The Implementations and Optimizations of Elliptic Curve Cryptography based Applications

Kirill Kultinov[*]
Wright State University, Dayton, Ohio, USA
Meilin Liu [†]
Wright State University, Dayton, Ohio, USA.
Chongjun Wang [‡]
Nanjing University, Nanjing, China.

December 24, 2024

Abstract

Elliptic Curve Cryptography (ECC) represents a promising public-key cryptography system due to its ability to achieve the same level of security as RSA with a significantly smaller key size. ECC stands out for its time efficiency and optimal resource utilization. This paper introduces two distinct new software implementations of ECC over the finite field GF(p), utilizing character arrays and bit sets. Our implementations operate on ECC curves of the form $y^2 \equiv x^3 + ax + b \mod p$.

We have optimized the point addition operation and scalar multiplication on a real SEC (Standards for Efficient Cryptography) ECC curve over a prime field. Furthermore, we have tested and validated the Elliptic Curve 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, and compared and analyzed their performances.

Key Words:Cryptography, ECC, point addition, ElGamal, ECDSA.

1 Introduction

Data security is very crucial for almost any system nowadays [20]. Cryptography is a mathematical tool utilized in software and hardware systems to provide security services, safeguard data and information in storage and transmission against unauthorized access or tampering, and facilitate key exchange between communicating parties. It plays a critical role in various applications. During the early stages of cryptography, symmetric key cryptographic systems [5] were used to encrypt and decrypt messages. Subsequently, public-key cryptography systems [2], including the Diffie-Hellman key exchange system and RSA, were developed in 1976 and 1977, respectively. These systems offered increased security compared to symmetric encryption methods as they were based on number theory, employing two separate keys: the public key and the private key. In contemporary times, public key cryptography holds immense importance as data integrity and confidentiality depend on it. It must ensure forward secrecy, ensuring information that's secure presently remains secure in the future [15]. RSA stands as the most popular public key cryptography algorithm, relying on the complexity of factoring large numbers for security [9]. However, with the advancing computational capabilities of computers, RSA struggles to provide sufficient forward secrecy without exponentially increasing key sizes. Due to the computational overhead of RSA systems with large key sizes, Elliptic Curve Cryptography (ECC), a public-key cryptography system rooted in algebra, gained popularity. ECC, developed in 1985 by Neal Koblitz and Victor Miller and widely adopted since 2005 [7], can achieve the same level of security as RSA but with much smaller key sizes. Table 1 demonstrates the key size comparisons between RSA and ECC for equivalent security levels. ECC stands as a promising public key cryptography system, excelling in time efficiency and resource utilization. The logic behind ECC is enrely unique compared to other cryptographic algorithms. It relies on the challenges associated with solving the discrete logarithm problem through point additions and multiplications

Table 1: Comparable key sizes in terms of computational effort for cryptanalysis

Symmetric Key Size (bits)	RSA Key Size (bits)	ECC Key Size (bits)
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

[*]Department of Computer Science and Engineering , Wright State University, Dayton, Ohio, MO. Email:

[†]Department of Computer Science and Engineering , Wright State University, Dayton, Ohio, MO. Email:meilin.liu@wright.edu .

[‡]Department of Computer Science and Technology ,Nanjing University, Nanjing, China. Email:

on elliptic curves. ECC's popularity continues to grow, finding applications across numerous systems and protocols. One of the most popular applications of ECC is facilitating key exchange between two communication parties. ECC is utilized in a variant of the Diffie-Hellman key exchange known as Elliptic Curve Diffie-Hellman (ECDH). Around 97% of popular websites support ECDH, specifically using Elliptic Curve Diffie-Hellman Ephemeral Elliptic Curve Digital Signature Algorithm (EDHE ECDSA) for key exchange during HTTPS connections [7]. Additionally, ECDSA finds widespread use in blockchain technology [14]. ECC also plays a role in the DNSSEC protocol, a secured version of DNS that shields DNS servers from DDoS attacks [7]. While it's feasible to implement DNSSEC using RSA as a signature algorithm, this approach exposes servers to various potential attacks [7]. Alternatively, Using ECDSA on DNS servers protects them from amplification attacks without requiring packet fragmentation or introducing additional complexities [23]. Mobile devices, and IoT devices have become integral to people's lives. However, they are vulnerable to attackers exploiting various vulnerabilities [4]. Mobile devices, and IoT devices often use embedded processors, require robust security mechanisms [6]. However, public key cryptography algorithms prove computationally expensive due to the computing capabilities and memory constraints of these devices. ECC's efficiency and strong security make it ideal for protecting IoT devices from cyber threats. For instance, a lightweight protocol proposed by a team of researchers leverages elliptic curves, and it is resistant to various attacks like man-in-the-middle and replay attacks [16]. ECC can also be employed in a one-time password (OTP) scheme based on Lamport's OTP algorithm [7]. Finally, ECC could be used in safeguarding smart grids and securing communication channels for autonomous cars [8, 3]. This paper concentrates on the new software implementation of ECC over the finite field GF(p) using character arrays and bit sets in the C++ programming language. Our implementation operates on ECC curves of the form $y^2 \equiv x^3 + ax + b \mod p$.

We have implemented and optimized the core elliptic curve operations, specifically point addition and scalar multiplication, on a real SEC (Standards for Efficient Cryptography) ECC curve over a prime field using two different approaches. In addition, the Elliptic Curve ElGamal encryption/decryption system and Elliptic Curve Digital Signature Algorithm (ECDSA) on a real SEC ECC curve with two different implementations are tested, and validated. The performances of these two different implementations are compared and analyzed. The rest of this paper is organized as follows: Section 2 provides basic background information used in this paper. It introduces the ECC cryptographic system, detailing point addition and point doubling operations. Section 3 describes the detailed implementation of ECC public- key systems on real SEC ECC curves over a prime field using two distinct implementations of the Big Integer objects: character arrays and bit sets. This section elaborates on the design of each component of the ECC system and introduces optimization techniques utilized to improve the efficiency of our implementations. Section 4 presents the experimental results of our ECC implementations in C++ on a Linux Ubuntu OS. It presents a comparison of the timing performance of fundamental operations such as point addition and point doubling using our implementations of Big Integer objects in ECC systems. Additionally, it presents the applications of these implementations in two widely used cryptographic schemes: the Elliptic Curve ElGamal encryption/decryption system and the Elliptic Curve Digital Signature Algorithm (ECDSA) (The implementations and performance comparisons of the ECDH key exchange system have been presented in [12]). These two cryptographic systems are tested and validated on a real SEC ECC curve. The performance of these Elliptic Curve cryptographic systems are compared and analyzed. Finally, Section 5 summarizes the paper and discusses the future work of this paper.

2 Background

In this section, we will introduce basic concepts and background information used in this paper.

2.1 Mathematical Background

Number theory and algebra play crucial roles in cryptography [21]. Cryptography algorithms rely on concepts from number theory, enabling these algorithms to remain secure against various attacks. The logic behind ECC differs significantly from other public-key cryptography algorithms, which can make it challenging to comprehend. In this section, we will introduce fundamental concepts, including ECC, point addition, scalar multiplication on ECC curves, and the applications of ECC.

2.2 ECC Concepts

Elliptic Curve cryptography is based on equations describing elliptic curves and computations involving points that belong to a given curve. In this section, we introduce the concepts of ECC as utilized in cryptography. Initially, we elaborate on the properties and operations of Elliptic curves over real numbers, as vital details can be visually demonstrated using geometry. Subsequently, we describe elliptic curves over GF(p), which are specifically employed in ECC.

