A Secure and Efficient ECC Scheme for IoT

Malek Al Khatib

Submitted to the
Institute of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Eastern Mediterranean University
July 2024
Gazimağusa, North Cyprus

Approval of the Institute of Graduate Studies and Research

Prof. Dr. Ali Hakan Ulusoy
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science in Computer Engineering.

Prof. Dr. Zeki Bayram
Chair, Department of Computer Engineering

We certify that we have read this thesis and that in our opinion it is fully adequate in scope and quality as a thesis for the degree of Master of Science in Computer Engineering.

Prof. Dr. Alexander Chefranov
Supervisor

Examining Committee

1. Prof. Dr. Alexander Chefranov

2. Assoc. Prof. Dr. Gürcü Öz

3. Asst. Prof. Dr. Öykü Akaydın

ABSTRACT

This thesis introduces the design, implementation, testing and evaluation of an efficient and secure Elliptic Curve Cryptography (ECC) system, previously designed in SE-Enc system article: DOI: 10.1109/ACCESS.2019.2957943, for the Internet of Things (IoT). SE-Enc system. The SE-Enc system is said to be a secure and efficient ECC encryption scheme successfully applied in later research on IoT systems. It is chosen because of this, but it has some security deficiencies concerning chosen plain text attack (CPA), chosen cipher text attack (CCA), and man-in-the-middle attacks (MITM), so in the thesis, several critical security and efficiency issues need to be addressed. Firstly, the use of a constant initial vector (IV) fails to effectively block CCA and CPA, thereby exposing the system to these significant security threats. Secondly, the implementation of the Diffie-Hellman Key Exchange mechanism leaves the system vulnerable to MITM attacks, which can compromise the security of key exchanges. Lastly, the current practice of using three padding bits on the least significant bit (LSB) for mapping points is insufficient, resulting in a higher number of rounds required for successful mapping. This method should be replaced with a more efficient approach to enhance the system's overall efficiency and security. Addressing these issues is crucial for developing a robust and secure Elliptic Curve Cryptography (ECC) scheme.

In this thesis, an ECC scheme has been implemented equivalent to SE-Enc, with some modifications which are done to fix the problems found: 1) random IV is used instead of constant IV in order to block CPA and CCA. 2) Using trusted public keys for key agreement using the Diffie-Hellman protocol will be included, which will help to block

MITM which isn't blocked in SE-Enc. 3) using 5 padding bits in the mapping function instead of 3 padding bits to ensure having correct encryption and decryption because 3 padding bits accepts 8 rounds at most in mapping and more rounds are needed to map the points.

Four experiments are conducted to prove that this ECC scheme has a good security and efficiency, where the first experiment's purpose is to prove that this scheme blocks chosen CPA and CCA, while this experiment is discussed in the article: DOI: 10.3390/s20216158, and this article is an updated SE-Enc, and proves that its scheme blocks CCA and CPA. The second experiment computes the successful rate for mapping points to ECC in each round and counts the rounds to check if these rounds are suitable for the used padded points and the results of my experiment show that 3 padding bits aren't enough, while the third experiment discusses the computation of the performance (time needed for compiling) of the encryption process of this research's ECC scheme and compared to the one in Sengupta's article: DOI: 10.1002/sec.1702, and my results show 31 seconds which is slightly less than Sengupta's scheme result, which is 27 seconds. And the last experiment is done to check if the size of padding bits has any effect on the performance, and based on the results, it doesn't have any effect. The experimental results show the deficiencies of SE-Enc which appears with using constant IV, instead of using random IV, and in this research's ECC scheme this is solved, and using trusted public keys for key agreement because Diffie-Hellman is susceptible for MITM attacks. Also SE-Enc uses 3 padding bits which won't work for all mapping point and the second experiment proves it. This thesis designed, implemented, and tested an efficient and secure ECC which can be used for IoT.

Keywords: SE-Enc**,** Elliptic Curve Cryptography (ECC), Internet of Things (IoT), Point Mapping, Diffie-Hellman Key Exchange, Initial Vector (IV).

ÖZ

Bu tez, Internet of Things (IoT) için daha önce SE-Enc sistem makalesinde: DOI: 10.1109/ACCESS.2019.2957943 tasarlanan verimli ve güvenli bir Eliptik Eğri Kriptografi (ECC) sisteminin tasarımını, uygulanmasını, test edilmesini ve değerlendirilmesini tanıtmaktadır. SE-Enc sistemi, IoT sistemleri üzerine yapılan sonraki araştırmalarda başarıyla uygulanan güvenli ve verimli bir ECC şifreleme şeması olarak kabul edilmektedir. Bu nedenle seçilmiştir, ancak seçilmiş düz metin saldırısı (CPA), seçilmiş şifre metni saldırısı (CCA) ve ortadaki adam saldırıları (MITM) ile ilgili bazı güvenlik eksiklikleri bulunmaktadır. Bu tezde, birkaç kritik güvenlik ve verimlilik sorununun ele alınması gerekmektedir. İlk olarak, sabit bir başlangıç vektörünün (IV) kullanılması, CCA ve CPA'yı etkili bir şekilde engelleyememekte, böylece sistemi bu önemli güvenlik tehditlerine maruz bırakmaktadır. İkinci olarak, Diffie-Hellman anahtar değişim mekanizmasının uygulanması, anahtar değişimlerinin güvenliğini tehlikeye atabilecek MITM saldırılarına karşı sistemi savunmasız bırakmaktadır. Son olarak, noktaları eşlemek için En Az Önemli Bit (LSB) üzerinde üç dolgu biti kullanma uygulaması yetersiz olup, başarılı eşleme için gereken daha fazla tur sayısına neden olmaktadır. Bu yöntem, sistemin genel verimliliğini ve güvenliğini artırmak için daha verimli bir yaklaşımla değiştirilmelidir. Bu sorunların ele alınması, sağlam ve güvenli bir Eliptik Eğri Kriptografi (ECC) şeması geliştirilmesi için çok önemlidir.

Bu tezde, SE-Enc sistemine benzer bir ECC şeması, bulunan sorunları çözmek için bazı değişikliklerle uygulanmıştır: 1) CPA ve CCA'yı engellemek için sabit başlangıç vektörü (IV) yerine rastgele başlangıç vektörü (IV) kullanılmıştır. 2) Diffie-Hellman

protokolü kullanılarak anahtar anlaşması için güvenilir genel anahtarların kullanılması, SE-Enc'de engellenmeyen MITM saldırılarını engellemeye yardımcı olacaktır. 3) 3 dolgu biti yerine 5 dolgu biti kullanarak şifrelemenin ve şifre çözmenin doğru olmasını sağlamak, çünkü 3 bit (LSB) en fazla 8 tur kabul eder ve noktaları eşlemek için daha fazla tur gereklidir.

Bu ECC şemasının iyi bir güvenlik ve verimliliğe sahip olduğunu kanıtlamak için dört deney yapılmıştır. İlk deneyin amacı, bu CPA ve CCA engellediğini kanıtlamaktır; bu deney makalede: DOI: 10.3390/s20216158 tartışılmıştır ve bu makale, SE-Enc'nin güncellenmiş halidir ve şemasının CCA ve CPA'yı engellediğini kanıtlar. İkinci deney, her turda ECC'ye noktaların eşlenmesinin başarı oranını hesaplar ve kullanılan dolgulu noktalar için bu turların uygun olup olmadığını kontrol etmek için turları sayar ve deney sonuçlarım, 3 dolgu bitinin yeterli olmadığını gösterir. Üçüncü deney, bu araştırmanın ECC şemasının şifreleme sürecinin performansının (derleme için gereken süre) hesaplanmasını ve Sengupta makalesindekiyle: DOI: 10.1002/sec.1702 karşılaştırılmasını tartışır ve sonuçlarım 31 saniye gösterir, bu da Sengupta'nın şema sonucundan (27 saniye) biraz daha azdır. Ve son deney, dolgu bitlerinin boyutunun performans üzerinde herhangi bir etkisi olup olmadığını kontrol etmek için yapılmıştır ve sonuçlara göre, performans üzerinde hiçbir etkisi yoktur. Deney sonuçları, sabit IV kullanmanın SE-Enc'deki eksikliklerini göstermektedir, rastgele IV kullanılarak bu araştırmanın ECC şemasında bu sorun çözülmüştür ve güvenilir genel anahtarlar kullanarak anahtar anlaşması yapılmaktadır, çünkü Diffie-Hellman MITM saldırılarına karşı savunmasızdır. Ayrıca, SE-Enc 3 dolgu biti kullanmaktadır ve bu tüm noktaları eşlemek için yeterli değildir ve ikinci deney bunu kanıtlamaktadır. Bu tez, IoT için kullanılabilecek verimli ve güvenli bir ECC tasarlamış, uygulamış ve test etmiştir.

Anahtar Kelimeler: SE-Enc, Eliptik Eğri Kriptografi (ECC), Nesnelerin İnterneti (IoT), Nokta Eşleme, Diffie-Hellman, Başlangıç Vektörü (IV).

DEDICATION

To everyone who touches my heart and to everyone who wishes for my success,

thank you for your unwavering support and encouragement.

ACKNOWLEDGMENT

With all my heart, I would like to thank my supervisor, Prof. Alexander Chefranov, for his support, direction and precious ideas during the stages of my research. He made this thesis and much of my career possible by his wisdom and vast knowledge in the field. I thank him for his unending support and patience during the difficult periods I went through during my research and for pushing me to always strive for the best. Thank you, Prof. Chefranov for all your time and for believing in me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

The Internet of Things (IoT) is rapidly becoming a wide-scale networked environment where smart devices and systems are linked through the internet to enhance communication and increase operational effectiveness [1]. The IoT provides a huge impetus to many industries; however, this does not come without a set of security challenges caused by the complexity of the technology and the sensitivity of the data involved [1].

The issue of security and efficiency in IoT network communications remains the central concern as the challenges grow more pronounced. Traditional cryptographic approaches are typically insufficient, both from security and performance perspectives, due to the fact that IoT devices are deployed in resource-constrained environments typical of IoT networks [2]. Hence, it is clear that we need solutions that not only secure data but are also agile enough to reflect the dynamic nature of IoT [2].

This thesis presents an Elliptic Curve Cryptography (ECC) scheme for IoT, aimed at addressing the aforementioned problems. ECC is widely acknowledged for its security and small key sizes compared to conventional cryptographic methods with preserving its security level, making it an ideal choice for systems that are power-constrained and have limited computational resources (less storage for example). In this context, the

proposed in [3] ECC-based scheme aims to strengthen the security of the network model designed for IoT applications and increase efficiency simultaneously.

The present thesis aims to design, implement, and test a secure and efficient ECC scheme for IoT following the guidelines of [3], with modifications addressing several identified issues. Firstly, the ECC scheme will incorporate a random initial vector (IV) to mitigate chosen plaintext attack (CPA) [24] and chosen cipher text attack (CCA) [24], which is a feature missing in [3] (Problem 1). Secondly, it will address the number of rounds needed for Mapping Points to ECC (Problem 2). Thirdly, the thesis will evaluate the performance (encryption time) of using 3 padding bits to 8 padding bits, determining the efficiency of using 3 bits compared to 5 and 8 padding bits (Problem 3). Lastly, the encryption time will be evaluated and compared to the results presented in [4] (Problem 4).

The purpose of this thesis is to focus on the security and efficiency of ECC scheme [3] for IoT, and to improve its security by using random IV instead of constant IV and using trusted public keys for key agreement process. These modifications help to block several type of attacks which lead to enhance security. Also, 5 padding bits are used instead of 3 padding bits as least significant bits (LSB), which means adding zero bits on the right side will affect the value, and 5 padding bits are used because it is sufficient for the number of rounds needed in mapping. In this thesis, an ECC scheme similar to [3] is designed, implemented, tested and evaluated by experiments which have similar results to [4], [3] and [5], using similar experimental settings, with taking into consideration the modifications that will lead to enhance the security of this scheme.

So, to summarize the contributions of the thesis, they are mentioned below:

1. A secure and efficient ECC scheme for IoT has been designed, implemented, and tested, building upon SE-Enc framework with some modifications which are mentioned in other contributions. In addition of experiments that are conducted later.
2. Implementation of a Random Initial Vector (IV**):** Introduced a random IV instead of a constant IV to block CPA) and CCA
3. Enhanced Key Agreement Protocol**:** Adding the use of trusted public keys for key agreement using the Diffie-Hellman protocol to mitigate man-in-the-middle (MITM) attacks.
4. Improved Padding Bit Scheme**:** Replaced the use of three padding bits with five padding bits in the mapping function to ensure correct encryption and decryption, reducing the number of rounds required for successful mapping, and checking the most optimal padding bits number.
5. Evaluating the encryption time compared to Sengupta [4]: Checking the performance (time) needed to encrypt the message in the thesis's scheme and compare it to [4].

The rest of the thesis is structured as follows:

- Chapter 2 considers the literature review that surveys the previous research in ECC Also, it includes an overview for IoT. The review points to the necessity of the ECC system model as suggested in [3], with slight modifications made for some algorithms to correct some typing mistakes and missing explanations.
- Chapter 3 considers the design, implementation a of my system that follow the principles of [3] with few modifications, which are using random initial vector instead of constant initial vector, and using 5 padding bits to the right in the mapping Algorithm 12 in [3] instead of 3 padding bits to the right.

- Chapter 4 discusses the experimental settings and results which are similar with ones in [4], [3] and [5].
- Chapter 5 concludes with a review of the contribution of my work, and directions for future improvements of ECC in IoT.
- At the end, the references and the appendices are given.

Chapter 2

REVIEW OF THE WORKS ON SECURE AND EFFICIENT ECC FOR IOT

This chapter is divided into several sections starting with the Internet of Things (IoT) covering its network architecture, significant fields of application of IoT, and the IoT specific security and efficiency challenges (section 2.1). This is followed in section 2.2 by an analysis of Elliptic Curve Cryptography (ECC) including its properties, functions like point addition, scalar multiplication and the procedures of ECC encryption and decryption. ECC security and efficiency challenges are also addressed. Section 2.3 includes survey of effective and secure ECC is provided by discussing 4 schemes: SE-Enc [3] and [5], Singh [6], Sengupta [4], and Barman [27]. Also in section 2.3, known experiments from the mentioned systems: SE-Enc [3] and [5], Singh [6], and Sengupta [4] with their settings and results are analyzed. At last, this section closes a problem definition (section 2.4).

