0. Similar to repeated squaring algorithm, we obtain a binary representation of the scalar multiplier m and traverse

Algorithm 13: The pseudocode of point multiplication

```
   Input: P, m
   Output: result
1  n = P;
2  result = (0,0);
3  convert m to binary;
4  i = 0;
5  while n < m.size do
6      if m[i] = 1 then
7          if result is point at infinity then
8              result = n;
           end
9          else
10             result = result.add(n);
           end
       end
11     n = double n;
   end
12 return result;
```

through each bit of the number. During each iteration we double the value of the point P using the point doubling operation described in algorithm 12. If the value of a bit at index i is equals to 1, then we need to perform a point addition $2^i * P + result$ and store it in the result value as explained above. However, we also check if one of the points is a point at infinity. Using property of elliptic curves $P + O = P$, we simply assign the current value of P to the result on line 8. We obtain the result of $m \times P$ after the iteration of the loop is completed.

3.3 Message Encoding

Suppose, Alice wants to send a message to Bob. Since ECC works with points on elliptic curves only, the message must be represented as a point on an elliptic curve over $GF(p)$. We recall that an elliptic curve is described by an equation 2.28, where parameters a, b, and prime number p must be specified. Using the parameters of the secp192r1 curve, the

equation 2.28 for an elliptic curve becomes:

$$y^2 \equiv x^3 + 627710173538668076383578942320766641608390870039032496127 6x$$
$$+2455155546008943817740293915197451784769108058161191238065$$
$$(\text{mod } 627710173538668076383578942320766641608390870039032496127 9)$$

Algorithm 14: The pseudocode of message encoding

```
   Input: M
   Output: Point
1  maxMessageSize = (p - K) / K;
2  if m > maxMessageSize then
3  |  return Point(-1, -1);
   end
4  m1 = m + 1;
5  if m1 * K < p then
6  |  j = 0;
7  |  while j < K do
8  |  |  x = m*K + j;
9  |  |  y2 = (x.pow(3,p) + b*x + c) % p;
10 |  |  roots = STonelli(y2, p);
11 |  |  if root.first != -1 AND root.second != -1 then
12 |  |  |  return Point(x, smaller root);
   |  |  end
13 |  |  j++;
   |  end
   end
14 else
15 |  return Point(-1, -1);
   end
16 return Point(-1, -1);
```

Algorithm 14 describes the pseudocode for representing a message as a point on a given elliptic curve with specified parameters described above. We make an assumption that a message has been represented as a number. The approach for encoding a message

is probabilistic and we consider it reasonable by choosing value of K equals to 10, which makes the probability of failure to map a message on an elliptic curve $1/2^{10}$. Lines 1-3 of the algorithm are used to check if the message is within the allowed limits defined by equation 2.33. If the integer used to represent a message is larger than the allowed maximum value, we return a point with coordinates $(-1, -1)$. Next, we check if the condition 2.32 is satisfied. The process of mapping a message on the curve starts at line 7, where we iterate through all possible values of the x-coordinate described by equation 2.34. For each value of x, we compute value of $y^2 \bmod p$ using the right hand side of the elliptic curve equation. We need to check if the square root exists in order to obtain the value of y. Having y^2 and p, we can find the square root of y^2 modulo p by using Tonelli-Shanks algorithm described below. If we are able to find the value of y-coordinate, then we return the x and y-coordinates of the point as an object of type Point. Otherwise, we return a point with coordinates $(-1, -1)$.

Algorithm 15 shows the pseudocode of Tonelli-Shanks algorithm. The algorithm is generally used to compute the square root r that satisfies the equation $r^2 \equiv n \pmod{p}$, where p is a prime. In other words, we can find the square root of n modulo p. The algorithm, takes an arbitrary number n and a prime number p as its inputs. The algorithm returns two square roots modulo p of the number n that we are interested in. Lines 1 and 2 of the algorithm are used for checking Euler's criterion, which says n has a square root if and only if the following condition is satisfied: $n^{\frac{p-1}{2}} \equiv 1 \pmod{p}$. If the criterion is satisfied, then we can find the square root of n. Otherwise, we return a pair of values $(-1, -1)$ indicating that the square root does not exist. Next, we are trying to find a positive value of q by factoring out powers of 2. If the counter for factoring out the powers of 2 is equal to 1, then we compute the value of both roots on lines 9 and 10. Otherwise, we search for a value of z that is a quadratic non-residue. Next, we define a number of variables to perform manipulations using repeated squaring algorithm. On line 17, we check if the value of t is equal to 1 before each iteration of the loop. If t is 1, then the root is found

Algorithm 15: The pseudocode of TonelliShanks algorithm

```
Input: n, p
Output: r1, r2
x1 = p - 1 expTemp = x1 / 2 temp = n.pow((p-1)/2, p);
if temp != 1 then
    return (-1, -1);
end
q = x1 counter = 0;
while q % 2 == 0 do
    q = q / 2;
    counter++;
end
if counter = 1 then
    r1 = n.pow((p+1) / 4, p);
    r2 = p - r1;
    return (r1, r2);
end
z = 2 temp = z.pow(x1 / 2, p);
while !(temp = x1) do
    z++;
end
c = z.pow(q,p) r = n.pow((q+1)/2, p) t = n.pow(q,p) m = counter;
while true do
    if t = 1 then
        r2 = p-1;
        return (r, r2);
    end
    i = 0 zz = t;
    while !(zz = 1)  i < (m - 1) do
        repeated squaring of zz;
        i++;
    end
    b = c;
    e = m-i-1;
    while !(e ¡ zero) do
        repeated squaring of b;
        e–;
    end
    update r, c, and t;
    m = i;
end
return (-1,-1);
```

because $t^{2^{m-1}} = 1$, which implies $r^2 = n(\mathrm{mod}\ p)$.

3.4 ECDH Implementation

One of the most important applications of ECC is Diffie-Hellman key exchange. We implement ECDH (Elliptic Curve Diffie-Hellman Key Exchange) technique using stream sockets over TCP. The ECDH logic for establishing secure communication is shown in figure 3.2. For illustration purpose, suppose Alice and Bob want to establish a secure shared secret key. In our socket set up, Bob acts as a server and Alice acts as a client. We make an assumption that Alice and Bob agreed on an elliptic curve and a base point G (the secp192r1 curve) in advance.