2.3 The introduction to ECC

Imagine a large yet finite set E consisting of points on the plane (x_i, y_i) derived from the elliptic curve. Within this set E, we define a group addition operator denoted by $+$, operating on two given points P and Q. This group operator enables the computation of a third point $R \in E$ such that $P + Q = R$.

Given a point $G \in E$, our focus lies in calculating $G + G + G + \cdots + G$ using this group operator. To be specific, for any arbitrary number $k \in Z$, we utilize the notation $k \times G$ to signify the repeated addition of point G to itself k times (the $+$ operator invoked $k - 1$ times). The fundamental concept behind ECC is the complexity involved in retrieving k from $k \times G$. An attacker would need to attempt all possible combinations of repeated additions: $G+G$, $G+G+G$, $G+G+G+\cdots+G$ [10]. This challenge constitutes the discrete logarithm problem, forming the foundation for the security of the ECC algorithm.

2.4 ECC Over Real Numbers

Elliptic curves have no direct connection with ellipses [10]. Instead, they are defined using cubic equations, which are also employed in determining the circumference of an ellipse [22]. These curves commonly adhere to a form known as the Weierstrass equation

The general form of an elliptic curve equation is given by:

$$y^2 + axy + by = x^3 + cx^2 + dx + e \tag{1}$$

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

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

The equation provided pertains to a field of real numbers, wherein the coefficients a and b, along with the variables x and y, are elements of the real number field.

Figure 1 shows examples of elliptic curves drawn from equations with different parameters a and b in equation (2):

Elliptic curves can be singular or non-singular. Figure 1 displays an example of a non-singular elliptic curve. Notice that the curves are smooth. Smooth curves fulfill the discriminant condition of a polynomial $f(x) = x^3 + ax + b$:

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

The elliptic curve described in Equation (2) represents a cubic polynomial, implying it possesses three distinct roots, denoted as r_1, r_2, and r_3. The discriminant is determined by the following formula:

$$D_3 = \prod_{i<j}^{3} (r_i - r_j)^2 \tag{4}$$

If the discriminant is zero, it indicates that two or more roots have merged, rendering the curve non-smooth [10]. Singular curves are unsuitable for cryptographic purposes as they are susceptible to being easily cracked. Therefore, our focus lies solely on non-singular curves. signifying that curves used in ECC algorithms must possess a non-zero discriminant.

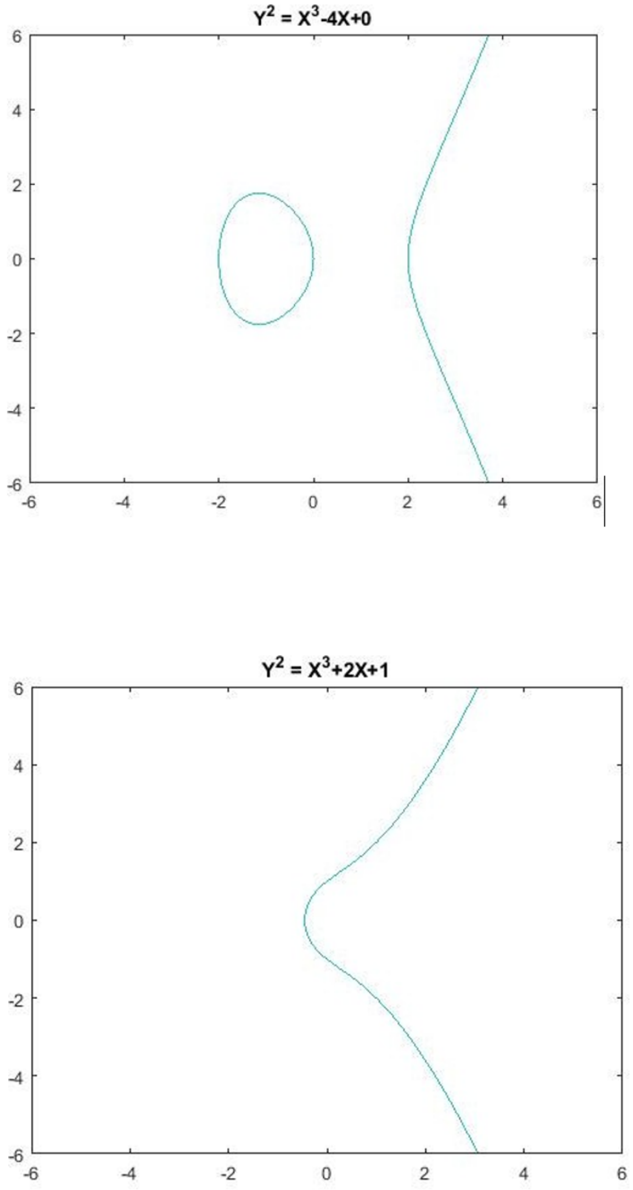


Figure 1: Examples of Elliptic Curves

2.4.1 The group operators in ECC

For an elliptic curve defined by Equation 2, the set of points belonging to the curve is denoted as E (a, b), including a distinguished special point at infinity, represented as O. The set E (a, b) forms an abelian group [10, 22] under a unique addition operator, denoted by +. This addition operator differs significantly from the traditional algebraic addition and is described as follows.

2.4.2 Point addition

Suppose we intend to add a point P to another point Q. This addition process involves the following steps:

1. Draw a straight line connecting points P and Q.

2. Identify the intersection point of the connecting line with the elliptic curve to obtain a third point R.

Consider a point $R = (x, y)$ on the curve. The reflection of R along the x-axis results in a point denoted by $-R = (x, -y)$. This reflection is feasible due to Equation (2), which can be reformulated as $y^2 = x^3 + ax + b$, signifying the curve's symmetry with respect to the x-axis.

Thus, the addition of two points results in P + Q = R, which is visually depicted in Figure 2. However, an exception occurs when the joining line of points P and Q fails to intersect with the elliptic curve. In such instances, we identify this situation as being at the distinguished point at infinity. This circumstance only arises when the line joining P and Q is parallel to the y-axis. The point at infinity allows us to establish the following properties:

• P + O = P: Adding point P with a point at infinity requires us to draw a line parallel to the y-axis. The line intersects the curve at another point, which acts as the mirrored reflection of

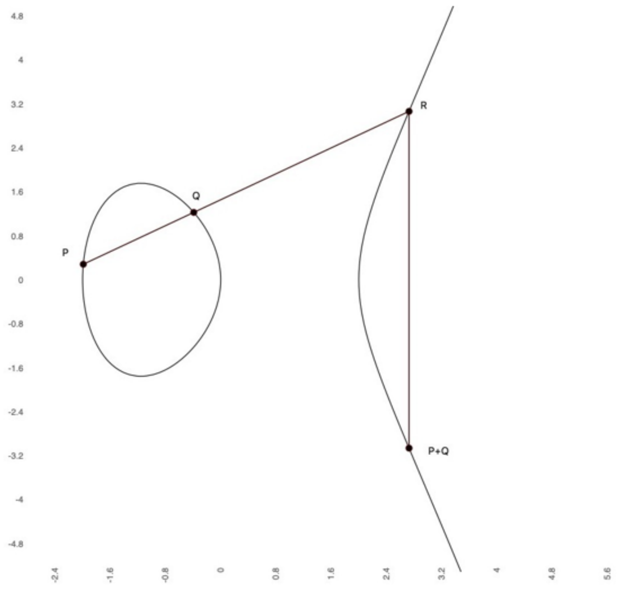


Figure 2: Point Addition Illustration on an Elliptic Curve.

