EPiC Series in Computing
Volume 98, 2024, Pages 89–102
Proceedings of 39th International Conference on Computers and Their Applications

The Implementations and Applications of Elliptic Curve Cryptography

Meilin Liu[1], Kirill Kultinov[1], and Chongjun Wang[2]

[1] Department of Computer Science and Engineering, Wright State University, Dayton, OH, USA
meilin.liu@wright.edu, kirillkultinov@gmail.com
[2] Department of Computer Science and Technology, Nanjing University, Nanjing, China
chjwang@nju.edu.cn

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 novel implementations operate on ECC curves of the form $y^2 = x^3 + ax + b \ (mod\ p)$. 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 Diffie-Hellman key exchange on a real SEC ECC curve using two different implementations of big integer classes. The performances of the two different implementations are compared and analyzed.

1 Introduction

Data security is very crucial for almost any system nowadays [1]. 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 [2] were used to encrypt and decrypt messages. Subsequently, public-key cryptography systems [3], 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 [4]. RSA stands as the most popular public key cryptography algorithm, relying on the complexity of factoring large numbers for security [5]. 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 [6], 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 entirely unique compared to other cryptographic algorithms. It relies on the challenges associated with solving the discrete logarithm problem through point additions and multiplications 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

Table 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

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 incorporate ECDH, specifically using Elliptic Curve Diffie-Hellman Ephemeral Elliptic Curve Digital Signature Algorithm (EDHE_ECDSA) for key exchange during HTTPS connections [6]. Additionally, ECDSA finds widespread use in blockchain technology [7]. ECC also plays a role in the DNSSEC protocol, a secured version of DNS that shields DNS servers from DDoS attacks [6]. While it's feasible to implement DNSSEC using RSA as a signature algorithm, this approach exposes servers to various potential attacks [6]. Alternatively, employing the ECDSA (Elliptic Curve Digital Signature Algorithm) on DNS servers protects them from amplification attacks without necessitating packet fragmentation or other complexities [8].

Mobile devices have become integral to people's lives and possess networking capabilities, making them vulnerable to attackers exploiting different vulnerabilities. Securing these devices is paramount. However, public key cryptography algorithms prove computationally expensive due to the computing capabilities and memory constraints of these devices. Fortunately, ECC is well-suited for use in two-factor authentication. For instance, a lightweight protocol proposed by a team of researchers leverages elliptic curves and is resistant to various attacks like man-in-the-middle and replay attacks [9]. ECC can also be employed in a one-time password (OTP) scheme based on Lamport's OTP algorithm and for IoT devices using ECDH [6]. Finally, ECC finds application in safeguarding smart grids and securing communication channels for autonomous cars [10, 11].

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 $y^2 = x^3 + ax + b \ (mod\ p)$. We have implemented the point addition operation and scalar multiplication on a real SEC (Standards for Efficient Cryptography) ECC curve over a prime field using two different approaches. The ECDH key exchange system on a real SEC ECC curve has been implemented with two different approaches and validated. The performances of these distinct implementations have been compared and analyzed.

The rest of this paper is organized as follows:

Section II provides essential background information used in this paper. It introduces the ECC cryptographic system, detailing point addition and point doubling operations.

Section III describes the detailed implementation of ECC public-key systems on real NIST 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 IV 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, we implement and test ECDH key exchange system on a real SEC ECC curve.