2.1 IoT Network Architecture, Application Fields, and Security and Efficiency Characteristics

The IoT links numerous devices, creating the connection between the digital and physical worlds which is both creative and revolutionary. Such devices, starting from common consumer objects to elaborate industrial systems, are made to gather, to send, and to process data by themselves. Their involvement into different sectors is transforming effectiveness and improving decision processes [1].

In section 2.1 we will cover the following:

- Section 2.1.1 IoT Network Architecture: This section discusses the IoT network architecture with its layers.
- Section 2.1.2 Application Fields: This section also focuses on detailing how IoT can be used across the various fields so as to see its potential.
- Section 2.1.3 Security and Efficiency Characteristics: This section discusses some of the key challenges that are inherent in IoT systems, specifically concerning security and efficiency, which are fundamental requirements for preserving data integrity and utilizing limited resources constraints.

2.1.1 IoT Network Architecture [7]

IoT network architecture refers to the arrangement or design of connecting devices like sensors, which form a network within the IoT environment. The architecture typically comprises several layers, each responsible for different functions:

- Perception Layer: consists of sensors and actuators that collect data from the physical environment
- Network Layer: transmits, with the aid of communication protocols and network infrastructure, the collected data from the perception layer to the higher layer, with ensuring reliable transfer by managing data routing, addressing, and error handling.
- Application Layer: contains data-driven applications that provide user interfaces and actionable insights. This layer consist of:

Figure 1 shows the IoT architecture:

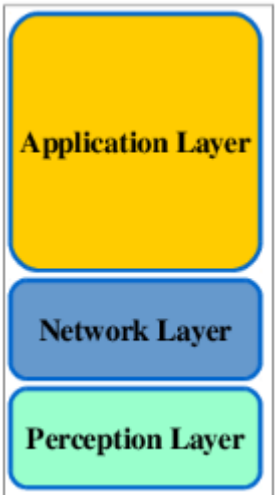


Figure 1: IoT Network Architecture [7]

2.1.2 Important Fields of Applications of IoT [8]

IoT is found in different fields of applications, like:

1. Smart Homes and Healthcare: IoT devices control home surroundings and measure health parameters providing comfort and valuable health data at a distance.

   For Instance: Remote Monitoring System for patients

   Layer 1: Perception Layer

   - Devices: Health monitor devices which are wearable such as smart watches, fitness bands
   - Function: Wearable devices will be responsible for data collection as heart rate, pressure, etc...

   Layer 2: Networking Layer

   - Protocols of communication: Zigbee, Wi-Fi
   - Function: Using the protocols the data will be transmitted from the IoT device to the central database

   Layer 3: Application Layer

- Platforms: Cloud-based platforms for health monitoring such as Apple watch
- Function: the collected data are analyzed carefully in order to provide health recommendations to the users

2. Industrial IoT: In the industrial sector, IoT improves manufacturing operations through automation and continuous tracking. Machinery sensors can foresee failures in advance, while intelligent systems can optimize supply chains and inventory control

   For example: Predictive Maintenance System [9]

   Layer 1: Perception Layer

   - Devices: Sensors equipped by machine for example the temperature sensors
   - Function: These sensors will collect real time data for the machines

   Layer 2: Networking Layer

   - Protocols of communication: Ethernet, Wi-Fi
   - Function: using the communication method data are transmitted to central

     Databases

   Layer 3: Application Layer

   - Platforms: The IoT platforms which are related to industrial field such as Siemens Mind Sphere [10].
   - Function: In order to optimize the maintenance system of the machines, the collected data will be used to predict potential failures.

3. Agriculture [11]: IoT technologies in agriculture provide a method for monitoring crop fields and automate irrigation systems which causes more crop yield and less waste. Drones and sensors collect information on the health of the crop, soil quality, and climate conditions.

   For example: Advanced Irrigation system [11]

   Layer 1: Perception Layer

   - Devices: Weather stations, moisture detectors in the soil.
   - Function: These sensors will collect data about weather conditions, soil moisture levels, and general data about the crop health

   Layer 2: Networking Layer

   - Protocols of communication: Lora, cellular networks
   - Function: using the communication method data are transmitted to central database through low-range communication protocols

   Layer 3: Application Layer

   - Platforms: The IoT platforms which are related to agriculture field such as Climate Field View.
   - Function: The collected data will be processed and analyzed in order to optimize the irrigation schedules; such optimization will regulate the quantity of water delivered to the crop regarding the crop stats.

2.1.3 Security and Efficiency Requirements of IoT [5]

The problem of design of IoT is efficiency and security [5] due to the required characteristics of security and efficiency and of IoT devices and networks. Let's first see Elliptic Curve Cryptography ECC) in IoT, while ECC in an IoT application is typically used in the **Security and Privacy Layer** to ensure secure communication

and data protection. Here's an outline of an IoT healthcare application structure highlighting where ECC is implemented:

1. **Sensors and Devices Layer**:
    - Health sensors and wearable devices collect patient data.
2. **Communication Network Layer**:
    - Wireless communication protocols (Wi-Fi or Bluetooth) transmit data to the gateway.
3. **Gateway Layer**:
    - Aggregates data from sensors, and preliminary data processing, and ensures secure transmission to the cloud.
4. **Cloud Platform Layer**:
    - Data storage, processing, and analysis using advanced algorithms and machine learning.
5. **Security and Privacy Layer** (where ECC is used):
    - **Encryption**: ECC is used to encrypt data during transmission between sensors, gateways, and the cloud to protect against eavesdropping.
    - **Authentication**: ECC provides secure key exchange mechanisms (e.g., Diffie-Hellman key exchange) to authenticate devices and users, preventing unauthorized access.
    - **Digital Signatures**: ECC-based digital signatures ensure the integrity and authenticity of data, confirming that it has not been tampered with.
6. **Healthcare Provider Interface**:
    - User interface for doctors and healthcare providers to monitor patient data and make informed decisions.
7. **Patient Interface**:

- Mobile apps or web portals for patients to track their health metrics, receive alerts, and communicate with healthcare providers.

Now let's consider these characteristics [5]:

Required Characteristics of ECC Security for IoT:

1. Data Protection [12]: The data in question typically includes health metrics, operational data, and personal preferences. Ensuring the privacy of this data is crucial in maintaining user trust. For instance, in healthcare, protecting patient health records is paramount to comply with regulations like HIPAA [6] (Health Insurance Portability and Accountability Act). HIPAA sets the standard for protecting sensitive patient data and mandates that healthcare providers implement measures to ensure the confidentiality, integrity, and availability of electronic protected health information.
2. Strong Encryption [13]: through several testing and analysis, ECC proved to be efficient encrypting technique, after compared with famous encryption techniques ECC was able to provide a shorter key which is at the same time harder to crack by 10k times, meaning that ECC is efficient regarding space and security, also ECC meets the NIST standards [7].

Required Characteristics of ECC Efficiency for IoT:

1. Resource Constraints [14]: Many IoT devices operate with limited computational capacity, memory, and energy. Efficient algorithms and protocols are needed to maximize their lifespan and performance. For example, using lightweight cryptographic protocols like ECC can reduce the computational burden on resource-constrained devices, allowing them to function effectively with minimal resources.

2. Cost Effectiveness [14]: IoT efficiency also implies cost-efficiency. Reducing power consumption and bandwidth needs lowers operational costs, making IoT solutions more affordable and scalable. For example, implementing energy-saving modes and efficient data aggregation techniques can reduce the overall cost of IoT deployments in smart agriculture, enabling broader adoption and more extensive implementation.
3. Scalability [13]: As the number of connected IoT devices grows, the potential attack surface expands, requiring robust security measures to protect the entire system. For example, in smart city applications, ensuring secure communication across thousands of sensors and devices is crucial to prevent large-scale disruptions. Effective security protocols must scale to handle increasing data traffic and device interactions without compromising performance.

In summary, ECC is particularly well-suited for IoT environments due to its efficiency and advanced security properties. Its smaller key sizes reduce the computational load and memory usage, making it ideal for devices with limited resources. Additionally, ECC provides strong security to protect sensitive data and ensuring secure communication, which is critical in IoT applications where data privacy, network security, device integrity, and compliance with regulations are paramount. By addressing the scalability of IoT, ECC helps maintain security as the number of connected devices grows.

2.2 Elliptic Curve Cryptography as an Efficient Tool for IoT Security

Elliptic Curve Cryptography is a major class of public key cryptosystems that use pairs of keys (one public and one private) for secure communication. The key size in ECC

is smaller than several cryptographic algorithms, allowing for faster computations and less storage requirement compared to other cryptographic algorithms [15] like RSA, without compromising security. The power of ECC comes from the difficulty of solving the Elliptic Curve Discrete Logarithm Problem (ECDLP), which is a mathematical challenge that forms the basis for the security of Elliptic Curve Cryptography (ECC) [16].

Point on Curve (G, P, Q, etc.): Elliptic curve points are the solutions of the equation of the curve (1) that are contained within the specified field. For instance, G is a "base point" or "generator point", because it is used to generate other points on the curve. This point is referred to as the starting point of cryptographic operations. Base point G can be any point on the curve, but this point is very important for its usage in the Encryption and Decryption of ECC, while more information will be discussed later in section 2.2.3.

Section 2.2 provides a comprehensive overview of Elliptic Curve Cryptography (ECC). Section 2.2.1 discusses elliptic curve over real numbers, which includes some mathematical operations for elliptic curve. Section 2.2.2 discusses elliptic curve over Fp, and some mathematical operations for elliptic curve. Section 2.2.3 covers background information which are needed to be known to proceed with ECC encryption and decryption. Section 2.2.4 explains the ECC encryption/decryption model 1 which is used in [4]. Finally, section 2.2.5 outlines encryption/decryption model 2 which is described in [3].

2.2.1 Elliptic Curve Over Real Numbers [12]

Before discussing points on elliptic curves over finite fields, it is helpful to introduce elliptic curves over the real numbers. An elliptic curve over the real numbers is defined by an equation of the form [12]:

$$y^2 = x^3 + ax + b \quad (1)$$

where a and b are real numbers. This equation describes a smooth, non-intersecting curve in the plane. For the curve to be non-singular (having no self-intersections), the discriminant

$$\Delta = 4a^3 + 27b^2 \quad (2)$$

must be non-zero.

Example 1: Check whether points P and Q are on (3)

Define the Curve:

The equation for the elliptic curve is as in (1):

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

Now, a = –4 and b =1.

Verify the discriminant (2):

The first test to do is checking that the curve is non-singular; checking the discriminant (2) for (3):

$\Delta = 4(-4)^3 + 27(1)^2 = 4(-64) + 27 = -256 + 27 = -229 \neq 0$

As Δ doesn't equal to 0, the curve is nonsingular.

Points on the Curve (3):

Let's prove that the following points lie on (3):

P= (x_P, y_P) = (3, 4)

Q= (x_Q, y_Q) = (0, 1)

Let's just see if these points lie on the elliptic curve (3).

1. For point P, let's check the equality of the left-hand side (LHS) of curve (3) and the right-hand side (RHS) of curve (3)

$y_P^2 = x_P^3 - 4x + 1$

$4^2 = 3^3 - 4 \times 3 + 1$

$16 = 16$

This proves that point P is on the curve (3).

2. For Q, let's check the equality of the left-hand side (LHS) of curve (3) and the right-hand side (RHS) of curve (3):

$y_Q^2 = x_Q^3 - 4x + 1$

$1^2 = 0^3 - 4 \times 0 + 1$

$1 = 1$

This proves that point Q is on the curve (3).

Figure 2 illustrates the elliptic curve for the equation (3) over real numbers:

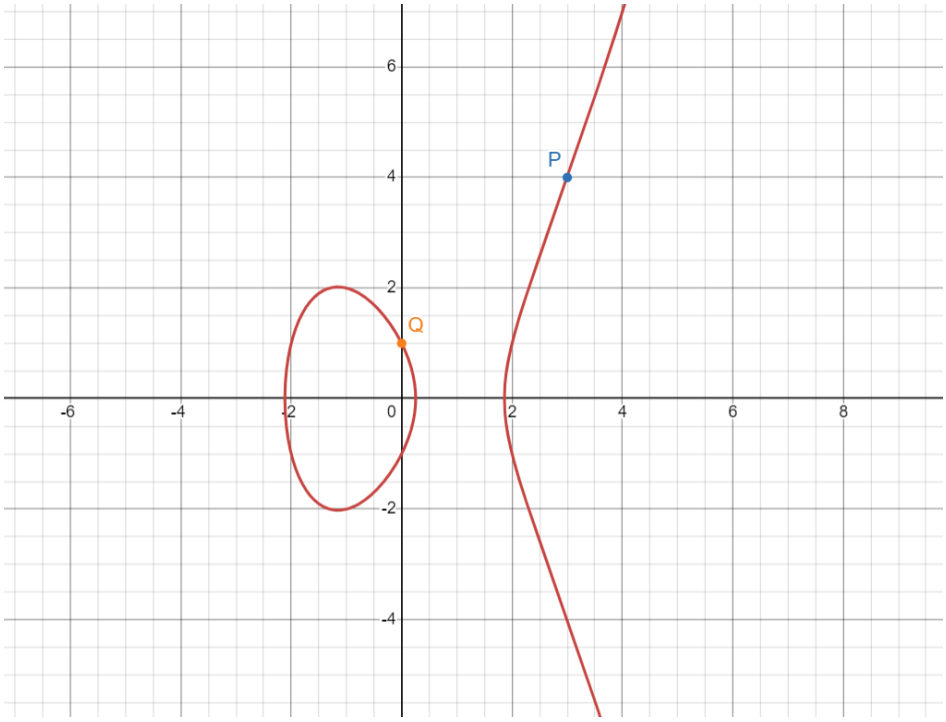


Figure 2: Elliptic Curve for Equation (3) Over Real Numbers

End of Example 1.