P concerning the x-axis. Consequently, reflecting that additional point P along the x-axis yields point P. Additionally, P serves as the additive inverse of P under the + group operator.

• If P is the mirrored reflection of Q concerning the x-axis, then Let $P = -Q$

$O + O = O$, and $O = -O$.

2.4.3 Point doubling

Elliptic Curve Cryptography (ECC) involves repeatedly adding a point to itself k times to obtain another point denoted as kG. This operation, known as point doubling, is expressed as $P + P = 2P$. Point doubling is similar to the addition of two distinct points P and Q. When adding a point to itself, point Q gradually converges towards point P until they coincide as the same point on the curve. Hence, the computation of $2P$ involves the following steps:

1. Draw a tangent line at point P.

2. Find the point of intersection of the tangent line with the elliptic curve to determine point R.

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

2.5 ECC Over GF(p)

Elliptic curves over real numbers are not well-suited for cryptography. Instead, prime numbers are favored due to the error-free arithmetic they offer in prime fields denoted as Z_p. ECC over $GF(p)$ operates solely with elements from the set $\{0, 1, \ldots, p-1\}$. This indicates that parameters a and b, along with variables x and y, belong to the set $GF(p)$. Furthermore, all operations are conducted modulo p. As a result, the form of the elliptic curve is given by:

$$y^2 \equiv x^3 + ax + b \pmod{p} \tag{5}$$

where the condition (3) is also satisfied in the form:

$$4a^3 + 27b^2 \not\equiv 0 \pmod{p} \tag{6}$$

A collection of points (x, y) on the elliptic curve over $GF(p)$ is represented by $E_p(a, b)$, including a distinguished point at infinity, labeled as O. These points no longer form a continuous curve but instead constitute a set of discrete points on the plane [10]. Consequently, it becomes impractical to visually depict point addition and point doubling geometrically. Nevertheless, all algebraic expressions and properties are valid under the modulo p operation. The primary difference lies in how slopes are calculated for point addition and doubling. In point addition, the slope of a line passing through points P and Q can be found as follows:

$$\alpha = y_Q - y_P x_Q - x_P{}^{-1} \pmod{p} \tag{7}$$

where $(x_Q - x_P)^{-1}$ denotes the multiplicative inverse modulo p. Similarly, the slope of a tangent line for point doubling is calculated as

$$\alpha = 3x_P^2 + a2y_P{}^{-1} \pmod{p} \tag{8}$$

Finally, the set $E_p(a, b)$ forms a group with an addition operator $+$. The prime number p represents the characteristic of the field Z_p. Prime finite fields where $p \leq 3$ are deemed unsafe for cryptographic purposes [10].

2.6 Point Encoding

In most cryptographic systems, it's necessary to transform a plaintext into a value applicable to a particular cryptographic algorithm [22]. The process of mapping messages to points on the elliptic curve is integral to ECC. Specifically, in ECC, converting a message into a point on the elliptic curve precedes the execution of point operations that lead to the generation of ciphertext.

However, there exists a significant challenge in converting a message to a point on the elliptic curve. There isn't a deterministic algorithm for specifying points on a curve over GF(p) [22]. Nonetheless, the Koblitz algorithm [17] provides a solution, enabling the discovery of an appropriate point on the elliptic curve with an exceedingly low probability of error.

For instance, considering an elliptic curve described in Equation (5), the plaintext m, represented as a number, is embedded within the x-coordinate of a point, with additional appended bits. Directly using the message m as the x-coordinate provides only a 50% chance that a square modulo p equals $m^2 + am + b$.

Instead, we select an integer K, which signifies a failure rate of $\frac{1}{2^K}$. The plaintext message must fulfill the following condition:

$$(m+1)^K < p \tag{9}$$

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

$$0 \leq m \leq p - K \tag{10}$$

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

$$x = mK + j \tag{11}$$

where j is within the range $0 \leq j < K$. We then 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 value will represent the y-coordinate. of the point. If we cannot find a square root for all potential j values, it means the given message cannot be mapped to a point on the given elliptic curve. The values obtained from equation (11) and the square root y generate a point $P_m = (x, y)$, which can then be utilized in encryption. To retrieve the plaintext m from the point, we use the following equation:

$$m = \left\lfloor \frac{x}{K} \right\rfloor \tag{12}$$

2.7 The applications of ECC

Repeated additions aren't directly utilized for encryption, as described by $m \times G$ [10]. Instead, the concept of point multiplication is employed in various cryptographic schemes and algorithms. In this subsection, we introduce the applications of ECC, the Elliptic Curve ElGamal cryptosystem, and the ECDSA cryptosystem.

2.8 ElGamal cryptosystem

The ElGamal cryptosystem is an asymmetric key encryption algorithm rooted in the ECDH key exchange systems presented in [12]. ElGamal, similar to RSA, is widely adopted for encryption purposes [22]. It can also be implemented utilizing ECC. The elliptic curve variant of ElGamal operates on points residing on a specified curve over GF(p) and involves repeated point addition operations, distinct from the exponentiation utilized in RSA [18].

Alex and Bob agree on an elliptic curve and a base point $B \in E_p(a,b)$. Alice selects a random large integer $a = 1, 2, \ldots, p-1$ as her private key, and similarly, Bob selects $b = 1, 2, \ldots, p-1$ as his private key. Subsequently, the public keys (p, B, G) are calculated, where $G_A = a \cdot B$ for Alice and $G_B = b \cdot B$ for Bob.

Suppose Alice intends to transmit a message m to Bob. The message is initially encoded into a point P_m. Alice then represents the ciphertext P_c as a pair of points on the curve:

$$P_c = [(a \cdot B), (P_m + a \cdot G_B)] \tag{13}$$

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 (i.e., $a \cdot B$). Then, Bob subtracts this product from the second point of P_c:

$$(P_m + a \cdot G_B) - [b \cdot (a \cdot B)] = P_m + a \cdot (b \cdot B) - b \cdot (a \cdot B) = P_m \tag{14}$$

Finally, Bob can decode the original message from the point P_m using equation (12).

2.9 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 cryptosystemboth 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 × 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 the range $1, 2, \ldots, p-1$. The public key $G_A = aB$ 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 for a message m as follows:

1. **Calculate** $e = \text{HASH}(m)$
   This step involves applying a cryptographic hash function to the message m to generate a hash value e. The hash function is typically SHA-256 or similar.
2. **Obtain value z by extracting the leftmost L_n bits of e, where L_n is the bit length of the group order n**
   The value z is extracted by truncating the hash e to L_n bits, where L_n is the bit length of the order n of the elliptic curve group.

   $$z = \text{leftmost } L_n \text{ bits of } e$$

3. **Select a cryptographically secure random integer k in the range** $\{1, 2, ..., n-1\}$
   A random integer k is selected in the range from 1 to $n-1$, which will be used for the point multiplication operation on the elliptic curve.
4. **Calculate a point** $(x_1, y_1) = k \cdot B$
   The point (x_1, y_1) is calculated by performing the point multiplication of the random integer k with the base point B of the elliptic curve.
5. **Calculate** $r = x_1 \mod n$
   The value of r is derived by taking the x-coordinate of the point (x_1, y_1) and reducing it modulo the order n. If $r = 0$, the process repeats by selecting a new k.

$$r = x_1 \mod n$$

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 $1, 2, \ldots, n-1$. If at least one number does not satisfy this condition, then the signature is invalid.
2. Calculate $e = \text{HASH}(m)$ using the hashing algorithm identical to the one used by Alice during the signature generation process.
3. Identical to the signature generation process, obtain value z by extracting the 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 $u_1 = zs^{-1} \mod n$ and $u_2 = rs^{-1} \mod n$.
6. Calculate a point $(x_1, y_1) = u_1 B + u_2 G_A$. If $(x_1, y_1) = O$, then the signature is invalid.
7. If $r \equiv x_1 \mod n$, then the signature is valid. Otherwise, the signature is invalid.