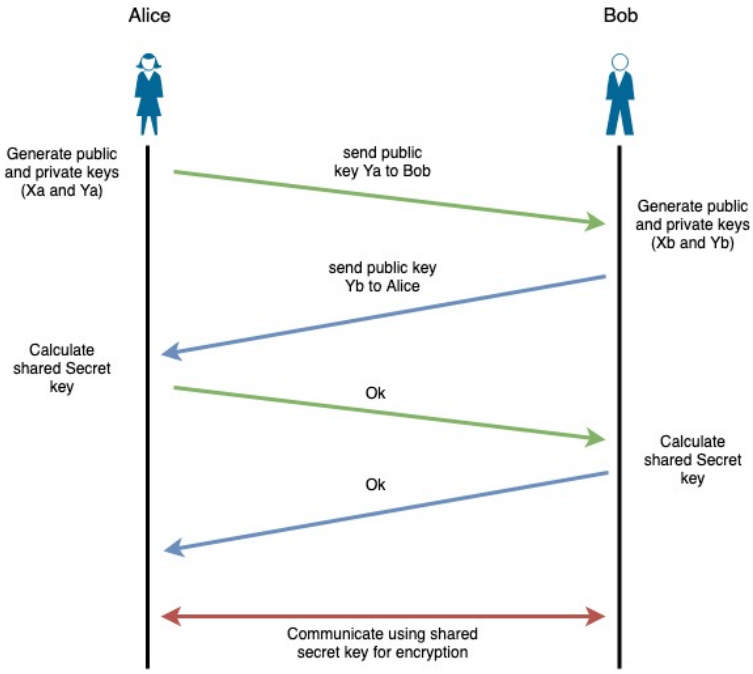

Figure 3.2: The Key Exchange Process Using ECDH.

Alice connects to Bob's machine over standard sockets provided by C++ language. Once socket communication is established, she generates a random number X_a as her private key. She calculates her public key $Y_a = X_a \times G$ using point multiplication operation.

Alice sends the generated public key to Bob. Similarly, Bob generates his private key X_b and public key $Y_b = X_b \times G$ and sends Y_b to Alice. Now, they can both calculate the shared secret key $X_a \times X_b \times G$. Once this process is completed, they exchange a flag message indicating that they are ready to securely send messages to each other using the established shared secret key.

Algorithm 16: The pseudocode for client socket key exchange

```
1  sockfd = socket(afinet, sockstream, 0);
2  if sockfd < 0 then
3  |  print(error opening socket);
   end
4  servername = gethostbyname("localhost");
5  connect(sockfd, serveraddress);
6  if not connected then
7  |  print(error connecting to the server);
   end
8  privKey = genPrivateKey();
9  pubKey = privKey * BasePoint;
10 write(sockfd, pubKey);
11 serverPubKey = read();
12 secretKey = privKey * setverPubKey;
13 write(sockfd, '1');
14 response = read();
```

Algorithm 16 shows the pseudo code for creating a socket, connecting to a server, and creating a shared secret session key between the client's machine and the server. We use standard Linux C built-in header files $types.h$, $socket.h$, and $netinet/in.h$ for socket creation and establishing the connection between the client machine and the server. Lines 1-3 are used to define a socket system call for creating a socket. If the returned value of the system call is negative, then there was an error in creating a socket. We handle this error on lines 2 and 3. Next, we describe the information about the server that the client will connect to. The final step of the socket creation is to connect to the specified server by using the socket created on line 1. If the connection is successful, we generate public and private keys. Private key is randomly generated using built-in $srand\,()$ function, which

takes time as an input seed. The public key is obtained by point multiplication operation. The public key is written to the socket, so the server can use it for calculating the shared secret key. Following the logic described in figure 3.2, we wait for the server's public key in order for the client to calculate the shared secret key. Finally, we send a flag, which has a value 1, in order to confirm the success of the ECDH key exchange process.

Algorithm 17: The pseudocode for the server socket key exchange

```
1  sockfd = socket(afinet, sockstream, 0);
2  if sockfd < 0 then
3  |   print(error opening socket);
   end
4  bind(sockfd, servAddr);
5  if not binded then
6  |   print(error binding socket);
   end
7  listen(sockfd, 5);
8  getClientAddress();
9  acceptConnection(sockfd, clientAdress);
10 clientPubKey = read();
11 privKey = genPrivateKey();
12 pubKey = privKey * BasePoint;
13 write(sockfd, pubKey);
14 response = read();
15 secretKey = privKey * setverPubKey;
16 write(sockfd, '1');
```

The pseudocode for ECDH key exchange on the server side is shown in algorithm 17. It is almost identical to the pseudocode for client side implementation except minor changes. Server side socket initialization requires binding a socket to a specified port and checking if this action was successful, which is shown on lines 4 and 5. In addition to this, the server has to accept a connection from the client. When connection is established, the logic for key exchange in diagram 3.2 is followed on lines 10-16.

3.5 ElGamal Implementation

The ElGamal cryptosystem can be used for encrypting/decrypting of symmetric keys. We implement a sample program that simulates ElGamal ecryption/decryption process between two parties using secp192r1 elliptic curve. The complete process of encryption and decryption is shown in figure 3.3.

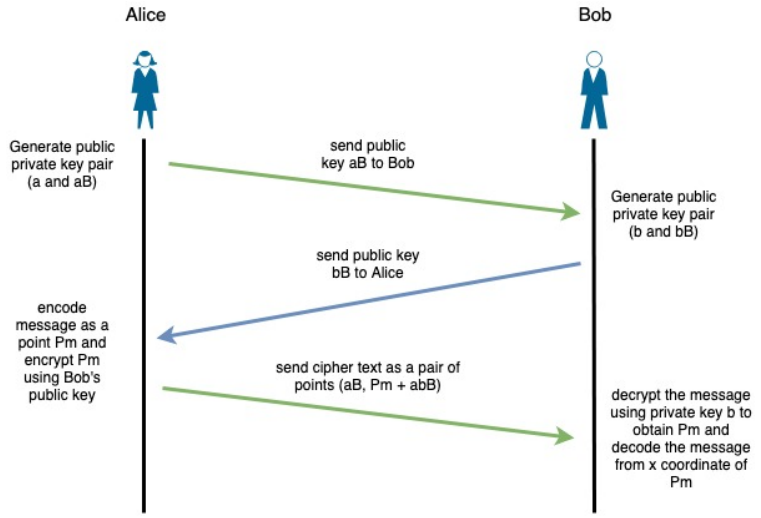

Figure 3.3: Message exchange using the ElGamal cryptosystem.