Point Addition Over Real Numbers in ECC [17]:

In ECC, point addition can be defined as mathematical calculation involving two points P and Q, where the result is to find a point R such that R= P+Q which defined in the coming lines. These operations are crucial to ECC for functions such as digital signatures and key exchanges in cryptographic algorithms that provide a very high level of security with small key sizes that can be implemented effectively in both hardware and software systems.

Addition of two points P and Q on an elliptic curve (1) (R=P+Q):

- $(x_P\ y_P)$: Coordinates for point P
- $(x_{Q,}\ y_Q)$: Coordinates for point Q
- λ: Slope of the straight-line PQ (line between the 2 points P and Q)
- $(x_{R,}\ y_R)$: Coordinates for point R which is shown in Figure 3

1. Addition of Distinct Points (P ≠ Q):

- Draw a straight line through points P and Q.

- This line will intersect the elliptic curve at exactly one more point, -R.

- Reflect this intersection point -R over the x-axis to get the point R.

For P not equal to Q (Figure 3), find:

$$\lambda = \frac{y_Q - y_P}{x_Q - x_P} \tag{4}$$

$$x_R = \lambda^2 - x_P - x_Q$$

$$y_R = \lambda(x_P - x_R) - y_P$$

2. Addition of a Point to Itself (Doubling, P = Q):

- If P = Q, the line through P and Q is the tangent to the curve at P.

- Draw the tangent line to the curve at P.

- This tangent will intersect the elliptic curve at another point, -R.

- Reflect this intersection point -R over the x-axis to get the point R.

If P equals Q (Figure 4 and Example 3), determine:

$$\lambda = \frac{3x_Q^2 + a}{2y_Q} \tag{5}$$

$$x_R = \lambda^2 - 2x_P$$

$$y_R = \lambda(x_P - x_{R,}) - y_P$$

Example 2: Addition of two points

For the following parameters for the Elliptic curve (1):

- a = -1
- b = 1

Thus, the equation of the curve becomes:

$$y^2 = x^3 - x + 1 \tag{6}$$

The following two points are on curve (6):

- P = (x_P, y_P) = (1,1)
- Q = (x_Q, y_Q) = (0, 1)

Calculate R =(x_R, y_R) = P + Q.

Steps to find R using (4):

1. Calculate the slope λ:

$$\lambda = \frac{y_Q - y_P}{x_Q - x_P} = \frac{1-1}{0-1} = \frac{0}{-1} = 0$$

2. Compute x and y for R on curve (6):

- $x_R = \lambda^2 - x_P - x_Q = 0^2 - 1 - 0 = -1$
- $y_R = \lambda(x_P - x_R) - y_P = 0(1-(-1)) - 1 = -1$

So, R = P + Q = (-1, -1)

Figure 3 shows the point addition for R = P + Q shown in Example 2 on curve (6) over real numbers:

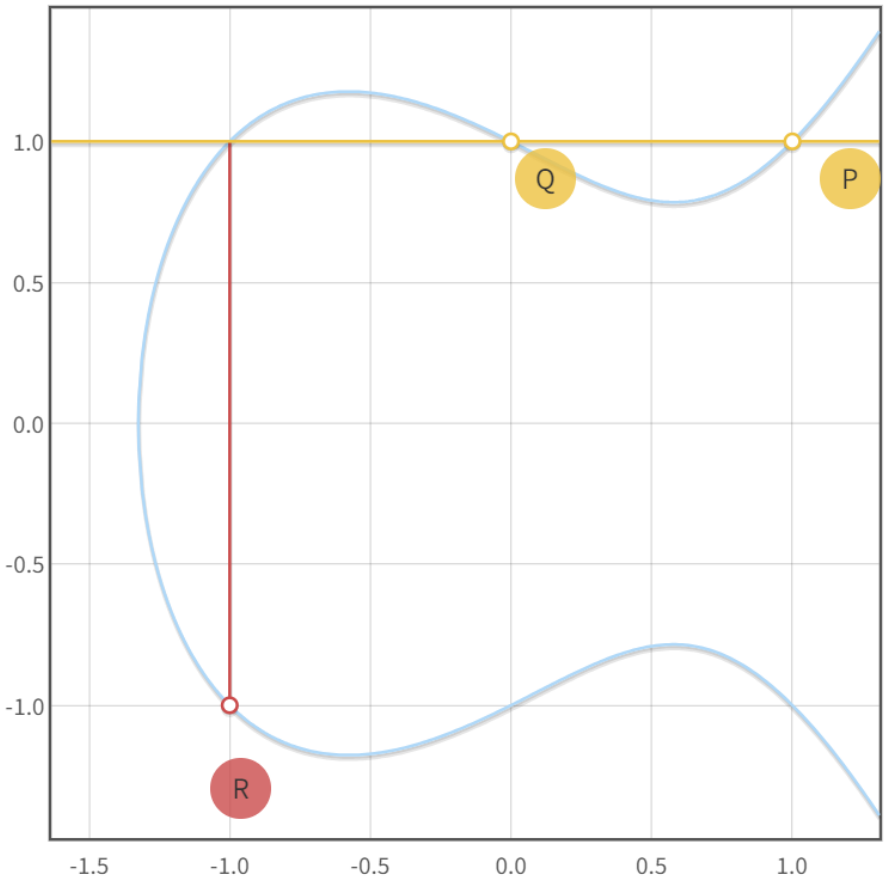


Figure 3: Point Addition for R = P + Q on Curve (6) Over Real Numbers

End of Example 2.

Example 3: Point doubling is held in this example

Elliptic curve (6) is used for Example 3.

The following point is on curve (6):

- Q = (x_Q, y_Q) = (1,1)

Calculate R =(x_R, y_R) = Q + Q.

Steps to find R using (5):

1. Calculate λ:

$$\lambda = \frac{3x_Q^2 + a}{2y_Q} = \frac{3(1)^2 - 1}{2(1)} = \frac{2}{2} = 1$$

2. Compute x and y for R on curve (6):

$$x_R = \lambda^2 - 2x_Q = 1^2 - 2(1) = -1$$

$$y_R = \lambda(x_Q - x_{R,}) - y_Q = 1(1 - (-1)) - 1 = 1$$

So, R = Q + Q = (-1, 1)

Figure 4 illustrates the point doubling operation for R = Q + Q discussed in Example 3 on curve (6) where a tangent line on point Q is drawn in order to find R:

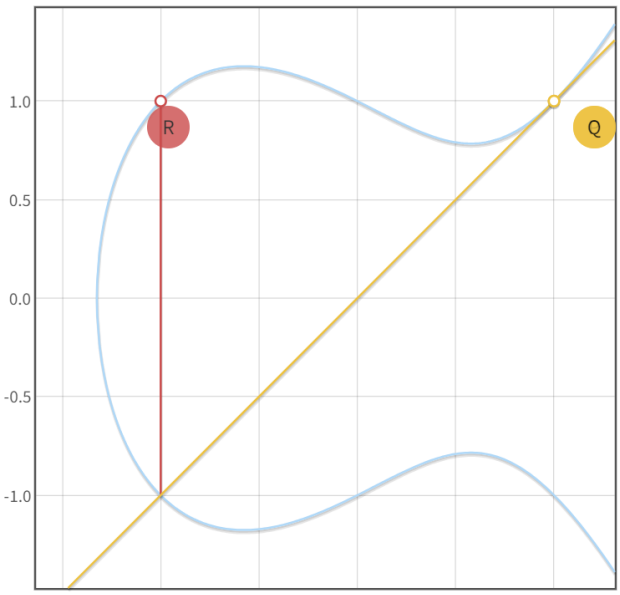


Figure 4: Point Doubling for R = Q + Q on Curve (6) Over Real Numbers

End of Example 3.

Subtraction of Two Points over Real Numbers [17]

Let's define the following points:

- (x_P, y_P): Coordinates for point P
- $(x_{Q,}\ y_Q)$: Coordinates for point Q
- $(x_{R,}\ y_R)$: Coordinates for point R

Then, subtraction of two points is denoted by R = P – Q, which is equivalent to:

$$R = (x_P, y_P) + (x_{Q,} - y_Q) \quad (7)$$

Now equation (7) represents an addition and has the same equations as point addition, so if $P \neq Q$ then (4) is used, and if $P = Q$ then (5) is used.

Definition of Group [12]:

In mathematics, a group is a set G that has a binary operation * that allows it to combine any two of its elements, a and b, to create a new element. Four essential qualities of this structure are closure, associativity, identity, and invertibility.

1. Closure: The outcome of the operation a * b is also in G for every a, b in G.
2. Associativity: The operator * is associative, which means that the equation (a * b) * c = a * (b * c) holds for all a, b, and c in G.
3. Identity Element: There is an identity element e in G such that the equation e * a = a * e = a hold for each element an in G.

4. Inverse Element: Given any element an in G, there is an inverse element b in G such that e is the identity element and a * b = b * a = e.

Elliptic Curve is Considered as Group Structure [17]:

Because the set of points on an elliptic curve and a defined addition operation satisfy all the requirements for a group, elliptic curve can be considered as a group.

1. Set of Points and Operation: The set G is made up of the points on an elliptic curve and the point at infinity O, which represent the identity element. Any two points on the curve can be combined to create a new point on the curve using the geometrically specified addition operation.

2. Closure: Any two points P and Q on the elliptic curve have a point on the curve associated with their sum, P + Q. This is because the intersection of lines and reflections on the curve is used to describe the addition operation.

3. Associativity: On elliptic curve points, the addition operation is associative. This indicates that the equation (P + Q) + R = P + (Q + R) is true for any three points P, Q, and R. The geometric architecture of point addition guarantees this characteristic.

4. Identity Element: The identity element is the point at infinity O. P + O = O + P = P is true for any point P on the curve.

5. Inverse Elements: Given any point P = (x, y) on the curve, P + (-P) = O can be found for its inverse, -P = (x, -y).

2.2.2 Elliptic Curves Over Finite Field F_p [17]

A non-singular elliptic curve over a finite field F_p is described by the equation:

$$y^2 \ mod \ p = x^3 + ax + b \ mod \ p \tag{8}$$

where a and b are coefficients that make the curve non-singular (it has no cusps or self-intersections) by ensuring:

$$4a^2 + 27b^2 \bmod p \neq 0 \tag{9}$$

and p is a prime number related to the finite field F_p, which is a field containing a finite number of elements, these elements are numbers from 0 to p−1 obtained by performing arithmetic operations (addition, subtraction, multiplication, and division) modulo p, where every element x in a finite field satisfies the equation x^p= x mod p, which impacts the structure and solutions of equations defined over the field.

Example 4: Verifying elliptic curve parameters

Define the Curve:

The equation for the elliptic curve is as in (8):

$$y^2 \bmod 23 = x^3 + x + 1 \bmod 23 \tag{10}$$

Now, a = 1 and b =1.

Verify the discriminant (9):

$\Delta = 4(1)^3 + 27(1)^2 \bmod 23 = 4(1) + 27 \bmod 23 = 4 + 27 \bmod 23 = 31 \bmod 23 = 8 \neq 0$.

As Δ doesn't equal to 0, the curve is nonsingular.

Figure 5 illustrates an elliptic curve (10) points:

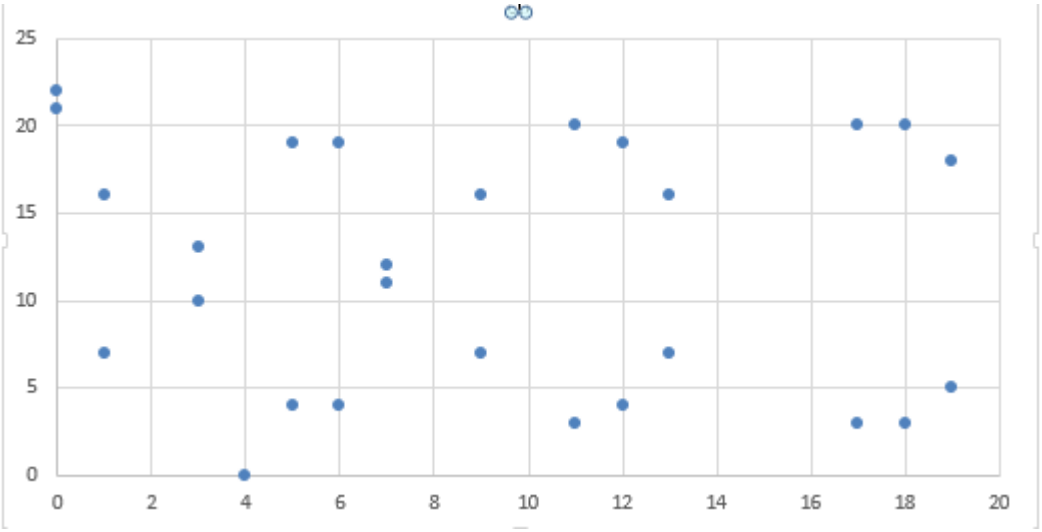


Figure 5: Elliptic Curve (10) Over F_{23}:(where p=23)

End of Example 4.

Addition of Two Points in ECC over Finite Field [17]

Addition of two points P and Q on an elliptic curve (R=P+Q) over F_P:

- $(x_P\ y_P)$: Coordinates for point P
- $(x_{Q,}\ y_Q)$: Coordinates for point Q
- $(x_{R,}\ y_R)$: Coordinates for point R
- λ: normally it is the slope for straight line PQ, but no slope available in finite field, so it is a variable used to obtain coordinates of R.

Point Addition (P ≠ Q): For P not equal to Q, find:

$$\lambda = \frac{y_Q - y_P}{x_Q - x_P} \bmod p \tag{11}$$

$$x_{R,} = \lambda^2 - x_P - x_Q \bmod p$$

$$y_R = \lambda(x_P - x_{R,}) - y_P \bmod p$$

Point Doubling (P = Q): If P equals Q, determine:

$$\lambda = \frac{3x_P^2 + a}{2y_P} \ mod\ p \tag{12}$$

$$x_R = \lambda^2 - 2x_P \ mod\ p$$

$$y_R = \lambda(x_P - x_{R,}) - y_{P\ mod\ p}$$

Extended Euclidean Algorithm [18]:

The Extended Euclidean Algorithm computes the greatest common divisor (GCD) of two integers while also finding coefficients that express the GCD as a linear combination of these integers. This algorithm is essential in applications such as finding modular inverses in cryptography.