2.10 Jacobian Projective Coordinates

As discussed in the previous sections, when points are represented in affine coordinates, the operations on the elliptic curve involve arithmetic additions, subtractions, multiplications, squaring, and the computation of modulo multiplicative inverses. As we are dealing with elliptic curves over GF(p), calculating multiplicative inverses, crucial for point addition and doubling operations requiring the calculation of slopes, is a fundamental process, as seen in Equations 7 and 8. The calculation of multiplicative inverses is computationally intensive, especially when involved in point multiplication, which necessitates multiple point addition and multiplication operations. Given this computational cost, representing elliptic curve points in projective coordinates, particularly in the Jacobian projective coordinate system, proves practical.

Utilizing Jacobian coordinates can significantly enhance the performance of ECC algorithms by reducing the number of computations involving multiplicative inverses on large integers [19].

A point represented in Affine coordinates (x, y) can be transformed into Jacobian coordinates (X, Y, Z). For instance, a point P with Affine coordinates (x_P, y_P) can be depicted in Jacobian coordinates as $(X, Y, Z) = (x_P, y_P, 1)$.

Conversely, a point expressed in Jacobian coordinates (X, Y, Z) can be converted back to Affine coordinates using the following 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 intend to add a point P with coordinates (X_P, Y_P, Z_P) to another distinct point Q with coordinates (X_Q, Y_Q, Z_Q). Initially, we define variables A, B, C, and D as described by the following equations:

$$A = X_P \cdot Z^2$$

$$B = Y_P \cdot Z^3$$

$$C = X_Q \cdot Z^2 - A$$

$$D = Y_Q + Z^3 - B$$

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

$$X_R = -C^3 - 2A \cdot C^2 + D^2$$

$$Y_R = -B \cdot C^3 + D(A \cdot C^2 - x_R)$$

$$Z_R = Z_P \cdot Z_Q \cdot C$$

When performing the 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:

$$A = 4X_P \cdot Y^2$$

$$B = 3X^2 + a \cdot Z^4$$

$$C = -2A + B^2$$

The coordinates of the point R are determined using the following equations:

$$X_R = C$$

$$Y_R = -8Y^4 + B \cdot (A - C)$$

$$Z_R = 2Y_P \cdot Z_P$$

3 ECC Implementations

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 shows these components

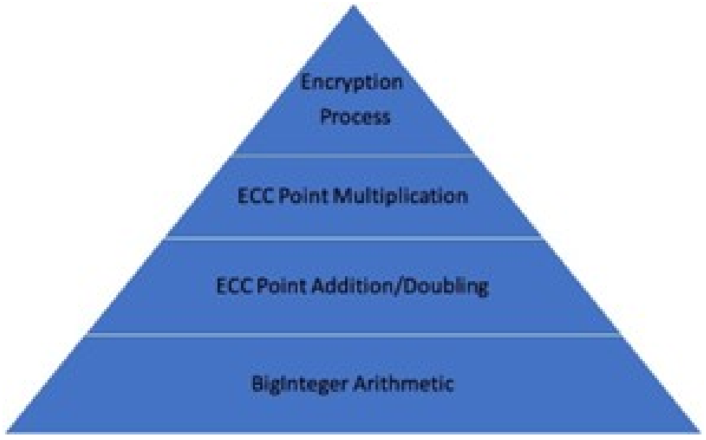


Figure 3: ECC Components Pyramid.

Encryption algorithms utilizing ECC properties rely on scalar multiplication, which combines point addition and doubling techniques. This operation requires handling big integers, as standard primitive data types are limited to 64-bit values. Big integer arithmetic is essential for representing plaintext messages as points on an elliptic curve, forming the foundation for ECC arithmetic and point operations. This section initially introduces algorithms and data structures within our custom Big Integer class. It then demonstrates implementations of elliptic curve point addition, doubling, and multiplication, utilizing two distinct Big Integer object implementations: character arrays and bit sets. Additionally, it illustrates the workings of the ElGamal encryption/decryption algorithms and the ECDSA cryptosystems, highlighting design choices and considerations made during the ECC implementations. Our demonstrations use a real SEC (Standards for Efficient Cryptography) ECC curve over a prime field, specifically the secp192r1 curve, with its parameters presented in Table 2 [1]. However, our implementation is compatible with any valid elliptic curve over GF(p).

3.1 Big Integer Class Implementation

Arithmetic operations involving large integers form the foundation of all arithmetic in public cryptography. To explore and potentially

enhance performance, we've developed our own Big Integer class. This class aims for flexibility by allowing users to provide implementations for Big Integer classes specific to elliptic

c

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

Table 2

curves. In our research for the master's thesis, we implemented the Big Integer class using character arrays and bit sets. The first Big Integer class utilizes a character array to represent each digit of a large number. Conversely, the second class employs an array of Boolean values to store the binary representation of integers, specifically using bit sets. Before implementing the Big Integer class, determining the most suitable data structure for representing large integers was essential. While considering linked lists as an option, their O(n) complexity for element access and the performance overhead introduced by node pointers were noted. Vectors from the standard C++ library, though providing ease of use and rich functionality, lacked control over dynamic array size changes during runtime. Considering these factors, arrays emerged as the optimal choice for faster performance, typically associated with primitive data types. The next consideration was determining the ideal data type for the array to hold. We chose the char data type to represent each digit of a big integer (0-9). Using the int data type wasn't memory-efficient due to its 32-bit occupancy per value. Alternatively, representing big integers as an array of long values might also be feasible. For instance, a 320-bit integer could potentially be stored in an array of long values with a size of 5. Additionally, the order of storing digits in the array needed consideration. While arithmetic operations often use the most significant bit (MSB) fashion, accessing the least significant bit (LSB) first is typically required. Hence, we store digits in the LSB format, simplifying value printing but somewhat limiting flexibility. Despite these considerations, our implementation allows for easy use of any elliptic curve by modifying only the parameters of the elliptic curve equation 5. The character array proves suitable for our goals, offering flexibility in working with ECC parameters of varying sizes and maintaining relative efficiency. The second implementation using bit sets is explored due to advantages in implementing arithmetic operations like addition and multiplication without data dependencies, while division and exponentiation use algorithms requiring fewer data manipulations. Each integer bit is stored as a Boolean value in a separate index of the array. Given that each Boolean value occupies 8 bits of memory, we intended to balance speed and memory. Similar to the character array version, numbers are stored in LSB format. Both implementations of the Big Integer class support all arithmetic operations, including addition, subtraction, multiplication, division, modulus, and modulo exponentiation. Additionally, comparisons and shift operators are implemented for each version of the Big Integer class, expanding their utility beyond cryptography. Arithmetic operations on elliptic curve points are essential for ECC applications. Scalar multiplication, involving repeated point addition, is crucial for forward secrecy. Our Point class, representing x and y coordinates of a point using Big Integer objects, implements add(), double(), and multiply() public member functions [11]. In this section, we illustrate ECC point operations in detail. Due to the space limit, the pseudocode algorithms are not presented in this paper.