Finally, Section V 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 [12]. 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 aim to 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 comprising 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 $G + G + G + ... + G$ using this group operator. To be specific, for any arbitrary number $k \notin G$, 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: $G + G$, $G + G + G$, $G + G + G + ... + G$ [13]. 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 [13]. Instead, they are defined using cubic equations, which are also employed in determining the circumference of an ellipse [14]. These curves commonly adhere to a form known as the Weierstrass equation

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

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 equation 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:

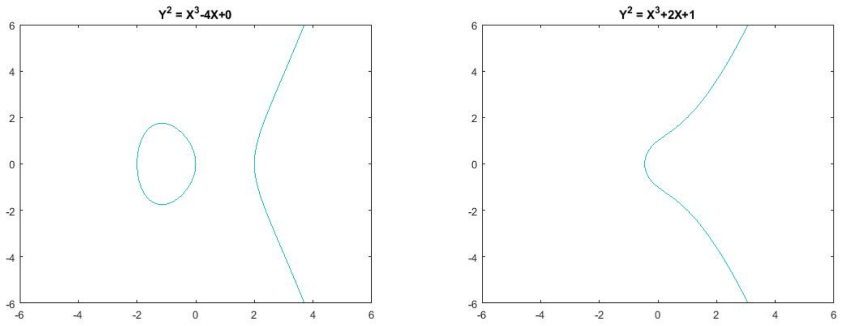


Figure 1: Examples of Elliptic Curves

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

If the discriminant is zero, it indicates that two or more roots have merged, rendering the curve non-smooth [13]. 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.

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 [13, 14] 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.
3. Perform a reflection of the point R along the x-axis, resulting in a point denoted by $-R$. This reflection is feasible due to the equation 2, which can be reformulated as $y = \sqrt{x^3 + ax + b}$, signifying the curve's symmetry concerning the x-axis.

Geometrically, point additions on a curve over real numbers can be visually represented, as depicted in Figure 2.

Thus, the addition of two points results in $P + Q = -R$. 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 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.

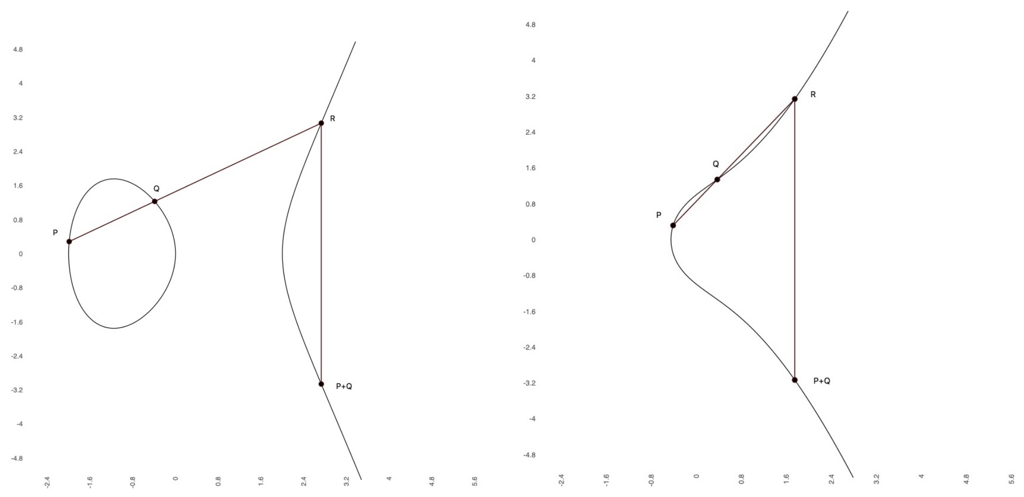


Figure 2: Point Addition on Elliptic Curves

- If P is the mirrored reflection of Q concerning the x-axis, then $P = -Q$.
- $O + O = O$ and $O = -O$

The aforementioned point addition process can be expressed algebraically. Let's consider the addition of two arbitrary points P and Q on an elliptic curve $E(a, b)$. This process results in three intersections with the curve, such that:

$$P + Q = -R \tag{5}$$

where it is a requirement that $P + Q + R = O$.