The Extended Euclidean Algorithm (EEA) [18] is represented in Algorithm 1:

Algorithm 1: Extended Euclidean Algorithm [18]

Extended Euclidean Algorithm
Inputs: 2 integer numbers $a\ and\ b$. Output: t_i (parameter used to obtain GCD, which represented in $as + bt = Gcd(a,b)$). 1. Initialization step: $r_0 = a, r_1 = b, s_0 = 1, s_1 = 0, t_0 = 0, t_1 = 1, i = 1$ While these parameters are represented in $as + bt = Gcd(a,b)$ 2. Iteration step i: 2.1. $q_i = \left\lfloor \frac{r_{i-1}}{r_i} \right\rfloor$, while this is a floor division which means dividing two integers and rounding down to the nearest integer. 2.2. $r_{i+1} = r_{i-1} - q_i r_i$ 2.3. $s_{i+1} = s_{i-1} - q_i x_i$ 2.4. $t_{i+1} = t_{i-1} - q_i y_i$ 2.5. $i = i + 1$ 2.6. Repeat step 2 until $r_{i+1} = 0$.

2.7. If $r_i = 1$, then return t_i
2.8. Otherwise, GCD isn't available for $a \; and \; b$.

Example 5: Here an example for point addition is done.

For the following parameters for the Elliptic curve (8):

- a = -1
- b = 1
- p = 17

Thus, the equation of the curve becomes:

$$y^2 \; mod \; 17 = x^3 - x + 1 \; mod \; 17 \qquad (13)$$

The following two points are on the curve (13):

- P = (x_P, y_P) = (1,1)
- Q = (x_Q, y_Q) = (5, 6)

Calculate R =(x_R, y_R) = P + Q.

Steps to find R using equation (11):

1. Calculate λ:
   $\lambda = \frac{y_Q - y_P}{x_Q - x_P} \; mod \; 17 = \frac{6-1}{5-1} = \frac{5}{4} \; mod \; 17$

To find $\frac{5}{4} \; mod \; 17$, we need the modular inverse of 4 modulo 17. The modular inverse of 4 modulo 17 can be found using the Extended Euclidean Algorithm [17]. The most common use of it is finding modular inverses.

Now, finding Modular Inverse of 4 Modulo 17, where the starting step is using EEA in Algorithm 1:

a. Initialization Step

   $r_0 = 17, r_1 = 4, s_0 = 1, s_1 = 0, t_0 = 0, t_1 = 1$

b. First Iteration

$$q_1 = \left\lfloor \frac{17}{4} \right\rfloor = 4$$

$$r_2 = r_0 - q_1 r_1 = 17 - 4(4) = 1$$

$$s_2 = s_0 - q_1 s_1 = 1 - 4(0) = 1$$

$$t_2 = t_0 - q_1 t_1 = 0 - 4(1) = -4$$

c. Second Iteration

$$q_2 = \left\lfloor \frac{4}{1} \right\rfloor = 4$$

$$r_3 = r_1 - q_2 r_2 = 4 - 4(1) = 0$$

$$s_3 = s_1 - q_2 s_2 = 0 - 4(1) = -4$$

$$t_3 = t_1 - q_2 t_2 = 1 - 4(-4) = 17$$

The algorithm terminates since r_3 = 0. The GCD is r_2 = 1, and the coefficients are s_2 = 1 and t_2 = -4.

17=4·4+1

1=17−4·4

Thus, the modular inverse of 4 modulo 17 is -4 and -4 mod 17 = 13.

So, the value for λ is:

λ = 5(13) mod 17 = 65 mod 17= 14

2. Compute x and y for R using (11):

- $x_R = \lambda^2 - x_P - x_Q \text{ mod } (17) = 14^2 - 1 - 5 \; mod \; 17 = 190 \; mod \; 17 = 3$
- $y_R = \lambda(x_p - x_R) - y_P \; mod(17) = 14(1-3) - 1 \; mod \; 17 = -29 \; mod \; 17 = 5$

Now, the result of R is

$$R = P + Q = (3, 5). \quad (14)$$

Let's prove that R in (14) is on the curve (13):

Calculations:

1. Substitute x=3 and y=5 into (13):

$LHS = y^2$ mod 17 =5^2 mod 17 = 25 mod 17 = 8

$RHS = x^3$ −x+1 mod 17= 3^3 −3+1 mod 17= 25 mod 17 = 8

Thus, R= (3, 5) is a point on the curve (13).

End of Example 5.

In the proceeding parts, all elliptic curves are related to curve (8).

Scalar Multiplication in ECC [19]

The results of scalar multiplication k × P, is the addition of a point P to itself k times, for instance 4 × P= P + P + P + P. It employs the double-and-add method.

The pseudo code for Scalar Multiplication represented in Algorithm 2:

Algorithm 2: Scalar Multiplication of Point P by Scalar k Algorithm [19]

Scalar Multiplication of Point P by Scalar k Algorithm

function ScalarMultiplication(P, k)
Input:
P is a point on the elliptic curve (8)
k is a scalar (positive integer)
Output:
Q is the result of $k \times P$
Step 1: Q = O # O is the identity element.
Step 2: N = P
Step 3: while k > 0 do
a. if k is odd then
Q = Q + N refers to point addition using (11)
end if
b. N = N + N # or $2 \times N$ refers to point doubling using (12)
c. k = $\left\lfloor \frac{k}{2} \right\rfloor$
end while
Step 4: return Q
end function

Example 6: This example covers the scalar multiplication represented in Algorithm 2

Let's use the same settings as in Example 5.

Thus, curve (13) is used.

Let's have the point P = ($x_{P,}$ y_P) = (1, 1) and k = 3.

$2 \times P$ should be calculated.

$2 \times P = (x_{2P}, y_{2P})$

R= $k \times P$ =3 $\times P = (x_{3P}, y_{3P})$. We want to find R.

Steps to find R:

1. Find $2 \times P$ using point doubling in (12):
    - $\lambda = \frac{3x_P^2 + a}{2y_P}$ mod17 = $\frac{3*1^2-1}{2*1}$ mod 17 = 1 mod 17 = 1
    - $x_{2P} = \lambda^2 - 2x_P$ mod17 $= 1^2 - 2 * 1$ mod 17 $= -1$ mod 17 = 16
    - $y_{2P} = \lambda(x_P - x_{2P}) - y_P$ mod 17 = 1(1 − 16) − 1 mod 17 = −16 mod 17 = 1

      $Now, 2 \times P = (x_{2P}, y_{2P}) = (16,1)$
2. Find 3P by adding P + 2P because k is odd using (11):
    - $\lambda = \frac{y_{2P} - y_P}{x_{2P} - x_P}$ mod (17) = $\frac{1-1}{16-1}$ mod (17) = 0 mod 17 = 0
    - $x_{3P} = \lambda^2 - x_P - x_{2P}$ mod (17) = $0^2 - 1 - 16$ mod 17 = −17 mod 17= 0
    - $y_{3P} = \lambda(x_P - x_{3P}) - y_P$ $mod(17)$ = 0(1-0) – 1 mod 17 = −1 mod 17 = 16

Finally, R= 3 $\times P$ = (0, 16).

End of Example 6.

2.2.3 Background Information Used for ECC

Before moving forward, some information should be introduced before discussing ECC encryption and ECC decryption.

Section 1 explains the base point G and the order n for subgroups of $n \times G$, where $n \times G$ represents the identity element O, which has order n, and n represents the number of points represented inside a subgroup. Section 2 discusses the conversions of data from binary to decimal, hexadecimal to binary, and vice versa. Section 3 introduces authenticated encryption (AE). At the end, section 4 introduces mapping points to elliptic curve and the reverse mapping of points to elliptic curve.

1. Base Point G and Order n in Elliptic Curve [17]

The base point G in elliptic curve cryptography is a base point on the curve that acts as the generator for the group of points on the curve. Because it is the starting point for all subsequent group points by scalar multiplication discussed in Algorithm 2, this base point is essential. The smallest positive integer such that $n \times G = O$, where O is the identity element (the point at infinity), is the order n of the base point G. {O, G, $2 \times G$, $3 \times G$,….. $(n-1) \times G$} provides points that together make up a cyclic subgroup, which means it is a repetitive cycle, so when obtaining the scalar multiplication of $(n+1) \times G$, the answer must be $1 \times G$, so that's why it's called cyclic subgroup of the elliptic curve group.

Example 7: Computing order n for subgroup $n \times G$.

Let's use equation (10).

G = (x_G, y_G)= (13, 7)

Now we need to calculate the order n for the subgroup $n \times G$ of the elliptic curve (10). Now scalar multiplication is done in each round using Algorithm 2, and because a lot of calculation are done, an online calculator [20] is used for this example:

For n = 1: G = (13, 7)

For n= 2: $2 \times G = (5, 4)$

For n= 3: $3 \times G = (17, 3)$

For n= 4: $4 \times G = (17, 20)$

For n= 5: $5 \times G = (5, 19)$

For n= 6: $6 \times G = (13, 16)$

For n= 7: $7 \times G = O = (7, infinity)$

The identity point O is calculated at n=7, which means the order for this subgroup of elliptic curve (10) equals to 7.

End of Example 7.

2. Conversions (Hex, Decimal, and Binary)

This part will discuss 4 types of conversions, which are:

1. Decimal to Binary
2. Binary to Decimal
3. Hexadecimal to Binary
4. Binary to Hexadecimal

Decimal to Binary Conversion

The process of converting a base-10 (decimal) number to a base-2 (binary) value is known as decimal to binary conversion. To accomplish this, divide the decimal value by two several times, then note the remaining amount. The binary version of the decimal number is made up of the remainders, which are read in reverse order. Steps for conversion are discussed in Algorithm 25 in Appendix C:

Example 8: This example is discussing the conversion of 13 from decimal to binary using Algorithm 25 in Appendix C

The decimal number 13 will be divided by 2, and the remainder is appended in the result from right to left, and then the same is done for quotient as the given decimal number 13, and the steps end when quotient reached zero:

13 ÷ 2 = 6 and the remainder is 1, then append 1 to the result: 1

6 ÷ 2 = 3 and the remainder is 0, then append 0 to the result: 01

3 ÷ 2 = 1 and the remainder is 1, then append 1 to the result: 101

1 ÷ 2 = 0 and the remainder is 1, then append 1 to the result: 1101

Quotient = 0, then we stop here.

So, the binary representation for 13 is 1101.

End of Example 8.

Binary to Decimal Conversion

The process of converting a base-2 (binary) number into a base-10 (decimal) value is known as binary to decimal conversion, which is discussed in Algorithm 26 in Appendix C.

Example 9: Convert the binary value 1101 to decimal using Algorithm 26 in Appendix C

The steps of Algorithm 26 are:

Step 1: Set the length for the given binary number: length of 1101: 4

Step 2: 4 iterations will be done from $i = 0$ till $i = 3$ to compute the result:

a. For $i = 0$:

   Result = $2^3 * 1$

b. For $i = 1$:

   Result = $2^3 * 1 + 2^2 * 1$

c. For $i = 2$:

   Result = $2^3 * 1 + 2^2 * 1 + 2^1 * 0$

d. For $i = 3$:

   Result = $2^3 * 1 + 2^2 * 1 + 2^1 * 0 + 2^0 * 1$

Step 3: Result = 13

Therefore, the decimal representation of the binary number 1101 is 13.

End of Example 9.

Hex to Binary Conversion

Converting a base-16 (hexadecimal) number to a base-2 (binary) value is known as hexadecimal to binary conversion, which is discussed in Algorithm 27 in Appendix C.

Example 10: Convert the hexadecimal value 2F to binary using Algorithm 27 in Appendix C

By using Algorithm 34 and the mapping object provided in step 1 of Algorithm 34:

We should pass through all bits in 2F, one by one, then write its representation in binary using mapping object provided in step 1 of Algorithm.

2: 0010

F: 1111

At the end, appending them in the results from left to right.

So, 2F representation in binary is: 00101111.

End of Example 10.

Binary to Hex Conversion

The process of converting a base-2 (binary) number to a base-16 (hexadecimal) number is known as binary to hexadecimal conversion, which is discussed in Algorithm 28 in Appendix C.

Example 11: Convert the binary value 11110000 to hexadecimal using Algorithm 28 in Appendix C.

By using Algorithm 28 and the mapping object in step 1 of Algorithm 28:

At the beginning, we should check if the length of the binary value is multiple of four. 11110000 has length of eight which is a multiple of four, while if it isn't multiple of four, padding zeros on left should be added until it becomes multiple of four.

Going on, we should separate all bits in 1111000 in group of four bits starting from left to right, then write its representation in hexadecimal using mapping object provided in step 1 of Algorithm 28.

1111: F

0000: 0

At the end, we should append the hexadecimal representations from left to right.

So, 11110000 represents in hexadecimal: F0.

End of Example 11.

3. Mapping Points to EC and Reverse Mapping to EC

Mapping points to EC is a very important method which is used to convert the original message to a point mapped on elliptic curve (8).

Mapping points to EC, in addition of encoding the message, are represented in Algorithm 3:

Algorithm 3: Steps for Mapping Algorithm [4]

Mapping Algorithm in [4]
Input: Message consisting of characters belonging to extended ASCII set. Output: Distinct points (X, Y) on the Elliptic curve $Ep(a, b)$ corresponding to the Message. Steps of the algorithm: Step 1: Begin Step 2: a: Consider M characters of the message at a time b: Convert each character into 8-bit ASCII codes c: Insert each 8-bit binary number into an array of length M * 8 bits Step 3: Append N 0's at the end of the array Step 4: Extract the (M * 8 + N)-bit number from the array, convert it to a decimal number, and store it in X Step 5: a: Find Y from the equation Y^2 ≡ X^3 + aX + b mod p b: If Y does not have a solution increment X by 1 and go to step 5a Step 6: After obtaining Y use the distinct point (X, Y) for encryption using ECC Step 7: Repeat step 2 to step 6 until the end of message Step 8: End

The input should include ECC parameters (a, b, and p) in addition to the mentioned input which is the original message.