1. **Big Integer Addition:** The addition operation involving big integers is one of the most crucial and fundamental operations in ECC. To support the addition operation, we overload the $+$ addition operator for the purpose of adding two Big Integer objects together. This operator takes two Big Integer objects, performs the addition operation, and returns the result as a Big Integer object.
2. **Big Integer Subtraction:** Our implementation of the Big Integer class supports subtraction operations by overloading the subtraction operator. As previously mentioned, addition may involve numbers with different signs. Therefore, we can convert subtraction into an addition operation. Specifically, the operation $A - B$ is transformed into $A + (-B)$, which calls the overloaded $+$ operator. Internally, `if-else` conditions are utilized to invoke either the `add()` or `subtract()` wrapper function within the addition operator function.
3. **Big Integer Multiplication:** The multiplication operation on two big integers is executed using the overloaded $*$ operator. Unlike other arithmetic operations previously discussed, multiplication doesn't necessitate the use of conditional statements to account for all potential cases regarding sizes and signs of the operators. However, the multiplication operation tends to be the most memory-intensive because it requires constructing an array of size $m \times n$, where m and n are the sizes of the first and second big integers, respectively. Additionally, there are two notable extreme cases to consider: when one of the operands is zero, resulting in a zero result, and when one of the operands is one, yielding the other operand as the result. For all remaining cases, multiplication logic similar to manual multiplication must be implemented.
4. **Big Integer Division Operations:** Division and modulo operations are closely related. As with all other mathematical operations in our Big Integer class, the operators $/$ and % are overloaded. For most cases, division operations entail manipulating the digits of both numbers. While repeated subtraction could be an option, it proves to be inefficient. Hence, we implement the long division algorithm within a `divide()` wrapper function, which is then invoked within the overloaded $/$ operator.
5. **Big Integer Modulo Operations:** The modulo operation relies on the division operation, implemented within the overloaded % operator, utilizing wrapper member functions described earlier in this section. Hence, it is compatible with both versions of the big integer classes.
6. **Big Integer Modulo Exponentiation:** The exponentiation operation is a fundamental part of numerous algorithms in ECC. Fundamentally, exponentiation involves repeatedly multiplying a number by itself.However, this method can overwhelm system resources, especially when handling large numbers [22]. As an alternative, we implement a repeated squaring algorithm, which involves a maximum of n multiplications, where n represents the length of the exponent in bits.

3.2 ElGamal Implementation

The ElGamal cryptosystem is utilized for encrypting and decrypting with symmetric keys. We have developed a sample program that simulates the ElGamal encryption and decryption process between two parties employing the secp192r1 elliptic curve. The complete process, encompassing encryption and decryption, is illustrated in Figure 4.

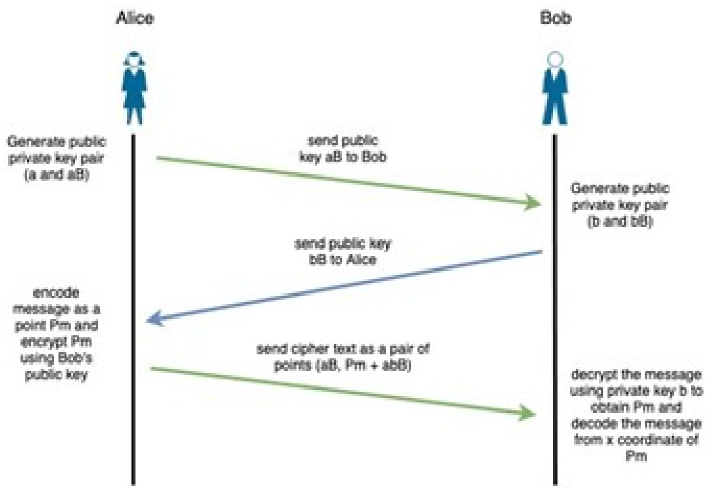

Alice
Bob
Generate public private key pair (a and aB)
send public key aB to Bob
Generate public private key pair (b and bB)
send public key bB to Alice
encode message as a point Pm and encrypt Pm using Bob's public key
send cipher text as a pair of points (aB, Pm + abB)
decrypt the message using private key b to obtain Pm and decode the message from x coordinate of Pm

Figure 4: Message exchange using the ElGamal cryptosystem.

Suppose Alice intends to send a message to Bob. We assume that both Alice and Bob have agreed upon a curve and a base point, denoted as B. Each party calculates their respective private keys. In our implementation, we utilize randomly generated numbers. The public key is obtained through scalar multiplication of the base point. Alice's public key is calculated using the equation $G_A = a \cdot B$. Similarly, Bob's public key is derived using the equation $G_B = b \cdot B$. Once Alice and Bob exchange their public keys, Alice can securely transmit a message to Bob.

Alice encodes the message as a point P_m on the curve and encrypts it using Bob's public key, employing equation 13. Upon receiving the ciphertext, Bob can recover the plaintext by decrypting the ciphertext using equation 14 and decoding the message using equation 12.

Similar to the implemented ECDH key exchange mechanism [12], in the established communication channel using sockets, Bob assumes the role of a server while Alice acts as a client. The pseudocode for the ElGamal cryptosystem on the client side is depicted in Algorithm 1.

The code segment including lines 1 to 3, is primarily used to initialize the socket for subsequent communication. The client establishes a connection with the server, which runs on the local host, depicted in line 5. Alice computes her public key via the point multiplication algorithm and transmits it to Bob. Upon receiving Bob's public key, obtained by reading a message from the socket (as shown in line 11), Alice proceeds to encode a message as a point on the chosen curve. Subsequently, she encrypts this message utilizing the function `encrypt()`, implementing the logic specified by equation 13, involving point addition and multiplication operations.

Algorithm 2 illustrates the ElGamal cryptosystem's server-side implementation. In this algorithm, lines 1-10 are dedicated to configuring a socket and initiating the listening process for incoming messages through the stream socket. Subsequently, Bob generates a private key using a built-in random number generator and computes his public key, which he then transmits to Alice. Subsequent incoming messages are regarded as ciphertext sent by Alice. Bob decrypts this ciphertext using the procedure detailed in Equation 14. The decryption process is demonstrated in the pseudocode below. The resulting decrypted message is passed to the decodeMessage () function, designed to handle a point embedding the encoded message within its x-coordinate. If Bob intends to send an encrypted message to Alice, he follows the procedure outlined in lines 12-14 of Algorithm 1, albeit using Alice's public key. ciphertext constitutes a pair of points on an elliptic curve within the ElGamal cryptosystem.