Suppose, Alice wants to send a message to Bob. We make an assumption that Alice and Bob agreed on a curve and a base point B. Each party calculates their own private key. In our implementation we use randomly generated number. Next, the public key is is obtained using a scalar multiplication of the base point. Alice's public key is calculated using $G_A = a \times B$ equation. Similarly, equation $G_B = b \times B$ used to calculate Bob's public key. After Alice and Bob exchanged their public keys with each other, Alice can securely send a message to Bob. Alice encodes the message as a point P_m on the curve and encrypts it using Bob's public key using equation 2.38. Once Bob receives the cipher text, he is able to recover the plaintext by decrypting the ciphertext as described in equation 2.39 and decoding the message using equation 2.35.

Similar to the implemented the ECDH key exchange mechanism, Bob acts as a server

and Alice acts as a client in the established communication channel using sockets. Algorithm 18 shows the pseudocode for the ElGamal cryptosystem on the client side.

Algorithm 18: The pseudocode for the ElGamal cryptosystem on the client side

```
1  sockfd = socket(afinet, sockstream, 0);
2  if sockfd < 0 then
3  |   print(error opening socket);
   end
4  servername = gethostbyname("localhost");
5  connect(sockfd, serveraddress);
6  if not connected then
7  |   print(error connecting to the server);
   end
8  privKey = genPrivateKey();
9  pubKey = privKey * BasePoint;
10 write(sockfd, pubKey);
11 serverPubKey = read();
12 encodedMessage = encodeMessage(message);
13 cipher = encrypt(encodedMessage, serverPubKey);
14 write(sockfd, cipher);
15 response = read();
```

Lines 1-3 are used for opening the socket for future communication. Client connects to the server machine running on the local host as shown on line 5. Alice calculates her public key using point multiplication algorithm 13 and sends it to Bob. After receiving Bob's public key by reading a message from the socket on line 11, Alice encodes a message as a point on the selected curve using algorithm 14. She also encrypts the message using the function called $encrypt\,()$ that implements the logic described by equation 2.38 that consists of point addition and multiplication operations presented in algorithms 10 and 13 respectively.

Algorithm 19 shows the implementation of the ElGamal cryptosystem for the server side. identical to algorithm 17, lines 1-10 are used to set up a socket and start listening for incoming messages over the stream socket. Bob generates a private key using built-in random number generator and calculates his public key, which is sent to Alice. Next, any

Algorithm 19: The pseudocode for ElGamal cryptosystem on the server side

```
1  sockfd = socket(afinet, sockstream, 0);
2  if sockfd < 0 then
3  |   print(error opening socket);
   end
4  bind(sockfd, servAddr);
5  if not binded then
6  |   print(error binding socket);
   end
7  listen(sockfd, 5);
8  getClientAddress();
9  acceptConnection(sockfd, clientAdress);
10 clientPubKey = read();
11 privKey = genPrivateKey();
12 pubKey = privKey * BasePoint;
13 write(sockfd, pubKey);
14 cipherText = read();
15 encodedMessage = decrypt(ciphertext);
16 plainText = decodeMessage(encodedMessage);
```

incoming messages are considered to be a ciphertext sent by Alice. Bob can decrypt the ciphertext following the logic presented in equation 2.39. The algorithm for decrypting messages is shown in the pseudocode below. The decrypted message is passed to the $decodeMessage\,()$ function, which takes a point with the encoded message embedded in the x-coordinate. If Bob wants to send an encrypted message to Alice, he follows the logic described in lines 12-14 of algorithm 18 but using Alice's public key.

Algorithm 20: The pseudocode for decrypting a message in the ElGamal cryptosystem

```
   Input: p1, p2
   Output: encodedMessage
1  product = b*p1;
2  product.y = -product.y mod p;
3  encodedMessage = p2 + product;
4  return encodedMessage;
```

Algorithm 20 shows the pseudocode for recovering the message represented as a point on a given curve from a ciphertext. The function for decrypting a message takes two points

$p1$ and $p2$ because ciphertext is a pair of points on an elliptic curve in the ElGamal cryptosystem. As shown in equation 2.39, Bob needs to calculate a point, which is the product of the first point of the ciphertext pair and his private key. This is simply done using point multiplication operation. Next, he subtracts this point from the second point of the ciphertext pair. However, there is no point subtraction operation. Instead, we represent subtraction by adding the negative point to the first point of the ciphertext pair. Recall from chapter 2, the negative of a point is the same point reflected against x-axis. However, we cannot simply change the sign of the y-coordinate because we work on the elliptic curves over $GF(p)$. We also need to use modulo operation as shown on line 2 of Algorithm 20. Finally, we add the point obtained on line 1 to the second point of the ciphertext pair. Encoded message is recovered, where the plaintext is encoded in the x-coordinate and can be recovered using equation 2.35.

3.6 ECDSA Implementation

The ECDSA can be used for verifying the integrity of the message and the authenticity of the sender. We implement a sample program to simulate the message exchange process with the support of ECDSA over the secp192r1 curve. The process is described in the diagram shown in 3.4.

Suppose, Alice wants to send a message along with the generated digital signature of the message to Bob. We make an assumption that both parties agreed on a curve, a base point B, and the order n. Alice and Bob calculate their public-private key pairs. Alice calculates her public key $G_A = a \times B$, where a is her private key. Similarly, Bob calculates his public key $G_B = b \times B$, where b is his private key. Next, Alice and Bob exchange their public keys. If Alice wants to send a message along with the digital signature, she generates a pair of values (r, s), which constitutes the digital signature, and sends it to Bob together with the original message. After Bob receives the message and the signature, he

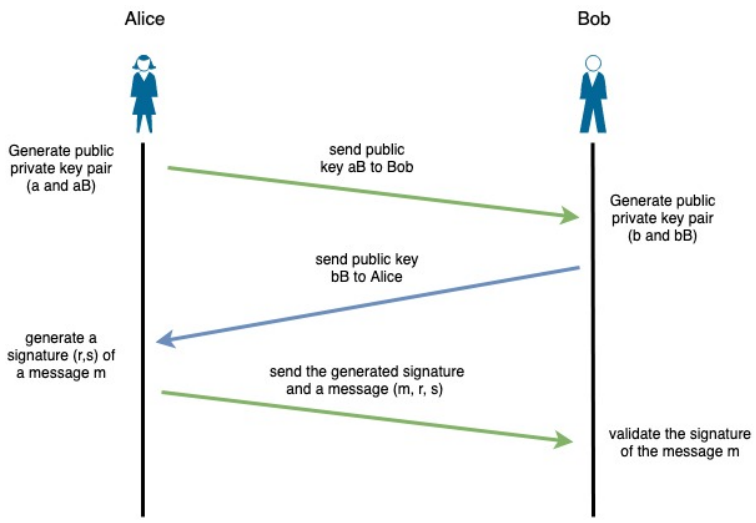

Figure 3.4: Message exchange using ECDSA.

is able to verify the integrity and authenticity of the message using the procedure described in chapter 2.