The input is the original message, while in step 2 in Algorithm 3, only M characters are taken from the original message, while

$$M \leq \left\lfloor \frac{size\ of\ (p)-8}{8} \right\rfloor \tag{15}$$

and M is equal to the specified equation except in the last array, while it is possible to be less than the specified equation. In the second round M characters will be taken from the remaining message and it will keep going till the original message ends.

N will be equal to 8 bits of 0s, unless the number of characters taken is less than (15) characters, because N represents the padding bits (on the right side) until the array of binary numbers reaches the size of p in total. N bits are added to the right side of each array because while searching for y value of x, x may be incremented by one several times, so the value of x may change, so the added N (N= 8 bits except in the last array, it may be more) bits on the right side will be removed in the decoding process in step 5 in Algorithm 4, which leads to neglect the effect of changing the value of x, if the incremental step is used.

In Algorithm 3, every array will contain size of p bits, representing M characters of the original message, which will be converted from binary to decimal value which represents the x-coordinate of the point. Then the y value will be calculated and then it will be a mapped point on elliptic curve. This process will be repeated until the original message is all taken inside array/s and the output will be represented as mapped point/s on the elliptic curve.

Example 12: This example is explaining the encoding process and computing for mapped point using Algorithm 3.

Let's use the equation of the standard EC curve represented in Appendix D, while the parameters of it is shown in Figure 73 in Appendix D.

And the original message is: "Hello, my name is Bella"

Find the Mapped Point using Algorithm 3:

Step 1: Begin

Step 2: a. by using (15) $M \leq \left\lfloor \frac{198-8}{8} \right\rfloor \leq 23$ characters, and the original message exactly equals to 23 characters, which means M = 23 characters and by it covers the whole message.

b. c. Convert each character using Ascii table to binary and concatenate them in an array:

01001000 01100101 01101100 01101100 01101111 00101100 00100000 01101101 01111001 00100000 01101110 01100001 01101101 01100101 00100000 01101001 01110011 00100000 01000010 01100101 01101100 01101100 01100001

Step 3: Append N 0s to the right side of the array, and here N = 8 bits because 8 bits are remaining to cover 192 bits array:

01001000 01100101 01101100 01101100 01101111 00101100 00100000 01101101 01111001 00100000 01101110 01100001 01101101 01100101 00100000 01101001 01110011 00100000 01000010 01100101 01101100 01101100 01100001 00000000

Step 4: Convert the binary array to a decimal number and store it in x:

x = 1775149307253347677983714827454043352226162973166734696704

Step 5: Find y using secp192k1 curve in Appendix D (searching for y code is discussed in Figure 72 in appendix B):

y = 424404154542925890313407247864906067739880166737495053409 3

Figure 6 shows the output for the code in Appendix B (Figure 72):

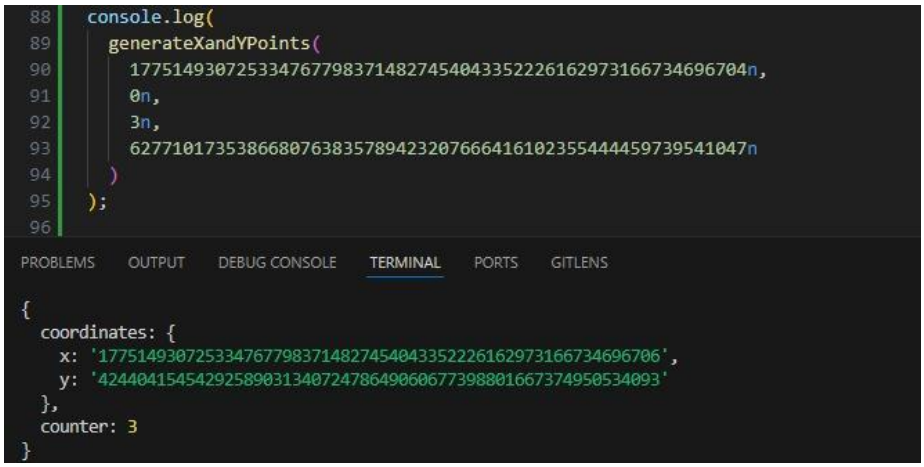


Figure 6: Output Point for Point Searching Code

Step 6: Return a mapped point on EC

(x = 177514930725334767798371482745404335222616297316673469670 4, y = 424404154542925890313407247864906067739880166737495053409 3)

Step 7: No remaining characters so stop here.

Mapped Point =

(177514930725334767798371482745404335222616297316673469670 4,

424404154542925890313407247864906067739880166737495053409 3)

End of Example 12.

Reverse Mapping Points to EC [4]:

Reverse mapping points to EC is represented in Algorithm 4:

Algorithm 4: Steps for Reverse Mapping Algorithm [4]

Reverse Mapping Algorithm in [4]
Input: Distinct points (X, Y) on the Elliptic Curve $Ep(a, b)$. Output: Original message sent by sender. Steps of the algorithm: Step 1: Begin Step 2: Ignore 'Y' coordinate Step 3: Convert 'X' coordinate into binary number and ignore the last N bits Step 4: Extract the rest of the bits and put it in a bit array Step 5: Start from the right most bit. Consider 8 bits from the array at a time: this 8-bit is nothing but the original alpha-numeric ASCII character which formed the original plaintext. Repeat this step until M characters are retrieved Step 6: Repeat the earlier steps for each cipher point pair sent by the sender Step 7: End

Algorithm 4 shows the reverse mapping of the point on elliptic curve. The input is the decrypted mapped points and the parameters of elliptic curve (a, b and p). While N

and M have the same conditions like the previous Algorithm 3. In step 5 in Algorithm 4, the added N bits on the right are removed, which proves the usage of adding N bits on the right in Algorithm 3.

The output is the decrypted message.

Example 13: This example is explaining the decoding (reverse mapping) using Algorithm 3.

Find the original message:

Mapped Point =

(177514930725334767798371482745404335222616297316673469670 4,

424404154542925890313407247864906067739880166737495053409 3)

Let’s use the equation of the standard EC curve represented in Appendix D, while the parameters of it is shown in Figure 73 in Appendix D.

Find the original message steps using Algorithm 4:

Step 1: Begin

Step 2: Let’s ignore y coordinates

Step 3: Select the x coordinate, and convert it to binary:

X=177514930725334767798371482745404335222616297316673469670 4

Binary (X) =

010010000110010101101100011011000110111100101100001000000110110101111001001000000110111001100001011011010110010100100000011010010111001100100000100001001100101010110110001101100011000010000000

Step 4: Discard the 8 bits on the right:

Binary without last 8 bits =

01001000011001010110110001101100011011110010110000100000011011010111
10010010000001101110011000010110110101100101001000000110100101110011
00100000010000100110010101101100011011000110000 1

Step 5: Obtain the message using Ascii table for every 8 bits starting from right:

M= “Hello, my name is Bella”

Step 6: No more mapped points, so the original message is:

“Hello, my name is Bella”

End of Example 13.

4. Elliptic Curve Digital Signature Algorithm (ECDSA) [21]

A cryptographic technique called the Elliptic Curve Digital Signature Algorithm (ECDSA) is used to guarantee the integrity and validity of digital messages and documents. Elliptic curve cryptography (ECC) is used in the Digital Signature Algorithm (DSA [21]) variation known as ECDSA. It is more effective than DSA since it offers the same level of security but with shorter key lengths [21].

ECDSA is a type of digital signature that makes it possible to confirm the authenticity and integrity of a message. There are two sections to it which are signing the message, and verifying the message.

Here are some applications which use ECDSA:

Cryptocurrencies: For signing transactions, for instance: Bitcoin.

Smart Cards: For secure authentication.

Software Signing [22]: To verify the authenticity of software updates.

ECDSA is used for signing and verifying the message.

ECDSA signing Algorithm 5:

Algorithm 5: Signing Message Algorithm [21] and [3]

Signing Message Algorithm in [21] and [3]

Input:
- The Sent Message M_sent,
- Sender's Private Key ds,
- Curve Generator Point G,
- ECC parameters (a, b and p)
Output:
- The signed encrypted point (r, s)
 	
1. Sender: obtain e = HASH(M_sent); // HASH is hash function
2. Sender: obtain z = most significant bit of e;
3. Sender: select k;
4. Sender: obtain r = x mod p where x is (x, y) = k.G and r ≠ 0;
5. Sender: if r == 0 go to step 3;
6. Sender: obtain s = (z + ds * r) * k^(-1) mod p if s == 0 go to step 3;
7. Sender: the pair (r, s) is the signature; // Sender sends (M_sent and (r, s) to the receiver)
8. Return (r, s);

ECDSA verifying Algorithm 6:

Algorithm 6: Verifying Message Algorithm [21] and [3]

Verifying Message Algorithm in [21] and [3]
Input: - M_sent (Sender's message which sent in step 7 in Algorithm 5), - (r, s) (signature obtained and sent to receiver in Algorithm 5), PU_s (public key for sender), - Curve Generator Point G, - ECC parameters (a, b and p) Output: - Verified message //(Verified message will return true or false) 1. Recipient: verify (r, s) are integers in [1, p − 1]; 2. Recipient: obtain e = HASH(M_sent); 3. Recipient: obtain z = most significant bit of e; 4. Recipient: obtain u_1 = es^(-1) mod p; 5. Recipient: obtain u_2 = rs^(-1) mod p; 6. Recipient: calculate (x_1, y_1) = u_1 * G + u_2 * PU_s; 7. Recipient: verify r ≡ x_1 mod p; 8. Return Verified message;

The sender uses signing message Algorithm 5 to obtain the signature (r, s), while the receiver uses verifying message Algorithm 6 to verify that $r \ mod \ p \ = \ x_1 \ mod \ p$ or not, which proves the verification of the message or not.

Both r and s should be integers between 1 and p-1, both included.

$e \ and \ z$ in both Algorithm 5 and Algorithm 6 are equal because they have the same formula.

Step 3 in signing Algorithm 5, k is a random number between 1 and n-1, both included.

In signing Algorithm 5, the values of r and s are:

- $r = k \times G$, while r is the x coordinate of $k \times G$ mod p, and $k \times G$ is obtained by using scalar multiplication of point G on curve (9) with scalar k using algorithm 2, also G is the base point of the curve (9). If r = 0, another k value should be generated and the calculation will be done again.
- $s = (z + d_s * r)k^{-1}$, while $s \neq 0$. Also, if s = 0, then a new value of k should be chosen.

Let's prove the correctness of verification in Algorithm 6 :

Now let's prove that $x_1 = k \times G$ also by the following:

$u_1 = s^{-1} \times z \ mod \ p$

$u_2 = s^{-1} \times r \ mod \ p$

$$s = \frac{z + d_s \times r}{k}$$

$PU_s = d_s \times G$

$(x_1, y_1) = \ u_1 \times G + \ u_2 \times PU_s$

$(x_1, y_1) = \ s^{-1} \times z \ \times G + \ s^{-1} \times r \ \times PU_s$

$(x_1, y_1) = \ s^{-1} \times z \ \times G + \ s^{-1} \times r \ \times d_s \times G$

$(x_1, y_1) = \ s^{-1} \times (z + r \times d_s) \ \times G$

$$(x_1, y_1) = \frac{k}{z + d_s \times r} \times (z + r \times d_s) \times G = k \times G$$

So, $r = x_1$.

End of proof.

2.2.4 ECC Encryption/Decryption Model 1 [4]

Encryption model 1 in Figure 7:

1. a. Bob is the sender.

   b. Alice (Receiver): Generates a private key d_A and computes the corresponding public key Q_A (Algorithm 7).

2. Message Encryption: Bob encrypts a plaintext message Pt using Alice's public key Q_A and sends the encrypted message C along with the point $P = k \times G$ (step 2 in Algorithm 8) to Alice.

3. Message Decryption: Alice receives from Bob the Cipher text C (step 4 in Algorithm 6) and point $P = k \times G$ (Step 2 in Algorithm 8), and then computes point $dP = d_A \times P$ (step 1 in Algorithm 9) and then computes the original message $M = C - dP$ (step 2 in Algorithm 9).

Figure 7 illustrates the steps for ECC encryption and decryption:

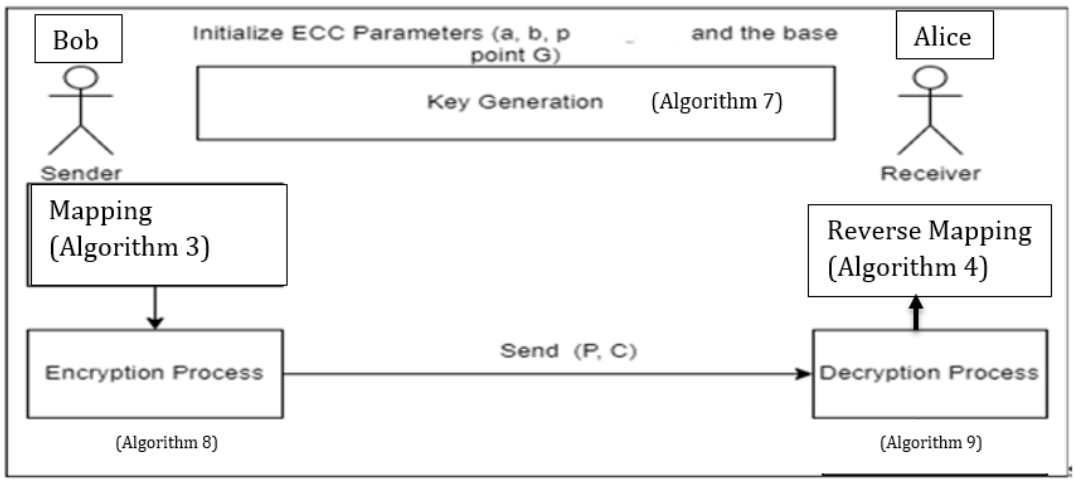


Figure 7: ECC Encryption and Decryption Model 1 [4]

Key Generation:

Public and Private Keys: The private key is generated by receiver—a randomly picked number, denoted d, and a public key denoted by Q, which is calculated by multiplying the private key with the base point G of the elliptic curve (In Elliptic Curve Cryptography (ECC), the base point G referred to as the fixed generator point, is a point with cyclic properties, meaning it generates a cyclic subgroup of the elliptic curve. Therefore, the public key Q = d×G is one point in this subgroup of elliptic curve, more information is discussed in 2.2.3.