Algorithm 1: 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();
```

As indicated in Equation 14, Bob's task is to compute a point that results from the product of the first point within the ciphertext pair and his private key.

Algorithm 2: 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, clientAddress);
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);
```

Algorithm 3: 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;
```

This operation is executed straightforwardly through point multiplication. Subsequently, Bob accomplishes the subtraction by addition of the negative point to the first point of the ciphertext pair since a direct point subtraction operation isn't supported. As previously discussed in Section 2, obtaining the negative of a point entails reflecting the same point across the x-axis. However, in elliptic curves over GF(p), simply altering the sign of the y-coordinate is inadequate. Modulo operation is also employed in line 2 of Algorithm 3. Ultimately, the point obtained in line 1 is combined with the second point of the ciphertext pair. The encoded message is then recovered, wherein the plaintext resides within the x-coordinate and can be retrieved employing Equation 12.

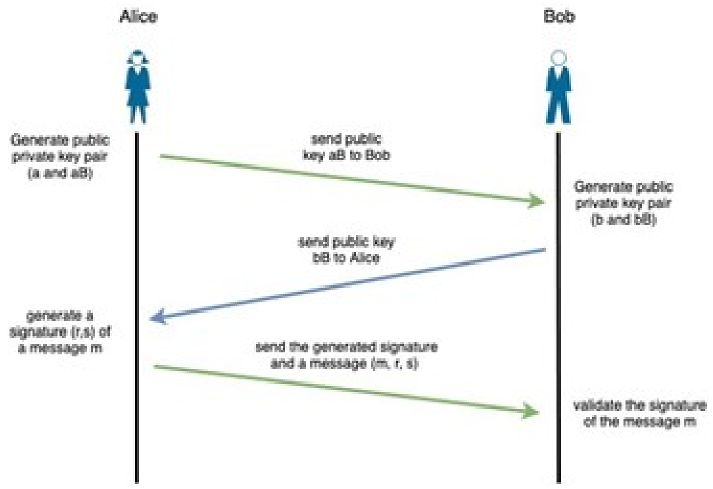

Alice
Bob
Generate public private key pair (a and aB)
send public key aB to Bob
Generate public private key pair (b and bB)
send public key bB to Alice
generate a signature (r,s) of a message m
send the generated signature and a message (m, r, s)
validate the signature of the message m

Figure 5: Message exchange using ECDSA.

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 $GA = aB$, where a is her private key. Similarly, Bob calculates his public key $GB = bB$, 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 is able to verify the integrity and authenticity of the message using the procedure described in Section 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 4 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 5 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 point multiplication operation to get an intermediate point. We obtain the first value of the signature

Algorithm 4: The pseudocode for ECDSA on the client side

```
Input: message
Output: signedMessage
1. sockfd = socket(AF$_{I}NET, SOCK_{S}TREAM, 0$);
2. if sockfd <0 then
3.     print(”Error opening socket”);
4. end
5. servername = gethostbyname(”localhost”);
6. connect(sockfd, serveraddress);
7. if not connected then
8.     print(”Error connecting to the server”);
9. end
10. privKey = genPrivateKey();
11. pubKey = privKey * BasePoint;
12. write(sockfd, pubKey);
13. serverPubKey = read();
14. signature = sign(message);
15. write(sockfd, message, signature);
```

Algorithm 5: 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;
11.    end
12.    kInverse = gcdExtend(k, n);
13.    s = kInverse * (z + r * privateKey) mod n;
14.    if s = 0 then
15.        return pair(r, s);
16.    end
17. return pair(r, s);
```

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.

Algorithm 6: 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, clientAddress);
10. clientPubKey = read();
11. privKey = genPrivateKey();
12. pubKey = privKey * BasePoint;
13. write(sockfd, pubKey);
14. clientMessage = read();
15. verify(clientMessage.signature, clientMessage.message);
```

The server side implementation is similar to the previously described implementations of ECC application, ElGamal. Algorithm 6 shows 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 7 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 i f 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 u1 and u2 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 n1 and n2 used in the final step of signature verification process. If these two values are identical, then the signature is valid and the veri f y () function returns true. Otherwise, false Boolean value is returned.

Algorithm 7: 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
```

4 Evaluation

In this section, we conduct a performance evaluation of the arithmetic operations performed by the Big Integer classes on operands of varying sizes. Additionally, we analyze the point operations essential for all applications of ECC on the secp192r1 curve.

4.1 Platforms

All arithmetic and point operations underwent testing on a PC equipped with a quad-core Intel(R) Core(TM) i7-7700K CPU running the Ubuntu 15.04 operating system. The execution times were measured as part of the testing process. The software program, developed for this evaluation, was compiled and executed using the standard GNU C++ compiler version 4.9.2. Additionally, the program underwent memory-related error checks using the Valgrind dynamic analysis tool [13].

4.2 Experimental Results

This subsection outlines the experimental findings in which we compare the time performance of arithmetic operations, including addition, subtraction, division, multiplication, and modulo exponentiation, performed using the Big Integer classes. Additionally, we conducted measurements to report the execution times for point addition, point doubling, and scalar multiplication operations over the secp192r1 curve. These operations were utilized in the implementation of both the Elliptic Curve ElGamal cryptosystem, and the ECDSA cryptosystem. The program underwent 20 executions, and the average running time for these operations is presented

4.3 Big Integer Arithmetic Operations

The execution time of multiple arithmetic operations is measured using operands of various sizes. Each operation's timing performance is compared between the two versions of the Big Integer classes: one implemented using an array of characters and the other using an array of Boolean values.

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 3: Comparison of performance: Addition Operation

Table 3 presents a comparison of the average execution time for the addition operation between the two versions of the Big Integer classes across various operand sizes. Each operand represents a randomly generated number of the size specified in the first column of the table. Notably, the results show that adding two 192-bit or 256-bit long numbers is marginally faster than adding two 160-bit long numbers in both implementations. However, the precise reason for this observation is challenging to determine. Moreover, the arithmetic operations of the Big Integer class implemented using a bit set exhibit slower performance across all operand sizes. This disparity in performance can be attributed to the larger number of loop iterations in the algorithm utilizing a bit set compared to the algorithm using arrays of characters. Additionally, the memory required to represent a specific big integer using a bit set is larger than that needed for the same big integer represented with a character array. This is due to the internal storage in C++, where every bit in the bit set is allocated one byte. Interestingly, the smallest difference in average execution time occurs when adding two 192-bit integers, which presents another challenge to explain. Please note that this research paper does not include a comparison of the average execution time for other mathematical operations.

4.4 Point Operations on the Secp192r1 Curve

The performance of essential operations—point addition, point doubling, and scalar multiplication—in ECC applications is detailed in Table 4. Among these operations, point addition proves to be the fastest, taking approximately 7 ms to execute. On the other hand, point doubling requires 3.5 times more time as it involves more computationally expensive tasks like multiplication and exponentiation. However, scalar multiplication emerges as the most resource- intensive point operation. It includes both point addition and point doubling operations. In this operation, a given point undergoes doubling at least n times, where n represents the size of the scalar multiplier in bits. Comparing the two implementations of the Big Integer classes, the point operations performed using a bit set are observed to be twice as slow as those conducted using character arrays.

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

Table 4: Comparison of performance: Point Operations on the curve secp192r1

Given that the Big Integer implementation outperforms the Bitset, we leverage the Big Integer implementation to contrast the performance between Affine and Jacobian coordinates. Table 5 displays the performance comparisons of point operations between the implementation employing Affine coordinates and the one utilizing Jacobian projective coordinates.

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

Table 5: Comparison of performance: Affine vs.Jacobian coordinates on the secp192r1 curve

The table illustrates a slight decrease in performance during the point addition operation. However, the point doubling operation displays nearly three times faster performance. This discrepancy arises due to the significantly reduced number of arithmetic operations involving Big Integers when employing Jacobian coordinates. Consequently, this optimization leads to a substantial enhancement in the efficiency of the point multiplication operation.