We simulate the ECDSA algorithm by building communication between two parties using sockets. Identical to the previously described implementations, Bob will act as the server and Alice will act as the client. Algorithm 21 shows the pseudocode for ECDSA Implementation on the client side. Lines 1-7 show the logic for establishing connection with the server using C++ standard sockets. The implementation of key generation and exchange is shown on lines 8-11. Before sending a message to the server, Alice generates the signature using $sign\,()$ function. The message is written to the socket along with the signature as shown on line 13.

Algorithm 22 shows the implementation of $sign\,()$ function, which is responsible for generating the digital signature of a given message. The algorithm takes a message as an input and returns a pair of values r and s that constitutes a signature. We use SHA-256 hashing algorithm provided by CryptoPP library in order to generate the hash of a message. Next, we extract 192 leftmost bits of the generated hash because the order of the secp192r1 curve is 192 bits long. Next, we generate a random number k and perform a

Algorithm 21: The pseudocode for ECDSA on the client side

```
1  sockfd = socket(afinet, sockstream, 0);
2  if sockfd < 0 then
3  |   print(error opening socket);
   end
4  servername = gethostbyname("localhost");
5  connect(sockfd, serveraddress);
6  if not connected then
7  |   print(error connecting to the server);
   end
8  privKey = genPrivateKey();
9  pubKey = privKey * BasePoint;
10 write(sockfd, pubKey);
11 serverPubKey = read();
12 signature = sign(message);
13 write(sockfd, message, signature);
```

Algorithm 22: The pseudocode for signing a message using ECDSA

```
   Input: message
   Output: r,s
1  hash = SHA256(message);
2  z = extract(hash);
3  while true do
4  |   k = generateRandomKey();
5  |   point = k*basePoint;
6  |   r = point.x mod n;
7  |   while r = 0 do
8  |   |   k = generateRandomKey();
9  |   |   point = k*basePoint;
10 |   |   r = point.x mod n;
   |   end
11 |   kInverse = gcdExtend(k, n);
12 |   s = kInverse * (z + r*privateKey) mod n;
13 |   if s = 0 then
14 |   |   return pair(r,s);
   |   end
   end
15 return pair(r,s);
```

point multiplication operation to get an intermediate point. We obtain the first value of the signature using modulo operation as shown on line 6. We check if the value of r is equal

to zero. If it is zero, then we enter a while loop that will iterate until we are able to obtain r distinct from zero. We compute the multiplicative inverse of k and obtain the value of s using equation on line 12. If the calculated value is zero, then we need to start over by going back to line 4. The algorithm returns a pair of values r and s as soon as the valid signature is generated.

The server side implementation is similar to the previously described implementations of ECC applications. Algorithm 23 show the server side implementation for ECDSA. More precisely, lines 1-14 are identical to the pseudocode used in ElGamal implementation on the server side. However, the server needs to verify the integrity and authenticity of the message using the received signature from the client.

Algorithm 23: The pseudocode for ECDSA on the server side

```
1  sockfd = socket(afinet, sockstream, 0);
2  if sockfd < 0 then
3  |   print(error opening socket);
   end
4  bind(sockfd, servAddr);
5  if not binded then
6  |   print(error binding socket);
   end
7  listen(sockfd, 5);
8  getClientAddress();
9  acceptConnection(sockfd, clientAdress);
10 clientPubKey = read();
11 privKey = genPrivateKey();
12 pubKey = privKey * BasePoint;
13 write(sockfd, pubKey);
14 clientMessage = read();
15 verify(clientMessage.signature, clientMessage.message);
```

Algorithm 24 describes the implementation of signature verification using ECDSA. The algorithm accepts two parameters. The first parameter is a pair, which holds two values r and s that constitute a generated digital signature. The algorithm returns a Boolean value to describe if the signature is valid. The second parameter is a message received by the

server. Line 1 is an if statement used to check if the values r and s are within a valid range. If at least one value is out of range, then the function returns false, meaning the signature is invalid. If the values are in the range, we perform a series of steps identical to the signature generation process as shown on lines 3 and 4. Next, we compute the multiplicative inverse of s using the Extended Euclidean Algorithm and obtain values of u_1 and u_2 as shown on lines 6 and 7. An intermediate point on the selected curve is obtained using scalar multiplication and point addition operation on line 8. If the calculated point is a distinguished point at infinity, then the signature is invalid. Otherwise, we calculate the values n_1 and n_2 used in the final step of signature verification process. If these two values are identical, then the signature is valid and the $verify\,()$ function returns true. Otherwise, false Boolean value is returned.

Algorithm 24: The pseudocode for verifying a signature using ECDSA

```
   Input: signature, message
   Output: valid
 1 if r or s are not in the range from 1 to n-1 then
 2 |   return false;
   end
 3 hash = SHA256(message);
 4 z = extract(hash);
 5 sInverse = gcdExtend(signature.s, n);
 6 u1 = sInverse * z mod n;
 7 u2 = (signature.r * sInverse) mod n;
 8 result = (u1*G + u2*publicKey) mod n;
 9 if result = pointAtInfinity then
10 |   return false;
   end
11 n1 = signature.r mod n;
12 n2 = result.x mod n;
13 if n1 = n2 then
14 |   return true;
   end
15 else
   |   return false;
   end
```

Chapter 4: Evaluation

In this chapter, we present the performance evaluation of the arithmetic operations of the Big Integer classes on the operands with various sizes, and the point operations required for all application of ECC on the secp192r1 curve.

4.0.1 Platforms

All arithmetic and point operations have been tested on a PC with a quad-core Intel(R) Core(TM) i7-7700K CPU running Ubuntu 15.04 operating system along with execution time measurements. The implemented software program was compiled and run using standard GNU C++ compiler version 4.9.2. The program was checked for memory-related errors using the dynamic analysis tool Valgrind [12].

4.1 Results

This section presents experimental results. We compare the time performance of the arithmetic operations, i.e., addition, subtraction, division, multiplication, and modulo exponentiation of the Big Integer classes. We also measure and report the execution time of point addition, point doubling, and scalar multiplication operations over the secp192r1 curve that was used to implement the ECDH key exchange mechanism, the ElGamal cryptosystem, and the ECDSA. The program was executed 20 times and the average running time to

perform these operations is reported.