ECC offers the high security using small key sizes. For example, with ECC, a 210-bit key is thought to offer the same strength of security as a 2048-bit key in RSA [23].

The pseudo code for key generation is represented in Algorithm 7:

Algorithm 7: Key Generation for Receiver Algorithm [4]

Key Generation Algorithm in [4]

KeyGeneration(a, b, p , G)
// Inputs: a, b, p - coefficients of the elliptic curve; G - base point
// Output: Q_A – receiver's public key (point on the curve (8)); d_A- receiver's private key (integer)
begin
// Step 1. Generate the receiver's private key d_A
d_A = any random number in range (2, p-1) // 2 and p – 1 are included
// Step 2. Compute the receiver's public key Q_A
$Q_A = d_A \times$ G // it is the operation to multiply a point G by an integer d_r according
// to Algorithm 2.
// Step 3. Return the sender's public key and private key
return (Q_A, d_A)
end

Encryption Process in the Encryption/Decryption Model 1:

The encryption process is shown in Algorithm 8:

Algorithm 8: ECC Encryption Process Algorithm [4]

ECC Encryption Algorithm in [4]

```
EllipticCurveEncryption(a, b, p, G, Pt, RecipientPublicKey)

// Inputs:
// a, b, p - coefficients and prime of the elliptic curve
// G - base point (a well-known point on the elliptic curve)
// Pt - Plaintext (represented as a point on the elliptic curve)
// RecipientPublicKey - public key of the recipient

begin

    // Step 1. Choose a Random Key:
    // Sender selects any random integer k from the range [2, p-1]
    k = choose a random integer between 2 and p-1

    // Step 2. Calculate Points on Curve:
    // Sender computes the point P = k * G, where G is the base point on the curve
   //(scalar multiplication by using Algorithm 2)
    P = k * G // (scalar multiplication by using Algorithm 2)

    // Step 3. Calculate the SharedSecret (shared secret key, used for
// encryption):
    // Sender computes k * Q, where Q is the public key of the recipient
   //(scalar multiplication by using Algorithm 2)
    Q = RecipientPublicKey
    SharedSecret = k * Q // (Algorithm 2)

    // Step 4. Construct CipherText (represented as a point):
    // Add the plaintext point Pt to the SharedSecret by using (11)
    C = Pt + SharedSecret // point addition by using (11)

    // Step 5. Sending Process:
    // The sender sends the data (P, C) to the receiver
    SendData(P, C)
End
```

Example 14: A key generation will be done in this example by Algorithm 7

Let us consider an elliptic curve (8) with parameters: a=2, b=2 over F_p, p=17 as in (16):

$$y^2 mod\ 17 = x^3 + 2x + 2 \bmod 17 \tag{16}$$

and base point G is given as G = (5, 1).

Bob wants to send a message to Alice. Key Generation according to Algorithm 7:

Step 1 in Algorithm 7:

Private Key: Alice selects the private key randomly, let's consider it as d_A= 6.

Step 2 in Algorithm 7:

Public Key: Alice computes her public key Q_A as $d_A \times G$. Multiplying them will give Q_A = 6 × (5, 1) = (16, 13) using scalar multiplication of point G on curve (16) with scalar d_A in Algorithm 2 (similar calculations are done in Example 6). Calculations are done in Appendix B.

End of Example 14**.**

In Example 15, an example is given for encrypting message represented by point.

Example 15: Encrypting Message by Algorithm 8

Let us consider ECC encryption of the plaintext represented by M= (9, 1) being a point that satisfies the curve (16) used in Example 14. Let Bob wish to send the plaintext message M. Mapping the message to its point isn't from the encryption steps and it is introduced in 2.2.3 in Algorithm 3.

To encrypt M= (9, 1), The steps for encryption process should be followed as mentioned in Algorithm 6:

Step 1 in Algorithm 8: Bob chooses an arbitrary integer k (random number) = 2, k should be in between 2 and 16, both included.

Step 2 in Algorithm 8: Compute P = $k \times G$: $k \times G = 2 \times (5, 1) = (6, 3)$, using scalar multiplication of point G on curve (16) with scalar k in Algorithm 2 (similar calculations are done in Example 6).

Step 3 Algorithm 8: Compute $k \times Q_A$: k×Q_A= 2 × (16, 13) = (9, 16), using scalar multiplication of point Q_A on curve (16) with scalar k in Algorithm 2 (similar calculations are done in Example 6). Calculations are done in Appendix B.

Step 4 in Algorithm 8: Encrypt the Message: Bob encrypts the message: C = M+

$k \times Q_A$= (9, 1) + (0, 11) = (0, 6), and it is done using point addition of two points, discussed in (11). Calculations are done in Appendix B.

Step 5 in Algorithm 6: Bob sends Alice (P, C) = ((6, 3), (0, 6)).

End of Example 15.

ECC Decryption Process:

In this section, ECC decryption process is represented in Algorithm 9 (ECC decryption):

Algorithm 9: ECC Decryption Process Algorithm [4]

ECC Decryption Algorithm in [4]

// ECC Decryption Algorithm
// Input: Encrypted data (P, C), private key d (recipient's private key), EC parameters a, b, and // p
// Output: Original message point M
 	
// Step 1: Calculate the Shared Secret Key
// The receiver computes dP from the point P using the private key d
// Given P = k * G, then dP = d * k * G
 	
1. dP = d * P // dP is the shared secret key
 	
// Step 2: Extract the Original Message
// By subtracting dP from C, retrieve the original message point M
// For subtraction: (x_P, y_P) + (x_Q, -y_Q)
 	
// The subtraction operation on elliptic curve involves computing the negative // of the point (x_Q,
// y_Q) modulo p
2. M = C - dP // Point Subtraction using (7)
 	
Return M

Description:

Upon receiving the encrypted data (P, C), the recipient performs the following steps to decrypt the message:

Step 1: Calculate the Shared Secret Key:

The receiver computes $d_A \times P$ from the point P using the private key d. Given that P = $k \times G$ from Algorithm 1b, then $d_A \times P = d_A \times k \times G$.

Let's prove the correctness of the decryption:

$dP = d_A \times P$ (step 1 in Algorithm 9) and $P = k \times G$ (step 2 in Algorithm 6), which means

$dP = d_A \times k \times G$

$SharedSecret = k \times Q_A$ (step 3 in Algorithm 8) and $Q_A = d_A \times G$ (step 1 in Algorithm 7), which means $SharedSecret = k \times d_A \times G$

Step 2: Extract the Original Message

$M = C - dP = C - d_A \times k \times G$ (step 2 in Algorithm 9)

C = Pt + $SharedSecret$(step 4 in Algorithm 8), then Pt = C - $SharedSecret$, which leads to $Pt = C - d_A \times k \times G$

Pt = M, which proves the correctness of encryption/decryption model 1.

Example 16: Decryption Process by Algorithm 9

This example is covering the decryption of the encrypted data sent by Bob to Alice in Example 15.

Now, the decryption for the sent points (C, P) in Example 15 will be shown here.

Decryption:

Curve (16) is used.

Alice is getting (P, C) = ((6, 3), (0, 6)).

Step 1 in Algorithm 9: Compute $d_A \times P$ = 6× (6, 3) = (0, 11), using scalar multiplication of point P on curve (16) with scalar d_A in Algorithm 2 (similar calculations are done in Example 6). The calculations are provided in Appendix B.

Step 2 in Algorithm 9: Decrypt the Message: C- d_A ×P= (0, 6) - (0, 11) = (9, 1) on curve (16) using (7), and then point doubling using (12). The calculations are provided in Appendix B.

The message point M= (9, 1) is the plaintext retrieved by Alice.

End of Example 16.

Reversing the mapped points in order to obtain a decrypted message of characters is done in section 2.2.3 in Algorithm 4 and the steps of Algorithm 4 are shown in Example 13.

The end of encryption/decryption model 1.

2.2.5 ECC Encryption/Decryption Model 2 [3]

Figure 8 shows the overall structure for ECC Encryption/Decryption model 2:

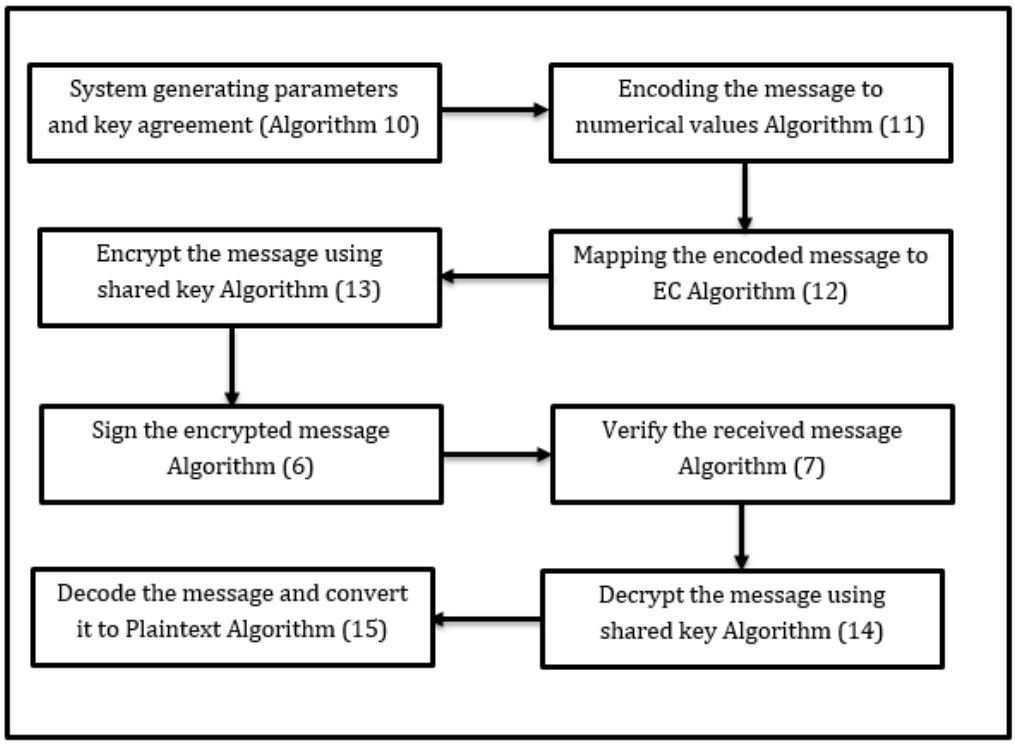


Figure 8: ECC Encryption/Decryption Model 2 Overall Structure [3]

This model passes through several steps listed as follows:

1. Initializing parameters:

   d_s : Private key of sender

   d_r : Private key of receiver

   G : Base point on elliptic curve

   PU_s: Public key for sender computed by $d_s \times G$ using scalar multiplication of point G on curve (8) with scalar d_s in The pseudo code for Scalar Multiplication represented in.

   PU_r: Public key for receiver computed by $d_r \times G$ using scalar multiplication of point G on curve (8) with scalar d_r in The pseudo code for Scalar Multiplication represented in.

   p: Large prime number

   a, b: EC coefficients

   (8): EC equation over F_p

   C_m: Cipher message (encrypted points)

   HASH: Hash function to sign the message C_m which is used to sign the message

   ksh: Shared session key (represented as a point)

   M: Number of characters in the message

   B: Number of blocks for each message

   N: Number of characters in each block

   IV: Initial vector ((size of p) – 8 bits)

   k: Random number selected between 2 and p-1, both included

   IV is said to be random variable in [3], but it isn't random.

2. Key Agreement:

   Both sender and receiver should have the same shared session key.

Algorithm 10 used for computing the shared session key for both sender and receiver which is known as ECC Diffie-Hellman Key Exchange [25]:

Algorithm 10: Key Agreement Algorithm [3]

Key Agreement Algorithm in [3]
Input: PUr, ds, PUs, dr, a, b, p // a, b, and p are the EC parameters Output: Shared key ksh 1. Parallel Execution: a. Sender: apply following scalar multiplication ds * PUr; b. Recipient: apply following scalar multiplication dr * PUs; 2. Both multiplications are equal; 3. Return ksh;

Algorithm 10 here discuss the key agreement between sender and receiver with the execution of step 1a and step 1b in Algorithm 10 can be in parallel which it's not mentioned in [3]. Further discussion will be done in the analysis of [3] in 2.3.1.

Sender side:

$ksh = d_s \times PU_r = d_s \times d_r \times G$

Receiver side:

$ksh = d_r \times PU_s = d_r \times d_s \times G$

So, both sender and receiver have the same shared session key

Below Example 17 shows key agreement process:

Example 17: Key Agreement by Algorithm 10

Let's use equation (16)

G= (5, 1)

$d_s = 4, PU_s = d_s \times G = 4 \times (5,1) = (3,1)$, using scalar multiplication in algorithm 2 on curve (16).

$d_r = 6, PU_r = d_r \times G = 6 \times (5,1) = (16,13)$, using scalar multiplication in Algorithm 2 on curve (16).

Computing the shared session key for both sender and receiver by using Algorithm 10:

Step 1 in Algorithm 10:

a. Sender: $ksh = d_s \times PU_r = 4 \times (16,13) = (9,16)$

b. Receiver: $ksh = d_r \times PU_s = 6 \times (3,1) = (9,16)$

Step 2 in Algorithm 10:

Sender's $ksh =$ Receiver's $ksh = (9,16)$

Step 3 in Algorithm 10:

Shared session key $ksh = (9,16)$.

End of Example 17.

3. Encoding the Message to Numerical Values:

   Encoding the message is a very important step, while the message should be converted from characters to integer number assigned to x coordinate to be ready for step 4.

In Algorithm 11, some phrases are corrected compared to the one in [3]. Further discussion will be done in the analysis of [3] in section 2.3.1.

In order to understand Algorithm 11 in a better way, some points should be discussed:

- $N \leq \left\lfloor \frac{size\ of\ p - 8}{8} \right\rfloor$ by (15), which is discussed in Algorithm 3 in 2.2.3.4
- $B = \left\lceil \frac{M}{N} \right\rceil$ (M is the number of characters in the original message), where this means ceiling division, which maps the result of division to the smallest integer greater than or equal to the division result.