4.5 Verification of the Correctness

We have successfully implemented the Elliptic Curve ElGamal and the ECDSA cryptosystems and verified the correctness of these implementations as demonstrated below.We verified the correctness of the ElGamal cryptosystem by simulating the encryption and decryption process from Alice

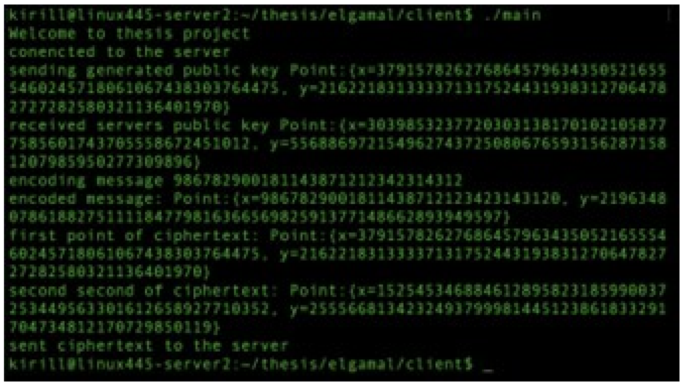
Table 6: Parameters of the ElGamal and the intermediate results of the ElGamal Cryptosystems

Parameter	Value
Message m	986782900181143871212342314312
P_m	x : 9867829001811438712123423143120 y : 2196348078618827511118477981636656982591377148662893949597
P_1 of ciphertext	x : 3791578262768645796343505216555460245718061067438303764475 y : 2162218313333713175244319383127064782727282580321136401970
P_2 of ciphertext	x : 152545346884612895823185990037253449563301612658927710352 y : 2555668134232493799981445123861833291704734812170729850119
Decrypted P_m	x : 9867829001811438712123423143120 y : 2196348078618827511118477981636656982591377148662893949597
Plaintext from P_m	986782900181143871212342314312

Table 7: Parameters and the Intermediate Results of ECDSA

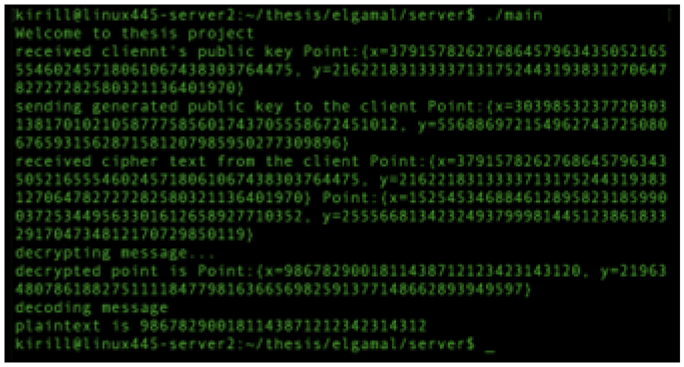
Parameter	Value
Message m	89382075487284788345345
HASH(m)	185F8DB32271FE25F561A6FC938B2E264306EC304EDA518007D17 64826381969
z in hex	185F8DB32271FE25F561A6FC938B2E264306EC304EDA5180
z in decimal	597630496134934525062152428636758271059776916513804145024
Generated r	1131376258843917720091875844748311029151964753646636471475
Generated s	4357797412442008277179215604970751649941568938148867756195
u_1	3046439475643938091811248233621120317830886743790315112337
u_2	4568854499746066067863265371343606136890756961925481503622
Calculated $r \mod n$	1131376258843917720091875844748311029151964753646636471475
$x_1 \mod n$	1131376258843917720091875844748311029151964753646636471475

to Bob, with Bob acting as the server and Alice as the client. Figures 6 and 7 display the output from the encryption and decryption processes on the client and server sides, respectively. Both parties successfully exchanged their public keys.Alice encoded a sample message as a number and encrypted it using the ElGamal encryption algorithm, representing the message as a point on the elliptic curve. The resulting ciphertext, a pair of points on the curve, was transmitted to Bob. Upon receiving the ciphertext, Bob decrypted the message and recovered the plaintext by decoding the embedded message within the x-coordinate of the point. In figure 7, the recovered plaintext by Bob matches the original message encoded and encrypted by Alice. This verifies the correctness of our implemented ElGamal cryptosystem.



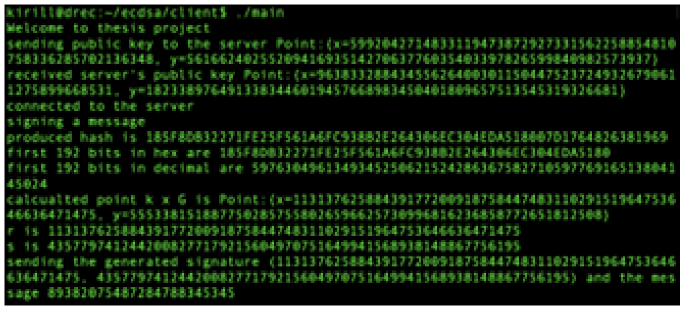
```
kirill@linux445-server2:~/thesis/elgamal/client$ ./main
Welcome to thesis project
conencted to the server
sending generated public key Point:{x=379157826276864579634350521655
5460245718061067438303764475, y=216221831333371317524431938312706478
2727282580321136401970}
received servers public key Point:{x=3039853237720303138170102105877
7585601743705558672451012, y=556886972154962743725080676593156287158
1207985950277309896}
encoding message 986782900181143871212342314312
encoded message: Point:{x=9867829001811438712123423143120, y=2196348
078618827511118477981636656982591377148662893949597}
first point of ciphertext: Point:{x=37915782627686457963435052165554
60245718061067438303764475, y=21622183133337131752443193831270647827
27282580321136401970}
second second of ciphertext: Point:{x=152545346884612895823185990037
253449563301612658927710352, y=2555668134232493799981445123861833291
704734812170729850119}
sent ciphertext to the server
kirill@linux445-server2:~/thesis/elgamal/client$ _
```

Figure 6: Client side of ElGamal Cryptosystem



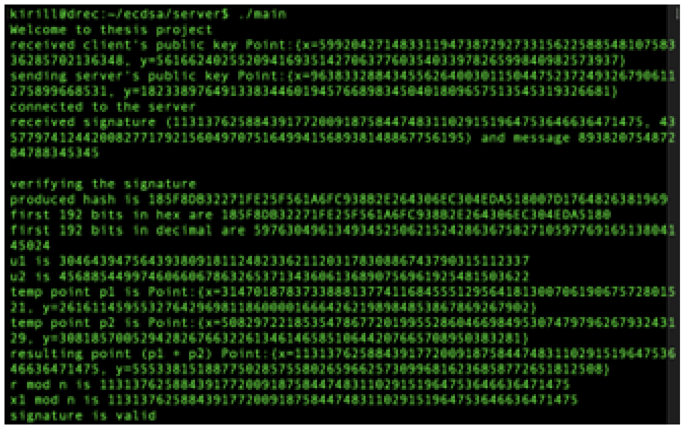
```
kirill@linux445-server2:~/thesis/elgamal/server$ ./main
Welcome to thesis project
received cliennt's public key Point:{x=37915782627686457963435052165
55460245718061067438303764475, y=21622183133337131752443193831270647
82727282580321136401970}
sending generated public key to the client Point:{x=30398532377203O3
1381701021058777585601743705558672451012, y=55688697215496274372508O
67659315628715812079859502773O9896}
received cipher text from the client Point:{x=379157826276864579634 3
505216555460245718061067438303764475, y=2162218313333713175244319383
127064782727282580321136401970} Point:{x=152545346884612895823185990
037253449563301612658927710352, y=2555668134232493799981445123861833
2917047348121707298501 19}
decrypting message...
decrypted point is Point:{x=986782900181143871212342314312 0, y=21963
480786188275111184779816366569825913771486628939495 97}
decoding message
plaintext is 986782900181143871212342314312
kirill@linux445-server2:~/thesis/elgamal/server$ _
```