2.4.3 Point doubling

ECC involves repeatedly adding a point to itself k times to obtain another point denoted as $k \times G$. This operation, known as point doubling, is expressed as $P + P = 2P$. Point doubling is akin 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,..., 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:

$$y^2 \equiv x^3 + ax + b \pmod{p} \tag{6}$$

where the condition 3 is also satisfied in the form:

$$\left(4a^3 + 27b^2\right) \neq 0 \pmod{p} \tag{7}$$

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 [13]. 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:

$$\alpha = (y_Q - y_P)(x_Q - x_P)^{-1} \pmod{p} \tag{8}$$

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

$$\alpha = \left(3x_P^2 + a\right)(2y_P)^{-1} \pmod{p} \tag{9}$$

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 [13].

2.6 Point Encoding

In most cryptographic systems, it's necessary to transform a plaintext into a value applicable to a particular cryptographic algorithm [14]. 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 $GF(p)$ [14]. Nonetheless, the Koblitz algorithm [15] 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 6, 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 $1/2^K$. The plaintext message must fulfill the following condition:

$$(m + 1)K < p \tag{10}$$

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

$$0 \leq m \leq \frac{p - K}{K} \tag{11}$$

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

$$x = mK + j \tag{12}$$

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 $x^3 + ax + b$ $(\underline{m}od\ p)$. 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 12 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:

$$m = [x/K] \tag{13}$$

2.7 Cryptographic Schemes

Repeated additions aren't directly utilized for encryption, as described by $m \times G$ [13]. Instead, the concept of point multiplication is employed in various cryptographic schemes and algorithms. In this subsection, we introduce the Elliptic Curve Diffie-Hellman key exchange scheme.

2.8 Elliptic Curve Diffie-Hellman

The Elliptic Curve Diffie-Hellman (ECDH) secret key exchange scheme is utilized to establish a shared secret key between two parties. Let's consider Alice and Bob aiming to establish a secure communication channel. They choose ECC parameters p, a, and b for an elliptic curve, defined by Equation 6, along with a base point $B \in E_p(a, b)$.

Alice selects an integer X_A as her private key and computes a point $Y_A = X_A \times G$ as her public key. Simultaneously, Bob selects his private key represented by the integer X_B and computes a point on the curve $Y_B = X_B \times G$. Both Alice's and Bob's public keys, Y_A and Y_B respectively, are shared between them. Subsequently, they calculate the shared secret key. Alice uses the following equation to derive the secret key:

$$K = X_A \times Y_B \tag{14}$$

while Bob obtains the secret key by:

$$K = X_B \times Y_A \tag{15}$$

In Equation 14 and 15, K represents a point on the elliptic curve within $E_p(a, b)$. Both equations yield the same value for K, denoted as $K = X_A \times X_B \times G$, due to the fact that $Y_B = X_B \times G$, $Y_A = X_A \times G$, and the elliptic curve group $E_p(a, b)$ adheres to the associativity property.

2.9 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 8 and 9. 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 [16].

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 $(x_P, y_P, 1)$. 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 $(X, -Y, Z)$.

3 ECC Implementations

Understanding the main components of ECC is crucial for implementing it in software engineering. 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 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 ECDH (Elliptic Curve Key Exchange), 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 [17]. However, our implementation is compatible with any valid elliptic curve over $GF(p)$.

Table 2:

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

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 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 6.

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 aimed to balance speed and memory, favoring speed with arrays of Boolean values. 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 [18]. 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 [14]. 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 ECDH Implementation

One of the critical applications of ECC is the Diffie-Hellman key exchange. We implement the ECDH (Elliptic Curve Diffie-Hellman Key Exchange) technique using stream sockets over TCP. The ECDH process for establishing secure communication is depicted in Figure 3. For illustrative purposes, let's consider a scenario where Alice and Bob aim to establish a secure shared secret key. In our socket configuration, Bob serves as the server, while Alice assumes the role of the client. For the setup, we assume that Alice and Bob have already agreed upon an elliptic curve and a base point G (the secp192r1 curve).

Alice establishes a connection with Bob's machine using standard sockets provided by the C++ language. Upon establishing socket communication, she generates a random number X_a as her private key. Employing the point multiplication operation, she computes her public key $Y_a = X_a \times G$. Subsequently, Alice transmits her generated public key to Bob.