4.1.1 Big Integer Arithmetic Operations

The execution time of several arithmetic operations is measured using operands of various sizes. For each operation, we compare the timing performance of the arithmetic operations of the two versions of the Big Integer classes implemented using array of characters and array of Boolean values respectively.

Table 4.1: Comparison of performance: Addition Operation

Operands Size in bits	**BigInteger addition in μs**	**Bitset addition in μs**
160	0.9024	1.2686
192	0.9602	1.0700
256	0.7904	1.1150
384	1.4684	2.7106
512	1.6644	2.2730

Table 4.1 reports the comparison of the average execution time of addition operation on the two versions of the Big Integer classes with various sizes. Each operand is a randomly generated number of size specified in the first column of the table. We can see that, for both implementations, adding two 192 or 256 bits long numbers is slightly faster than adding two 160 bits long numbers, which is hard to explain. Also, all the arithmetic operations of the Big Integer class implemented using bit set is slower on all operands sizes, which is because the number of loop iterations described in algorithm 2 is larger than the number of loop iterations of the Big Integer class using arrays of characters. In addition, the memory size to represent a certain big integer with bit set is larger than the memory size to represent the same big integer with character array, because internally, every bit in the bit set is stored with one byte in C++. However, the smallest difference in average execution time occurs when the two 192 bits integers are added, which again is hard to explain.

Table 4.2 reports the average execution time of subtraction operation on the two ver-

Table 4.2: Comparison of performance: Subtraction Operation

Operands Size in bits	**BigInteger subtraction in μs**	**Bitset subtraction in μs**
160	0.7214	3.3352
192	0.5934	2.8514
256	0.4484	2.8286
384	0.6130	4.0692
512	0.8030	4.7016

sions of the Big Integer classes with various sizes. Each operand is randomly generated with a specified size. However, we make sure that the operands are not identical in order to avoid the extreme case, which automatically assigns 0 to the the result of subtraction. The execution time of the subtraction operation of the Big Integer classes implemented with bit set is slower as well. However, the differences of execution time between the two implementations of the Big Integer classes are larger than the differences of the execution time of the addition operation. Despite the fact that addition and subtraction operations have the same time complexity, the subtraction operation of the Big Integer classes implemented using arrays of characters is faster than addition. As for the Big Integer class implemented with the bit set, the results are opposite. More precisely, the subtraction operation is slower than the addition operation for the Big Integer class implemented with the bit set.

Table 4.3: Comparison of performance: Multiplication Operation

Operands Size in bits	**BigInteger multiplication in μs**	**Bitset multiplication in μs**
160	30.3884	162.2900
192	50.8660	347.0710
256	62.8070	477.0820
384	148.6238	1408.4024
512	236.4642	2487.5310

The average execution time of the multiplication operation on the two versions of the Big Integer classes on operands with various sizes is reported in table 4.3. Each operand is randomly generated with specified size. We can see that the execution time of the multiplication operation of the Big Integer classes implemented with bit set is way slower than the multiplication operation of the Big Integer classes implemented using character arrays,

which is because the multiplication of binary numbers requires increased number of loop iterations for repeated additions to accumulate the result of multiplication as described in algorithm 5. The table also shows that the larger the size of the operand, the greater is the difference in the execution time between the two different implementations. Finally, the Big Integer classes implemented with Boolean arrays take more space compared to the Big Integer classes implemented with a character array. As the number of loop iterations increases, more time is needed to perform computations.

Table 4.4: Comparison of performance: Division Operation

Operands Size in bits	**Big Integer division in μs**	**Bit set division in μs**
160	103.0684	161.9710
192	194.2446	347.4570
256	223.0538	477.5352
384	500.7140	1403.7164
512	905.4234	2486.3946

Table 4.1 compares the average execution time of the division operation of the two versions of the Big Integer classes with various operand sizes. In this thesis, we divide a randomly generated large number of various sizes by 2 in order to achieve the execution time as close as possible to the worst case scenario. Similar to previously presented measurements, the division operation of the Big Integer classes implemented with bit set is slower than the division operation of the Big Integer classes implemented with character arrays, which is because the memory size to represent a certain big integer with bit set is larger than the memory size to represent the same big integer with character array, because internally, every bit is stored with one byte in C++, so the division operation of the Big Integer classes using bit set takes longer time even without the performance overhead for computing a look up table as in the Big Integer classes using character array. Finally, we observe that as the size of the dividend increases, the execution times increases significantly.

Table 4.5 reports the average execution time of power-modulo operation of the two

Table 4.5: Comparison of performance: Power-Modulo Operation

Operands Size in bits	Big Integer power-modulo in ms	Bit set power-modulo in ms
160	104.8746	315.4860
192	156.6182	489.4174
256	191.8902	558.2576
384	343.8816	943.9348
512	495.5048	1269.3418

versions of the Big Integer classes on big integers of various sizes. We randomly generate big integers of various sizes and measure the execution times of $pow\,()$ described in algorithm 9. For the purpose of this measurement, we calculate $A^B \bmod p$, where A, B, and p have equal length in bits. This is the worst case scenario which, does not occur often in the applications of ECC but often seen in the applications of RSA. The results in the table show us that this power-modulo operation is the most computationally intensive operation compared to the other arithmetic operations because it requires n multiplications and modulo operation, where n is the size of the exponent in bits. The execution time and the size of the operands have linear dependency. Just as any other operation, the power-modulo operation of the Big Integer classes implemented with bit set is slower.

4.1.2 Point Operations on the Secp192r1 Curve

Any ECC application requires point addition, point doubling, and scalar multiplication operations. We report the performance of these operations in table 4.6. Point addition operation is the fastest point operation and takes roughly 7 ms. Point doubling is 3.5 times slower since it involves more multiplication and exponentiation operations, which are expensive as shown in tables 4.3 and 4.5. Scalar multiplication is the most expensive point operation since it consists of point addition and point doubling operations. A given point is doubled at least n times, where n is the size of a scalar multiplier in bits. The point operations of the Big Integer classes implemented with bit set is 2 times slower than the point operations of the Big Integer classes implemented with character arrays.

Table 4.6: Comparison of performance: Point Operations on the curve secp192r1

Operands Size in bits	BigInteger implementation in ms	Bitset implementation in ms
Point Addition	7.0504	19.4550
Point Doubling	25.4880	58.7686
Scalar Multiplication	448.0902	1046.708

Since the Big Integer implementation is faster than the Bitset, we use the Big Integer implementation to compare performance between Affine and Jacobian coordinates. Table 4.7 shows the performance comparisons of point operations between the implementation that uses Affine coordinates and the implementation using Jacobian projective coordinates.