Figure 7: Server side of ElGamal Cryptosystem



```
kirill@drec:~/ecdsa/client$ ./main
Welcome to thesis project
sending public key to the server Point:{x=59920427148331119473872927331562258854810
758336285702136348, y=5616624025520941693514270637760354033978265998409825739 37}
received server's public key Point:{x=963833288434556264003011504475237249326790 61
1275899668531, y=182338976491338344601945766898345040180965751354531932668 1}
connected to the server
signing a message
produced hash is 185F8DB32271FE25F561A6FC93882E264306EC304EDA518007D1764826381969
first 192 bits in hex are 185F8DB32271FE25F561A6FC93882E264306EC304EDA5180
first 192 bits in decimal are 5976304961349345250621524286367582710597769165138041
45024
calcualted point k x G is Point:{x=111313762588439177200918758447483110291519647536
46636471475, y=5553381518877502857558026596625730996816236858772651812508}
r is 11131376258843917720091875844748311029151964753646636471475
s is 4357797412442008277179215604970751649941568938148867756195
sending the generated signature (111313762588439177200918758447483110291519647536 46
636471475, 4357797412442008277179215604970751649941568938148867756195) and the mes
sage 893820754872847883453 45
```

Figure 8: Client side of ECDSA



```
kirill@drec:~/ecdsa/server$ ./main
Welcome to thesis project
received client's public key Point:{x=599204271483311194738729273315622588548107583
36285702136348, y=5616624025520941693514270637760354033978265998409825739 37}
sending server's public key Point:{x=963833288434556264003011504475237249326790611
275899668531, y=18233897649133834460194576689834504018096575135453193266 81}
connected to the server
received signature (11131376258843917720091875844748311029151964753646636471475, 43
57797412442008277179215604970751649941568938148867756195) and message 893820754872
84788345345

verifying the signature
produced hash is 185F8DB32271FE25F561A6FC93882E264306EC304EDA518007D1764826381969
first 192 bits in hex are 185F8DB32271FE25F561A6FC93882E264306EC304EDA5180
first 192 bits in decimal are 5976304961349345250621524286367582710597769165138041
45024
u1 is 3046439475643938091811248233621120317830886743790315112337
u2 is 4568854499746066067863265371343606136890756961925481503622
temp point p1 is Point:{x=3147018783733888137741168455512956418130070619067572801 5
21, y=2616114595532764296981186000016664262198984853867869267902}
temp point p2 is Point:{x=5082972218535478677201995528604669849530747979626793243 1
29, y=3081857005294282676632261346146585106442076657089508328 1}
resulting point (p1 + p2) Point:{x=111313762588439177200918758447483110291519647536
46636471475, y=5553381518877502857558026596625730996816236858772651812508}
r mod n is 11131376258843917720091875844748311029151964753646636471475
x1 mod n is 11131376258843917720091875844748311029151964753646636471475
signature is valid
```

Figure 9: Server Side of ECDSA

Table 6 reports the parameters utilized by the ElGamal cryptosystem, along with details about the plaintext message, intermediate results, and the recovered plaintext obtained during the ElGamal encryption/decryption process. Similar to the ElGamal cryptosystems, we also verified the correctness of the ECDSA cryptosystems. In our implementation, Alice acts as a client and Bob acts as a server. Figures 8 and 9 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 8 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 values of the calculated intermediate parameters presented in [12], are required for verifying the validity of the digital signature. Most importantly, we see that r mod n and x1modn are also equivalent. This means that the digital signature is valid and the implemented ECDSA cryptosystem is correct. Table 7 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 x1 mod n are also equivalent.

References

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 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. doi: 10.1109/CITS.2016.7546385

4 P. Emami-Naeini, J. Dheenadhayalan, Y. Agarwal, and L. F. Cranor. Are consumers willing to pay for security and privacy of IoT devices? In 32nd USENIX Security Symposium (USENIX Security 23), pages 1505–1522, Anaheim, CA, Aug. 2023. USENIX Association.

5 R. Ganesan. Computer system for securing communications using split private key asymmetric cryptography, 1996.

6 E. Griffor, C. Greer, D. Wollman, and M. Burns. Framework for cyber-physical systems: Volume 1, overview, 2017.

7 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. ISBN 978-1-4503-4855-3.

8 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. ISSN 1751-8628. doi: 10.1049/iet-com.2016.0091

9 T. Juhas. The use of elliptic curves in cryptography, 2007.

10 A. Kak. Lecture notes on "computer and network security". elliptic curve cryptography and digital rights management, March 2018.

11 K. Kultinov. Software implementations and applications of elliptic curve cryptography, 2017.

12 M. Liu, K. Kultinov, and C. Wang. The implementations and applications of elliptic curve cryptography. In 39th International Conference on Computers and Their Applications (CATA), New Orleans, LA, 2024.

13 N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. SIGPLAN Not., 42 (6):89–100, June 2007. ISSN 0362-1340.

14 S. Rahmadika and K.-H. Rhee. Blockchain technology for providing an architecture model of decentralized personal health information. International Journal of Engineering Business Management, 10:1847979018790589, 2018.

15 H. S. B. Ravi Kishore Kodali1 and N. Prof. Narasimha Sarma. Optimized software implementation of ecc over 192-bit nist curve. JUl 2013

16 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

17 A. T. Reney Brandy, Naleceia Davis. Encrypting with elliptic curve cryptography. pages 9–17, 07 2010.

18 S. K. S. Rosy Sunuwar. page 4, 12 2015. URL https://cse.unl.edu/ ssamal/crypto/EEECC.pdf.

19 I. Setiadi, A. Miyaji, and A. I. Kistijantoro. Elliptic curve cryptography: Algorithms and implementation analysis over coordinate systems. 11 2014.

20 A. Sghaier. Software implementation of ecc using gmp library. 03 2016.

21 V. Shoup. A Computational Introduction to Number Theory and Algebra, Version 2. 2008

22 W. Stallings. Cryptography and network security : principles and practice, 7th edition. Boston : Pearson, [2011], 2011. ISBN 9780134444284.

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.

Authors

Kirill Kultinov received his master's degree and bachelor's degree in Computer Science from the Department of Computer Science and Engineering at Wright State University, Dayton, OH, United States, in 2019 and 2017, respectively. He is currently working as a Cyber Security Engineering Expert at Elektrobit Company. His research interests include cryptography, ECC, security risk analysis, and security vulnerability analysis.

Meilin Liu is an associate professor at the department of Computer Science and Engineering at Wright State University. She received her Ph.D. degree in Computer Science from The University of Texas at Dallas in 2006. Her research interests include optimizing compiler for specific architectures, parallel computing, GPU computing, embedded systems, and information security.

Chongjun Wang is a full Professor at the Department of Computer Science and Technology at Nanjing University. He received his Ph.D in Computer Science from Nanjing University, China in 2004. His research interests are Intelligent Agent and Multi-Agent Systems, Complex Network Analysis, Big Data and Intelligent Systems.