Likewise, Bob generates his private key X_b and public key $Y_b = X_b \times G$, which he sends to Alice. Subsequently, both parties can compute the shared secret key $X_a \times X_b \times G$. Once this process concludes, they exchange a flag message indicating their readiness to securely transmit messages to each other using the established shared secret key.

Algorithm 1: 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 1 displays the pseudocode for establishing a socket, connecting to a server, and creating a shared secret session key between the client's machine and the server. We utilize standard Linux C built-in header files for socket creation and connection establishment between the client and the server.

Lines 1-3 define the socket system call to create a socket. If the system call returns a negative value, it indicates an error in socket creation, which is handled in lines 2 and 3. Subsequently, the necessary information about the server the client will connect to is described.

The final step involves connecting to the specified server using the socket created in line 1. Upon successful connection, we generate public and private keys. The private key is randomly generated using the built-in *srand*() function, seeded by time. The public key is derived through a point multiplication operation. This public key is written to the socket for the server's use in calculating the shared secret key.

Following the logic outlined in Figure 3, the client waits for the server's public key to compute the shared secret key. Ultimately, a flag with a value of 1 is transmitted to confirm the success of the ECDH key exchange process.

The pseudocode for the server-side ECDH key exchange is shown in Algorithm 2. It closely resembles the implementation on the client side with some minor differences. On the server side, socket initialization involves binding the socket to a specified port and verifying the success of this action, as depicted in lines 4 and 5. Furthermore, the server must accept a connection request from the client. Once the connection is established, the key exchange logic outlined in Diagram 3 is followed, as demonstrated in lines 10-16.

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.

Algorithm 2: 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');
```

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 [19].

4.2 Experimental Results

This subsection outlines the experimental findings wherein we compare the time performance of arithmetic operations, including addition, subtraction, division, multiplication, and modulo exponentiation, performed using the Big Integer classes.

Additionally, we measured the execution times for point addition, point doubling, and scalar multiplication operations over the secp192r1 curve [18]. These operations were utilized in the implementation of the ECDH key exchange mechanism. 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. Due to space limit, only the timing performance of point addition is reported and compared in this paper.

Table 3: 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 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 comprises 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.

Table 4: 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

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.

Table 5: 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

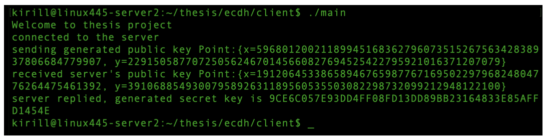
The table illustrates a slight decrease in performance during the point addition operation. However, the point doubling operation displays nearly three times faster performance. This discrepancy arises due to the significantly reduced number of arithmetic operations involving Big Integers when employing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

4.5 Verification of the Correctness

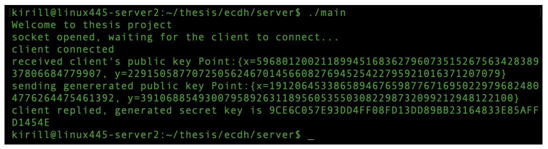
We have successfully implemented the ECDH key exchange mechanism and verified the accuracy of these implementations as demonstrated below.

Figure 3 displays the screenshots depicting the established shared secret key during the ECDH key exchange process on the client and server sides. It is evident from the images that both parties have effectively exchanged their public keys and have independently calculated an identical shared secret key. This shared secret key, represented in hexadecimal format, matches on both ends, thereby validating the correctness of the ECDH key exchange mechanism and the Big Integer classes.

Table 6 presents the parameters utilized in the ECDH key exchange process, along with the resulting shared secret key generated on both the server and client sides.