Table 4.7: Comparison of performance: Affine v. Jacobian coordinates on the curve secp192r1

Operands Size in bits	Affine coordinates in ms	Jacobian coordinates in ms
Point Addition	7.0504	8.2093
Point Doubling	25.4880	8.3371
Scalar Multiplication	448.0902	294.576

The table shows that we have a slight performance decrease in point addition operation. However, point doubling operation is almost 3 times faster. This happens because the number of arithmetic Big Integer operations is way smaller when using Jacobian coordinates. Thus, this allows us to obtain a significant performance improvement during point multiplication operation.

4.1.3 Verification of the Correctness

We implemented the ECDH key exchange mechanism and ElGamal encrypting/decrypting algorithms and verified the correctness of these implementations, as illustrated in the following.

Figures 4.1 and 4.2 show the screenshots of the established shared secret key during ECDH key exchange on the client and server side respectively. We can see that both parties

successfully exchanged their public keys with each other and are able to calculate the shared secret key independently. The generated shared secret key is represented in hexadecimal format and is identical on both ends, which verifies the correctness of ECDH key exchange mechanism and the Big Integer classes.

```
kirill@linux445-server2:~/thesis/ecdh/client$ ./main
Welcome to thesis project
connected to the server
sending generated public key Point:{x=596801200211899451683627960735152675634283893
37806684779907, y=22915058770725056246701456608276945254227959210163712070 79}
received server's public key Point:{x=191206453386589467659877671695022979682480477
76264475461392, y=391068854930079589263118956053550308229873209921294812 2100}
server replied, generated secret key is 9CE6C057E93DD4FF08FD13DD89BB23164833E85AFF
D1454E
kirill@linux445-server2:~/thesis/ecdh/client$ _
```

Figure 4.1: Client side of the ECDH Key Exchange.

```
kirill@linux445-server2:~/thesis/ecdh/server$ ./main
Welcome to thesis project
socket opened, waiting for the client to connect...
client connected
received client's public key Point:{x=596801200211899451683627960735152675634283893
37806684779907, y=22915058770725056246701456608276945254227959210163712070 79}
sending genererated public key Point:{x=191206453386589467659877671695022979682480
4776264475461392, y=391068854930079589263118956053550308229873209921294812 2100}
client replied, generated secret key is 9CE6C057E93DD4FF08FD13DD89BB23164833E85AFF
D1454E
kirill@linux445-server2:~/thesis/ecdh/server$ _
```

Figure 4.2: Server side ECDH Key Exchange.

Table 4.8 reports the parameters used by the ECDH key exchange process, and the generated shared secret key on the server side and client side.

Table 4.8: Parameters of the ECDH, and the Generated Shared Secret Key

Parameter	**Value**
Alice's private key	784926782903412329323451
bob's private key	563794302987362142789234
base point	x: 602046282375688656758213480587526111916698976636884684818 y: 174050332293622031404857552280219410364023488927386650641
Alice's public key	x: 596801200211899451683627960735152675634283893780668477990 7 y: 229150587707250562467014566082769452542279592101637120707 9
Bob's public key	x: 191206453386589467659877671695022979682480477626447546139 2 y: 391068854930079589263118956053550308229873209921294812210 0
generated secret key	384721045761190002877775880267880970823033308432675486241 4

We validate the correctness of the ElGamal cryptosystems by simulating the encryption/decryption process when a message is sent from Alice to Bob using Bob's public key, where Bob acts as a server and Alice acts a client. Figures 4.3 and 4.4 show the output of the encryption/decryption process using the Elgamal cryptosystem on the client and server side respectively. We can see that both parties successfully exchange their public key. Alice encrypted a sample message represented as a number, encoding the message as a point on the elliptic curve, using the ElGamal encryption algorithm to generate the ciphertext, which is a pair of points on the elliptic curve, and send the ciphertext to Bob. Bob receives the ciphertext, decrypts the message and recovers the plaintext by decoding the message, which is embedded in the x-coordinate of the point. In figure 4.4, we can see that Bob recovers the plaintext of the message, which is identical to the message that was encoded and encrypted by Alice. That verifies the correctness of the implemented ElGamal cryptosystem.

```
kirill@linux445-server2:~/thesis/elgamal/client$ ./main
Welcome to thesis project
conencted to the server
sending generated public key Point:{x=379157826276864579634350521655
546024571806106743830376447 5, y=216221831333371317524431938312706478
272728258032113640197 0}
received servers public key Point:{x=303985323772030313817010210587 7
758560174370555867245101 2, y=556886972154962743725080676593156287158
120798595027730989 6}
encoding message 986782900181143871212342314312
encoded message: Point:{x=986782900181143871212342314312 0, y=2196348
078618827511118477981636656982591377148662893949597}
first point of ciphertext: Point:{x=379157826276864579634350521655546
024571806106743830376447 5, y=216221831333371317524431938312706478272
72825803211364019 70}
second second of ciphertext: Point:{x=152545346884612895823185990037
253449563301612658927710352, y=255566813423249379998144512386183329 1
704734812170729850119}
sent ciphertext to the server
kirill@linux445-server2:~/thesis/elgamal/client$ _
```

Figure 4.3: Client side of ElGamal Cryptosystem.

Table 4.9 reports the parameters used by the ElGamal cryptosystem, and the plaintext message, the intermediate results and the recovered plaintext during the ElGamal encryp-

Figure 4.4: Server side of ElGamal Cryptosystem.

tion/decryption process.

Table 4.9: Parameters of the ElGamal and the intermediate results of the ElGamal Cryptosystems

Parameter	**Value**
message m	98678290018114387121234231 4312
P_m	x: 98678290018114387121234231 43120
	y: 219634807861882751111847798163665698259137714866289394 9597
P_1 of cipertext	x: 379157826276864579634350521655546024571806106743830376447 5
	y: 216221831333713175244319383127064782727282580321136401970
P_2 of ciphertext	x: 152545346884612895823185990037253449563301612658927710352
	y: 255566813423249379998144512386183329170473481217072985011 9
decrypted P_m	x: 98678290018114387121234231 43120
	y: 219634807861882751111847798163665698259137714866289394 9597
plaintext from P_m	98678290018114387121234231 4312

Similar to the ECDH key exchange process and the ElGamal cryptosystems, we also evaluate the correctness of the ECDSA. In our implementation, Alice acts as a client and Bob acts as a server. Figures 4.5 and 4.6 show the output of the implemented ECDSA using sockets for client and server side respectively. We can see that both parties successfully exchanged public keys between each other. Also, figure 4.5 shows the calculated values r

and s that constitute the digital signature. We see that the server side obtained the same hash values of the message using SHA-256 algorithm. The figure 4.2 shows values of the calculated intermediate parameters, which are required for verifying the validity of the digital signature. Most importantly, we see that $r \mod n$ and $x_1 mod n$ are also equivalent. This means that the digital signature is valid and the implemented ECDSA is correct.

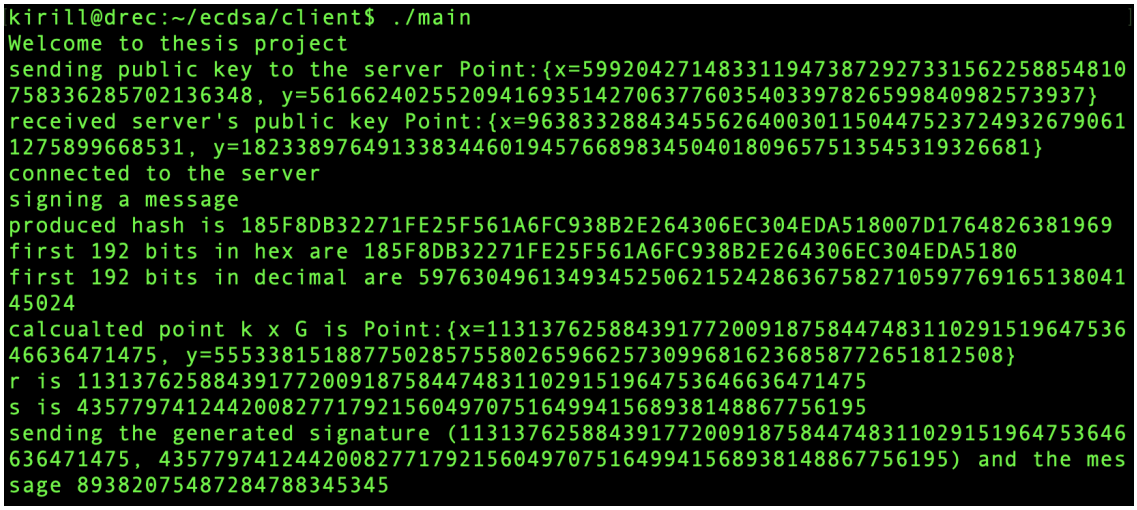

Figure 4.5: Client side of ECDSA.

Table 4.10 summarizes the obtained results and intermediate parameters used in ECDSA. Both parties are able to obtain identical values of the parameter z. We also see that generated value of r by the client side is identical to the received value of r on the client side. Finally, the values of $r \mod n$ and $x_1 \mod n$ are also equivalent.

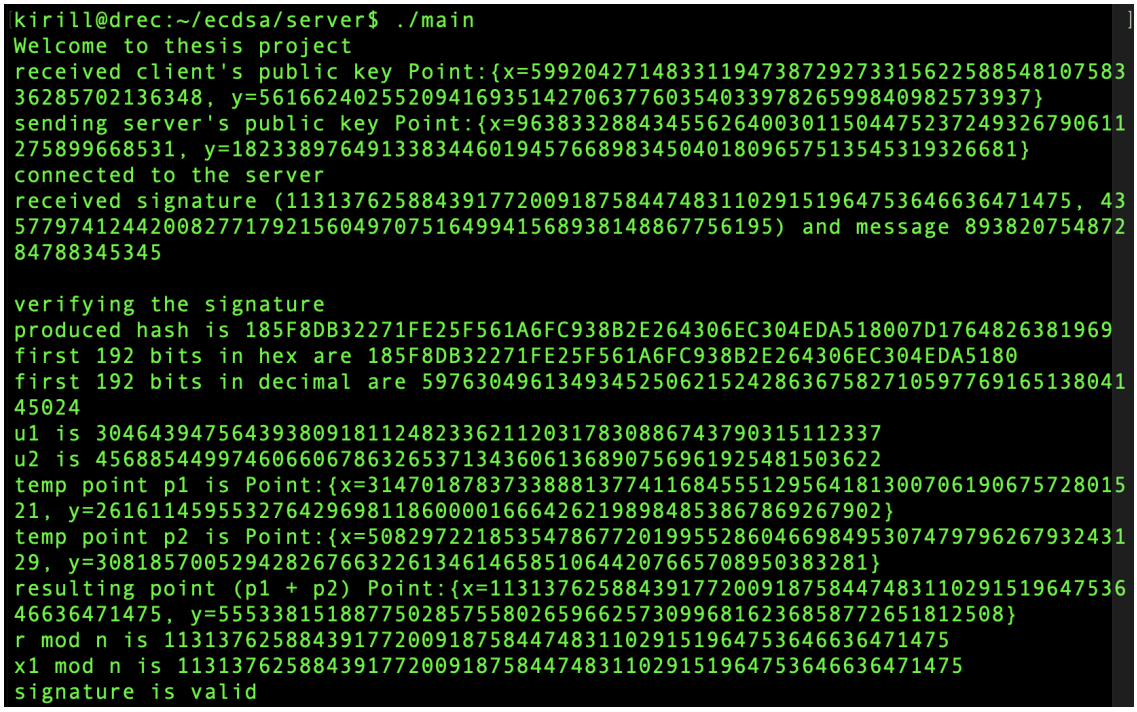

Figure 4.6: Server Side of ECDSA

Table 4.10: Parameters and the intermediate results of ECDSA

Parameter	**Value**
message m	89382075487284788345345
HASH(m)	185F8DB32271FE25F561A6FC938B2E264306EC304EDA518007D1764826381969
z in hex	185F8DB32271FE25F561A6FC938B2E264306EC304EDA5180
z in decimal	5976304961349345250621524286367582710597769165138041 45024
generated r	1131376258843917720091875844748311029151964753646636471475
generated s	4357797412442008277179215604970751649941568938148867756195
u_1	30464394756439380918112482336211203178308867437903151123 37
u_2	4568854499746066067863265371343606136890756961925481503622
calculated $r \bmod n$	1131376258843917720091875844748311029151964753646636471475
$x_1 \bmod n$	1131376258843917720091875844748311029151964753646636471475

Chapter 5: Conclusions and Future Work

Public key cryptography is very important in assuring data integrity and confidentiality, establishing a shared secret key between two communication parties. The widely used RSA algorithm is not able to provide forward secrecy as the computational capabilities of systems increase. In comparison, ECC is able to provide forward secrecy because it requires the attackers to solve discrete logarithm problem to recover the plaintext of the encrypted messages. In addition to this, ECC achieves the same level of security as RSA but with a much shorter key size. The three most important applications of ECC, are the ECDH (Elliptic Curve Diffie-Hellman) key exchange, the ElGamal cryptosystems, and the Elliptic Curve Digital Signature Algorithm. These applications are being adopted in different application areas such as autonomous cars, smart grids, mobile devices, and blockchain.