```
kirill@linux445-server2:~/thesis/ecdh/client$ ./main
Welcome to thesis project
connected to the server
sending generated public key Point:{x=59680120021189945168362796073515267563428389
37806684779907, y=2291505877072505624670145660827694525422795921016371207079}
received server's public key Point:{x=19120645338658946765987767169502297968248047
76264475461392, y=3910688549300795892631189560535503082298732099212948122100}
server replied, generated secret key is 9CE6C057E93DD4FF08FD13DD89BB23164833E85AFF
D1454E
kirill@linux445-server2:~/thesis/ecdh/client$ _
```



```
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=59680120021189945168362796073515267563428389
37806684779907, y=2291505877072505624670145660827694525422795921016371207079}
sending genererated public key Point:{x=191206453386589467659877671695022979682480
4776264475461392, y=3910688549300795892631189560535503082298732099212948122100}
client replied, generated secret key is 9CE6C057E93DD4FF08FD13DD89BB23164833E85AFF
D1454E
kirill@linux445-server2:~/thesis/ecdh/server$ _
```

Figure 3: Client side and Server side of the ECDH Key Exchange

Table 6: 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: 5968012002118994516836279607351526756342838937806684779907 y: 2291505877072505624670145660827694525422795921016371207079
Bob's public key	x: 1912064533865894676598776716950229796824804776264475461392 y: 3910688549300795892631189560535503082298732099212948122100
generated secret key	3847210457611900028777758802678809708230333084326754862414

5 Conclusions And Future Work

This paper focuses on the software implementations of ECC over the finite field GF(p) by utilizing Big Integer classes through character arrays and bit sets in the C++ programming language.

From the performance results reported in section IV, we conclude that the big integer classes using bit sets are not suitable for cryptography due to their performance. Additionally, the big integer classes using arrays of characters could be further optimized.

We have tested and validated one of the most popular applications of ECC: the ECDH key exchange. We utilized standard C++ stream sockets operating over the TCP protocol to simulate ECC applications on a real secp192r1 curve. However, our implementation works on any valid elliptic curve, accommodating different parameters of the same size and parameters with varying sizes.

In our future work, we plan to optimize our implementations of the big integer classes and point operations. Moreover, the execution time of arithmetic operations of the big integer classes could be improved by parallelizing these operations using GPU.

Furthermore, we intend to boost the performance of point operations by utilizing mixed addition operating on a combination of Projective coordinates and Affine coordinates. Finally, we aim to enhance both the performance and security of ECC applications by implementing the Montgomery ladder algorithm for scalar multiplication of points on a given elliptic curve. This implementation aids in defending ECC applications against side-channel attacks.

References

[1] A. Sghaier, "Software implementation of ecc using gmp library," 03 2016.

[2] R. Ganesan, "Computer system for securing communications using split private key asymmetric cryptography," Patent, 1996. [Online]. Available: https://patents.google.com/patent/US5737419A/en

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

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

[5] T. Juhas, "The use of elliptic curves in cryptography," 2007.

[6] 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*, ser. CISRC '17. New York, NY, USA: ACM, 2017, pp. 6:1–6:7. [Online]. Available: http://doi.acm.org/10.1145/3064814.3064818

[7] S. Rahmadika and K.-H. Rhee, "Blockchain technology for providing an architecture model of decentralized personal health information," *International Journal of Engineering Business Management*, vol. 10, p. 1847979018790589, 2018. [Online]. Available: https://doi.org/10.1177/1847979018790589

[8] 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*, vol. 25, no. 2, pp. 738–750, April 2017.

[9] 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*, vol. 4, pp. 4394–4407, 2016.

[10] D. He, H. Wang, M. K. Khan, and L. Wang, "Lightweight anonymous key distribution scheme for smart grid using elliptic curve cryptography," *IET Communications*, vol. 10, no. 14, pp. 1795–1802, 2016.

[11] 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)*, July 2016, pp. 1–6.

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

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

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

[15] A. T. Reney Brandy, Naleceia Davis, "Encrypting with elliptic curve cryptography," 07 2010, pp. 9–17.

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

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

[18] K. Kultinov, "Software implementations and applications of elliptic curve cryptography," 2019.

[19] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1273442.1250746