This thesis focuses on the software implementations of ECC over finite field GF(p) with two distinct implementations of the Big Integer classes using character arrays, and bit sets in C++ programming language. Our implementation works on the ECC curves of the form $y^2 = x^3 + ax + b \ (mod\ p)$. The arithmetic operations of the two different versions of the Big Integer classes using character array and bit sets are implemented and the correctness of these arithmetic operations are verified. The performance evaluation of the arithmetic operations of the two versions of the Big Integer classes on the operands

with various sizes are reported. From the performance results reported in chapter 4, we conclude that the big integer classes using bit set is not suitable for cryptography due to its slow performance. In addition, the big integer classes using arrays of characters could be further optimized.

The point operations including point addition, point doubling, and scalar multiplication operations are implemented on a real SEC (Standards for Efficient Cryptography) ECC curve, the secp192r1 curve, with the support of the big integer classes, and the correctness of these point operations is verified. The timing performances of the point operations required for the ECDH key exchange, the ElGamal cryptosystems, and the ECDSA on the secp192r1 curve are reported.

Three most popular applications of ECC, the ECDH key exchange, the Elgamal cryptosystems, and the ECDSA are tested and validated. We use standard C++ stream socket operating over TCP protocol in order to simulate the ECC applications on a real secp192r1 curve. However, the implementation works on any valid elliptic curve with different parameters of the same size and parameters with different sizes.

In the future work, we plan to optimize our implementations of the big integer classes and point operations. First of all, the performance of the arithmetic operations of the Big Integer classes could be improved by reducing the memory used to store the big integers. Memory consumption can be reduced by representing a big integer as an array of integers. Thus, the 160 bit long integer will require only 5 32-bit integers.

In addition, the execution time of arithmetic operations of the big integer classes could be improved by parallelizing the arithmetic operations using GPU. When the performance of arithmetic operations of the big integer classes is improved, the overall performance of the ECC applications will be also improved.

Additionally, we plan to improve the performance of the point operations by using mixed addition operating on the mixture of both Projective coordinates and Affine coordinates. Finally, we plan to further improve the performance and security of the ECC appli-

cations by implementing Montgomery ladder algorithm for scalar multiplication of points on a given elliptic curve, which helps to defend the ECC applications from side-channel attacks.

Bibliography

[1] Sec 2: Recommended elliptic curve domain parameters), http://www.secg.org/sec2-v2.pdf, 2013.

[2] Asymmetric cryptography (public key cryptography), https://searchsecurity.techtarget.com/definition/asymmetric-cryptography, 2019.

[3] Characteristic (algebra), https://en.wikipedia.org/wiki/characteristic_(algebra), 2019.

[4] std::vector, http://www.cplusplus.com/reference/vector/vector/, 2019.

[5] Owen Astrachan. *THE LARGE INTEGER CASE STUDY IN C++*. College Entrance Examination Board and Educational Testing Service.

[6] A. Dua, N. Kumar, M. Singh, M. S. Obaidat, and K. Hsiao. Secure message communication among vehicles using elliptic curve cryptography in smart cities. In *2016 International Conference on Computer, Information and Telecommunication Systems (CITS)*, pages 1–6, July 2016.

[7] Ravi Ganesan. Computer system for securing communications using split private key asymmetric cryptography, 1996.

[8] R. Harkanson and Y. Kim. Applications of elliptic curve cryptography: A light introduction to elliptic curves and a survey of their applications. In *Proceedings of the*

12th Annual Conference on Cyber and Information Security Research, CISRC '17, pages 6:1–6:7, New York, NY, USA, 2017. ACM.

[9] D. He, H. Wang, M. K. Khan, and L. Wang. Lightweight anonymous key distribution scheme for smart grid using elliptic curve cryptography. *IET Communications*, 10(14):1795–1802, 2016.

[10] Tibor Juhas. The use of elliptic curves in cryptography, 2007.

[11] Avinash Kak. Lecture notes on computer and network security. elliptic curve cryptography and digital rights management, March 2018.

[12] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[13] Sandi Rahmadika and Kyung-Hyune Rhee. Blockchain technology for providing an architecture model of decentralized personal health information. *International Journal of Engineering Business Management*, 10:1847979018790589, 2018.

[14] Harpreet Singh Budwal Ravi Kishore Kodali1 and N.V.S. Prof. Narasimha Sarma. Optimized software implementation of ecc over 192-bit nist curve. JUl 2013.

[15] A. G. Reddy, A. K. Das, E. Yoon, and K. Yoo. A secure anonymous authentication protocol for mobile services on elliptic curve cryptography. *IEEE Access*, 4:4394–4407, 2016.

[16] Anna Tracy Reney Brandy, Naleceia Davis. Encrypting with elliptic curve cryptography. pages 9–17, 07 2010.

[17] Suraj Ketan Samal Rosy Sunuwar. page 4, 12 2015.

[18] Iskandar Setiadi, Atsuko Miyaji, and Achmad Imam Kistijantoro. Elliptic curve cryptography: Algorithms and implementation analysis over coordinate systems. 11 2014.

[19] Anissa Sghaier. Software implementation of ecc using gmp library. 03 2016.

[20] Victor Shoup. *A Computational Introduction to Number Theory and Algebra, Version 2*. 2008.

[21] William Stallings. *Cryptography and network security : principles and practice.* Boston : Pearson, [2011], 2011.

[22] William Stallings. *Cryptography and network security : principles and practice, 7th edition*. Boston : Pearson, [2011], 2011.

[23] R. van Rijswijk-Deij, K. Hageman, A. Sperotto, and A. Pras. The performance impact of elliptic curve cryptography on dnssec validation. *IEEE/ACM Transactions on Networking*, 25(2):738–750, April 2017.

[24] Nithesh venkata Ramana Surya Bommireddipalli. Tutorial on elliptic curve arithmetic and introduction to elliptic curve cryptography, 2017.

[25] Eric W. Weisstein. "monic polynomial." from mathworld–a wolfram web resource., http://mathworld.wolfram.com/monicpolynomial.html, 2019.