The pseudo code for encoding the message which is represented in Algorithm 11:

Algorithm 11: Encoding Algorithm [3]

Encoding the Message to Numerical Values Algorithm in [3]
Input: The original message, p, and IV Output: Encoded message 1. Sender: obtain the M of the original message while M is the number of characters in the original message; 2. Sender: calculate N = floor((number of bits of p - 8) / 8) representing the number of characters in one block; 3. Sender: calculate B = ceiling (M / N), where B represents the number of blocks, where ceiling division is the operation of dividing two numbers and mapping the result of division to the smallest integer greater than or equal to the division result. 4. Sender: divide the message into B blocks; 5. Sender: divide each block into N characters; 6. Sender: convert each character to binary using ASCII table; 7. Sender: XOR the first block with IV; 8. Sender: for each subsequent block, XOR it with the previous XORed block; 9. Sender: pad 3 bits of zeros into each XORed block; 10. Sender: add 5 zeros to the left of the encoded message to make it 192 bits; 11. Return Encoded message;

In encryption/decryption model, the encoding process in Algorithm 3 mentioned 8 padding bits of zeros are added to the right. On the other hand, in step 9 of Algorithm 11 in encryption/decryption model 2, 3 bits of zeros are padded to the right. As mentioned in 2.2.3 (section 3), the purpose of adding the padding bits to the right is eliminate the possible incremental of x value in order to obtain y (in encryption/decryption model 2, mapping points to EC is discussed alone in Algorithm 12), which eliminated by removing the padded bits in the decoding process. Based on this info, a lot of questions must be asked now. Why 3 padded bits are added in model 2, while 8 bits is added in model 1? Are 3 padding bits enough to eliminate the change of x value later? Are 8 padding bits enough?

3 bits means 2^3 rounds to find y, meaning maximum to add 7 to the x value, and 8 bits means 2^8 rounds, meaning maximum to add 255 to the x value. Here Problem 2 is introduced, where the question is: how many padding bits needed to encode and decode the points correctly?

To clarify Algorithm 11, Example 18 is provided below:

Example 18: Encoding the message by Algorithm 11

Let's use the equation of the standard EC curve represented in Appendix D, while the parameters of it is shown in Appendix D in Figure 73.

Original Message: "Hello, my name is Bella"

IV:

"00000000000000000000000000000000000000000010111001101110011101100011100010110101010101111101111000101111001011010111100010101101000101111101110011100101011101101001110111101010100101111 0"

Step 1 in Algorithm 11: M = 23 characters.

Step 2 in Algorithm 11: $N = \{N_1 \leq \left\lfloor \frac{size\ of\ p\ -8}{8} \right\rfloor\}$ = 23 characters, because 23 characters are in M.

Step 3 in Algorithm 11: $B = \left\lceil \frac{M}{N_1} \right\rceil = \left\lceil \frac{23}{23} \right\rceil = 1$ block, so we have $B = \{B_1\}$

Step 4 in Algorithm 11: Divide the message into blocks. Since we have only 23 characters in the original message, then B_1= "Hello, my name is Bella"

Step 5 in Algorithm 11: B_1= ['H', 'e', 'l', 'l', 'o', ',', ' ', 'm', 'y', ' ', 'n', 'a', 'm', 'e', ' ', 'i', 's', ' ', 'B', 'e', 'l', 'l', 'a']

Step 6 in Algorithm 11: From ASCII table, the decimal value representation of each character will be concerted to binary and then concatenated in one block from right to left.

“01001000011001010110110001101100011011110010110000100000011011010111001001000000110111001100001011011010110010100100000011010010111001100100000010000100110010101101100011011000110000 1”

Step 7 in Algorithm 11: $B_1' = B_1$ XOR IV:

“01001000011001010110110001101100011011110010110000100000011011010111001001000000110111001100001011011010110010100100000011010010111001100100000010000100110010101101100011011000110000 1” XOR

“00000000000000000000000000000000000000001011100110111001110110001110001011010101010111110111100010111100101101011110001010110100010111110111001110010101110110100111011110101001011110” =

“01001000011001010110110001101100011011110010101100110011011010110011101111101010011000100011001111010000111010000110000101101110100101100010100111101011101111110001101111000110001111111”

Step 8 in Algorithm 11: No subsequent Blocks, so no need for this step in Example 18.

Step 9 and 10: Padding 3 bits of zero to B_1' to the right and 5 bits of zero to the left

B_1'=

“00000010010000110010101101100011011000110111110010101100110011011010110011001111101010011000100011001111010000111010000110000101101110100101100010100111101011101111110001101111000110001111111000”

Step 10 in Algorithm 11: Return encoded message which is $B = \{B_1'\}$.

End of Example 18.

4. Mapping the Encoded Message to EC:

The pseudo code for mapping points to EC which is represented in Algorithm 12:

Algorithm 12: Mapping the Encoded Messages Algorithm [3]

Mapping the Encoded Messages Algorithm in [3]
Input: Encoded block Output: Mapped points 1. Sender: obtain the decimal value for the encoded block; 2. Sender: obtain y_1 from the EC equation; 3. Sender: if x_1 cannot have corresponding y_1: 4. Sender: increment x_1 by 1; 5. Sender: repeat steps 2 – 4 until find y_1; 6. Return Mapped points;

In step 1 in Algorithm 12, the converted decimal value should be stored in x_i, while i is representing the block number.

From step 2 to step 5 in Algorithm 12, x_1 and y_1 should be replaced by x_i and y_i.

A step should be added between step 5 and step 6 in Algorithm 12, which increments i, and if $i < B$, then repeat step 1 till step 5.

Some x values have two y values, so based on that, in step 2 in Algorithm 12, any y value can be selected because it won't affect the encryption or decryption process.

Here EC coefficients a, b and p should be in the input, but they consider EC coefficients as global variables which are visible for the functions used, so this will be assume for all remaining functions in encryption/decryption model 2.

Example 19 shows mapping points to EC by Algorithm 12:

Example 19:

Let's use the equation of the secp192k1 curve represented in Appendix D, while the parameters of it is shown in Appendix D in Figure 73.

Encoded Block (the output of previous Example 18): $B = \{B_1'\}$

B_1'=

“000000100100001100101011011000110110001101111100101011001100110101101100111011111101010011000100011001110100011101000011000010110111010010110001010011110101111011111100011011110001100011111111000”

Step 1 in Algorithm 12: Convert B_1' from binary to decimal and store it in x_1:

x_1= 554734158517406194175870991365168240902251594000314004440

Step 2 in Algorithm 12: Searching for y is discussed in Appendix B using the code in Figure 72 to map a point using x.

x incremented =

554734158517406194175870991365168240902251594000314004445 (6 rounds)

y = 320284701887986563814794735854340940804574126287510471593 4.

Figure 9 shows the output of the code provided in Appendix B (Figure 72):

```
88  console.log(
89    generateXandYPoints(
90      554734158517406194175870991365168240902251594000314004440n,
91      0n,
92      3n,
93      627710173538668076383578942320766641610235544445973954104 7n
94    )
95  );
96
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS

```
{
  coordinates: {
    x: '554734158517406194175870991365168240902251594000314004445',
    y: '320284701887986563814794735854340940804574126287510471593 4'
  },
  counter: 6
}
```

Figure 9: Another Result for Mapping a Point Code

Step 3 in Algorithm 12: A value for $y_1$is found, so go to Step 6 in Algorithm 12

Step 6 in Algorithm 12:

Mapped Point:

{(554734158517406194175870991365168240902251594000314004 45,

320284701887986563814794735854340940804574126287510471 5934)}

End of Example 19.

4. Encrypt the Message using Shared Key:

   In this step, each mapped point should be added to the shared session key.

   The pseudo code for encrypting mapped points is represented in Algorithm 13:

Algorithm 13: Encrypting the Message Algorithm [3]

Encrypt the Message Using Shared Key Algorithm in [3]
Input: Mapped points, k_sh Output: Encrypted points 1. Sender: obtain k_sh; 2. Sender: add k_sh from the mapped point; 3. Sender: repeat step 2 for all mapped points; 4. Return Encrypted points;

Step 1 in Algorithm 13 shouldn't be there because ksh is already computed in Algorithm 8. Then ksh will be added to each mapped point.

Step 2 in Algorithm 13 is done by point addition using (11).

Example **20** shows the encryption of mapped points by Algorithm 13.

Example 20:

Let's consider equation (16).

Mapped point = {(3, 1)} (mapped point result in Example 19 can't be used here because I used small p for simplicity).

$ksh = (9, 16)$ (the same one obtained in Example 17).

Find the encrypted mapped points:

Step 1 in Algorithm 13: ksh is already available and $ksh = (9, 16)$.

Step 2 in Algorithm 13: Mapped point + ksh while both points are on (15) and it is done by using point addition in (8)

(3, 1) + (9, 16) = (7, 6), where similar example for point addition using (11) is done in Example 4.

Step 3 in Algorithm 13: No mapped points are remaining

Step 4 in Algorithm 13: Encrypted point = {(7, 6)}.

End of Example 20.

5. Signing and Verifying the Message:

Signing and verifying the message are discussed in 2.2.3 by Algorithm 5 and Algorithm 6, respectively.

6. Decrypt the Message using Shared Key:

The decryption of encrypted mapped points is represented in Algorithm 14:

Algorithm 14: Decrypting the Message Algorithm [3]

Decrypt the Message Using Shared Key Algorithm in [3]
Input: Encrypted points, k_sh Output: Mapped points 1. Recipient: obtain k_sh; 2. Recipient: subtract k_sh from the encrypted point; 3. Recipient: repeat step 2 for all encrypted points; 4. Return Mapped points;

Decrypting the encrypted points is done by subtracting the encrypted mapped point from ksh.

Step 1 in Algorithm 14 is already done in Algorithm 8.

Example 21 shows the decryption of mapped points by Algorithm 14:

Example 21:

Let’s use equation (16).

Encrypted Mapped Point= {(7, 6)}.

ksh = (9, 16).

Step 1 in Algorithm 14: $ksh = (9, 16)$

Step 2 in Algorithm 14: Encrypted Mapped Point - ksh by using point subtraction and equation (7).

(7, 6) – (9, 16) = (7, 6) + (9, -16) by using equation (7).

(7, 6) + (9, -16) = (7, 6) + (9, 1) because -16 mod 17 = 1.

(7, 6) + (9, 1) = (3, 1) by using equation (11) which used for point addition.

Step 3 in Algorithm 14: No more encrypted points.

Step 4 in Algorithm 14: Mapped Point = {(3, 1)}.

End of Example 21.

7. Decoding the Mapped Points to the Decrypted Message

This step converts the mapped points to plain text message.

The pseudo code for decoding the mapped points to the decrypted message is represented in Algorithm 15:

Algorithm 15: Decode the Mapped Points to Decrypted Message Algorithm [3]

Decoding the Message and Convert it to Plaintext Algorithm in [3]
Input: Mapped points Output: The decrypted plain text message 1. Recipient: obtain x_i value for the mapped point; 2. Recipient: convert x_i to the binary value; 3. Recipient: remove the padding 3 bits for each block; 4. Recipient: XOR first block with IV; 5. Recipient: for each block, XOR it with the previous block; 6. Recipient: repeat step 4 for all blocks; 7. Recipient: convert each 8 bits into its corresponding char; 8. Recipient: repeat step 8 for all blocks; 9. Recipient: for each N char aggregate to single blocks; 10. Return The decrypted plain text message;

All of these steps will be more understood while moving through Example 22.

Example 22: Decoding the Decrypted Mapped Points

IV:

"00000000000000000000000000000000000000000000000010111001101110011101100011100010110101010101111101111000101111001011010111100010101101000101111101110011100101011101101001110111101010100101111010"

Let's use the equation of the scep192k1 curve represented in Appendix D, while the parameters of it is shown in Appendix D in Figure 73.

Mapped Point:

{(5547341585174061941758709913651682409022515940003140044 5,

3202847018879865638147947358543409408045741262875104715934)} (the output of Example 18, so we can prove the work if the same original message is found)

Step 1 in Algorithm 15:

Mapped Point:

{(5547341585174061941758709913651682409022515940003140044 5,

3202847018879865638147947358543409408045741262875104715934)}

One x value will be there because the mapped point only includes one point.

x_1 = 554734158517406194175870991365168240902251594000314004 45

Step 2 in Algorithm 15:

Binary representation of x_1 =

000000100100001100101011011000110110001101111100101011001100110110100111110101001100010001100111010001110100001100001011011101001011000101001111010111011111100011011110001100011111110 1

Step 3 in Algorithm 15:

Binary representation of x_1 after removing 3 padding bits =

000000100100001100101011011000110110001101111100101011001100110110101100110111111010100110001000110011101000111010000110000101101110100101100010100111101011110111111000110111100011000111111

Step 4 in Algorithm 15: x_1 XOR IV (3 bits are added to the left of x_1 to have the same size of IV in order to XOR x_1 with IV)

“000000001001000011001010110110001101100011011111001010110011001101101100111111010100110001000110011101000111010000110000101101110100101100010100111101011110111111000110111100011000111111” XOR

“000000000000000000000000000000000000000000000000000010111001101110011101100010111001011010101010111110111100010111100101101011110001010110100010111110111001110010101110110100111011110101010010111110” =

“010010000110010101101100011011000110111100101100001000000110110101111001000011011100100000011011100110000101101101100101001000000110100101110011001000000100001001100101010110110001101100011000010”

Step 5 in Algorithm 15: Go to step 6 in Algorithm 15, no other points available

Step 6 in Algorithm 15: Go to step 7 in Algorithm 15, no other points available

Step 7 in Algorithm 15: ASSCII representation for each 8 bits starting from left:

```
[
    ("01001000", 'H'),
    ("01100101", 'e'),
    ("01101100", 'l'),
    ("01101100", 'l'),
    ("01101111", 'o'),
    ("00101100", ','),
    ("00100000", ' '),
```

("01101101", 'm'),

("01111001", 'y'),

("00100000", ' '),

("01101110", 'n'),

("01100001", 'a'),

("01101101", 'm'),

("01100101", 'e'),

("00100000", ' '),

("01101001", 'i'),

("01110011", 's'),

("00100000", ' '),

("01000010", 'B'),

("01100101", 'e'),

("01101100", 'l'),

("01101100", 'l'),

("01100001", 'a')

]

Step 8 in Algorithm 15: No more blocks, go to step 9 in Algorithm 15.

Step 9 in Algorithm 15: Decrypted message for each block= {“Hello, my name is Bella”}.

Step 10 in Algorithm 15: Decrypted message: “Hello, my name is Bella”.

End of Example 22.

The end of encryption/decryption model 2.

In this model problem 1 should be discussed. Here is Problem 1, where IV should be random in order to block chosen plaintext attack (CPA) [24], which means the attacker

will have the full ability to select some arbitrary plaintext for encryption and get the corresponding ciphertext while the attacker know the encryption algorithm and has the ability to encrypt messages from sender side using the sender's secre key but without knowing it. The attacker aim is to obtain some clue about the key. Also, the usage of random IV is to block chosen ciphertext attack (CCA) [24], which is defined as an attack for which the attacker is allowed to select a ciphertext and obtain its decryption under an unknown key. It is a much stronger attack than the chosen plaintext attack, especially in public-key cryptography, and same as CPA, the attacker has the ability to encrypt and decrypt messages in this case. Also, Deffie-Hellman key exchange is exposed for man-in-the-middle [28] (MITM) attack so trusted public keys should be used to block MITM attack, while MITM attack is a cyberattack where an attacker intercepts and potentially alters the communication between two parties without their knowledge. This attack allows the attacker to eavesdrop, steal information, or manipulate data exchanged in the communication. The solution is done using trusted public keys to block Man-in-the-Middle (MITM) attacks in Diffie-Hellman key exchange involves Certificate-Based Authentication, where each party presents a digital certificate from a trusted Certificate Authority (CA) to verify the authenticity of the public keys exchanged. This modification is done in Algorithm 21.

Proof by Equations:

Let's denote the attacker as A, sender as S, and recipient as R.

Without CA (Vulnerable to MITM):

1. The attacker will intercept and alter the public key for sender and receiver by his/her public key:

A intercepts PU_s from S and sends PU_a to R.

A intercepts PU_r from R and sends PU_a to S.

2. For key calculation, now the attacker will have the shared secret key for both sender and reciever:

S computes: k_s(sender's shared key) = $d_s \times PU_a$

R computes: k_r (receiver's shared key) = dr * PU_a

A computes two keys:

k_{sa} (sender's shared key calculated by attacker) = $d_a \times PU_s$

k_{ra} (receiver's shared key calculated by attacker) = $d_a \times PU_r$

In this case, the attacker will know both shared keys for sender and receiver.

Encryption/Decryption oracle: An encryption oracle is a theoretical black box used in cryptographic analysis, particularly in chosen plaintext attacks (CPA), that provides the encryption of any submitted plaintext, allowing adversaries to analyze corresponding ciphertexts to break the encryption scheme. Conversely, a decryption oracle, used in chosen ciphertext attacks (CCA), is a black box that decrypts any submitted ciphertext (except those submitted for encryption) and returns the plaintext, helping adversaries to exploit the decryption process and potentially reveal the secret key or construct valid ciphertexts that can expose sensitive information.

Proof of Insecurity Regarding CPA with Constant IV Using Equations

Abbreviations:

- IND-CPA: Indistinguishability under Chosen Plaintext Attack, which is a security notion for encryption schemes, where an adversary cannot distinguish between the ciphertexts of two chosen plaintexts, even after having access to the encryption oracle. It ensures that the encryption scheme remains secure under the assumption that the attacker can choose

plaintexts and see their corresponding ciphertexts but cannot gain any additional information about the plaintext from the ciphertext.

- A: Adversary
- E: Encryption scheme
- Enc: Encryption function (Algorithm 11, Algorithm 12, Algorithm 13)
- k: shared secret key (isn't known by the attacker, but is used in the encryption algorithm)
- IV: random Initialization Vector
- m_0, m_1: Messages chosen by the adversary
- c: Ciphertext

The Encryption/Decryption model 2 in [3] is not IND-CPA secure if the IV is constant. This is because the probability that the adversary can distinguish between ciphertexts of chosen plaintexts becomes significant when the IV does not change. The adversary, A, can submit as many messages as desired to the encryption oracle. For each query, A receives the corresponding ciphertext. If the IV is constant, the encryption process becomes predictable.

The adversary submits q queries to the challenger to encrypt plaintexts of their choice. The adversary then sends two messages, m_0 and m_1, to the challenger for encryption. The challenger encrypts one of the messages using a randomly chosen bit b, resulting in c_b =Enc(k, $m_b \oplus$ IV). The adversary's task is to guess the value of b.

Consider the following games:

1. Game 0: The adversary requests encryption of a message m_0 and receives c_0 = Enc(k, $m_0 \oplus$ IV).

2. Game 1: The adversary sends two messages m_0 and m_1, where m_0 is the same as in Game 0. The challenger encrypts one of them, m_b, and sends c_b =Enc(k, $m_b \oplus$ IV) to the adversary.

3. Game 3: The adversary attempts to distinguish between m_0 and m_1 based on the received ciphertexts. The probability of A guessing b is high because the ciphertexts in both games are generated with the same IV. So both ciphertexts will be the same if b = 0, and if it is different then b = 1, which means guessing between bits 0 or 1 is easy for the attacker.

This results in the messages m_0 and m_1 being distinguishable. The ciphertext from m_0 in Game 0 is the same as the ciphertext from m_0 in Game 1 because the same IV is used in each game. Hence, the adversary's advantage Adv_IND-CPA[A, E] is non-zero, proving the scheme's insecurity under CPA when the IV is constant.

Proof of Insecurity Regarding CCA with Constant IV Using Equations

Abbreviations:

- IND-CCA: Indistinguishability under Chosen Ciphertext Attack, which is a stronger security notion where an adversary cannot distinguish between the ciphertexts of two chosen plaintexts, even when the attacker has access to a decryption oracle, except for the challenge ciphertext. It ensures that even if an attacker can decrypt arbitrary ciphertexts (other than the challenge ciphertext), they still cannot gain any information about the plaintext from the challenge ciphertext.
- Dec: Decryption function (Algorithm 14, 15 and 16)

Proof:

The proof for Adv_IND-CCA[A, E] starts by noting the adversary's ability to access both the encryption and decryption oracle. The adversary, A, can submit chosen ciphertexts to the decryption oracle and chosen plaintexts to the encryption oracle. The encryption process is defined as:

c = Enc(k, m, IV)

The adversary submits two distinct messages, m_0 and m_1, to the encryption oracle and receives the corresponding ciphertexts c_0 and c_1:

c_0 =Enc(k, $m_0 \oplus IV$)

c_1 =Enc(k, $m_1 \oplus IV$)

In the challenge phase, the adversary chooses m_0 and m_1 again, and the challenger selects a random bit b. The challenger then encrypts m_b and returns the ciphertext c_b:

c_b = Enc(k, m_b, IV)

To distinguish between m_0 and m_1, the adversary modifies the ciphertext c by XORing it with a constant IV and submits it to the decryption oracle:

$c' = c \oplus IV_i$The decryption oracle decrypts c' to obtain:

Dec(k, c') = Dec(k, c $\oplus$ IV_i)

Since the decryption process relies on IV:

Dec(k, Enc(k, $m_b \oplus IV$) $\oplus$ IV)

The adversary can then compare this result with the known plaintexts m_0 and m_1 to determine b because the ciphertext is always the same and unique ciphertext for every plaintext, so this will lead to compute the shared key as well. So guessing between the bits is easy for the attacker, as a result, it will lead the attacker to guess both the ciphertext and the plaintext.

This demonstrates that Encryption/decryption model 2 in [3] isn't secure under CCA when a constant IV is used. Therefore, using constant IV is essential to maintaining the security of the encryption scheme under chosen ciphertext attacks.

2.3 Secure and Efficient ECC Systems

In this section, we explore various secure and efficient systems implemented using Elliptic Curve Cryptography (ECC). These systems are evaluated based on their features, security mechanisms, and efficiency in handling cryptographic tasks. The primary focus is on SE-Enc [3] in 2.3.1 which focuses on a secure and efficient ECC for IoT. Also, some brief discussion will be done for Singh [6], and Sengupta [4] systems in section 2.3.2. Moreover, section 2.2.3 shows a brief discussion for Barman system [27]. Also section 2.3.4, analysis of the known experimental results on security, mapping EC points successfully and encoding/decoding operations will be done. Finally, section 2.3.5 shows a summary which leads to choose SE-Enc [3] as a main reference for this thesis.

2.3.1 SE-Enc System Description [3]

The system called SE-Enc, which has been developed by Hisham N. Almajed and Ahmad S. Almogren [3], employs an advanced Elliptic Curve Cryptography (ECC) that is specifically designed for resources-constrained high-security environments. This system is represented in Figure 10.

This system takes the original message as an input and sends it encrypted to the receiver, then the output of the decryption process should be the decrypted message (same as the original message taken in the input). The following diagram, represented in Figure 10, provides a detailed visual representation of the SE-Enc [3] encoding and

encryption process, illustrating each step from message encoding to mapping onto the elliptic curve and the secure encryption and decryption processes:

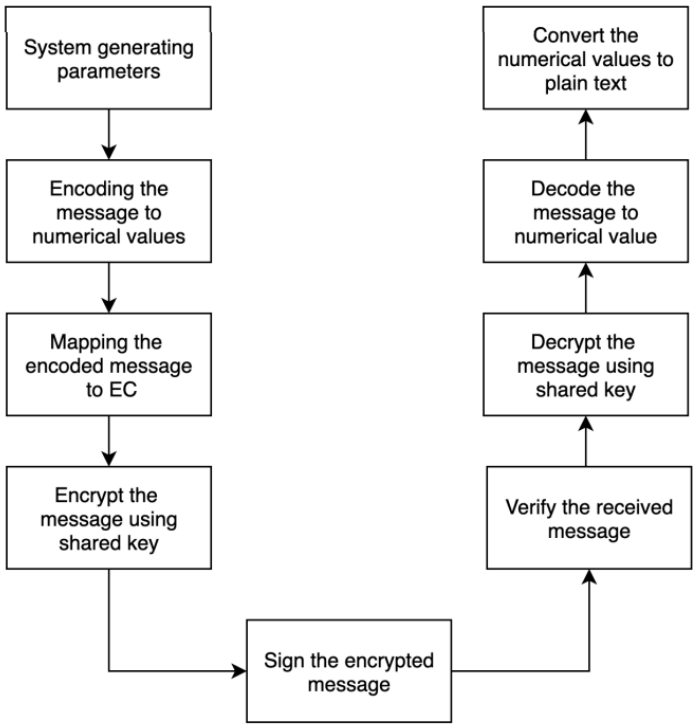


Figure 10: SE-Enc Encryption and Decryption Process [3]

The parts of Figure 10 will be discussed in the detailed algorithms used in SE-Enc.

Detailed Algorithms Used in SE-Enc:

The following list shows the algorithms used in SE-Enc and referred to which part in Figure 10:

Algorithm 10: Key Agreement Algorithm, refers to system generating parameter's part, while this is used to agree for the same shared key between the 2 parties (sender and receiver).

Algorithm 11: Message Encoding Algorithm, refers to encode plaintext to numerical values part

Algorithm 12: Block Map for EC Algorithm, refers to mapping encoded message to EC part

Algorithm 13: Encrypting Mapped Points Using Shared Key (ksh) Algorithm, refers to encrypt the message using shared key part

Algorithm 5: Signing the Message Algorithm, refers to sign the encrypted message part.

Algorithm 6 Verifying the Message Algorithm, refers to verify the received message part.

Algorithm 14: Decrypted points Algorithm, refers to decrypt the message using shared key part.

Algorithm 16: Decoding Mapped Points to Binary Values Algorithm, refers to decode the message to numerical value part.

Algorithm 17: Decoding Binary Values to Text Algorithm, refers to convert the numerical values to plain text part.

1. Initialization of System Parameters:

System parameters for SE-Enc [3] are represented in Figure 11, however H which represents the hash function is notated in a wrong way because other notation is used in the algorithms for [3]. Hence, the correct notation is HASH. Also, the initial vector isn't a random number. In addition, k is a random number as mentioned in Figure 11, but it has to be between 2 and p-1, both included, because it is used for scalar multiplication in Algorithm 2, so it isn't reasonable to accept 1 as a value for k. These adjustments are done in step 1 for encryption/decryption model 2 in section 2.2.5.

ECC parameters a, b, and p, in addition of base point G are considered as global variables to be used in all SE-Enc system algorithms even if they don't appear in the input of the algorithms.

Figure 11 shows the system parameters' notations and their description:

Notation	Description of notations
d_s	Sender private key
d_r	Recipient private key
G	Base point on elliptic curve
PU_s	Sender public key = $d_s * G$
PU_r	Recipient public key = $d_r * G$
p	Large prime number (192-bit)
a, b	EC coefficients, s.t. $4a^3 + 27b^2 \ mod \ p \neq 0$
$y^2 \equiv x^3 + ax + b \ mod \ p$	EC equation to map points to EC
H	Hash function to sign the message C_M
k_{sh}	Shared session key
M	Total number of characters in the message
B	Number of blocks for each message
N	Number of characters on each block
IV	Random initial vector (192-bit)
k	Randomly securely selected from $[1, p-1]$
C_M	The encrypted message (all encrypted points)

Figure 11: List of Notations Used to Generate Scheme Parameters [3]

In this step, key agreement algorithm (Algorithm 10) will be done between sender and receiver. Algorithm 10 helps to establish a shared key between the sender and receiver, which is essential for secure encryption of communication.

Step 1 and step 2 in Algorithm 10 can be executed in parallel, while it is corrected from the one in [3] in the encryption/decryption model 2 in section 2.2.5 (Algorithm 10). In the encryption/decryption model 2, all details related to key agreement are explained there.

2. Message Encoding refers to Algorithm 11 in 2.2.5:

Algorithm 11 transforms the plaintext message to the format suitable for ECC processing, which includes padding, converting to binary and XORing. Figure 12 is illustrating message encoding Algorithm 11:

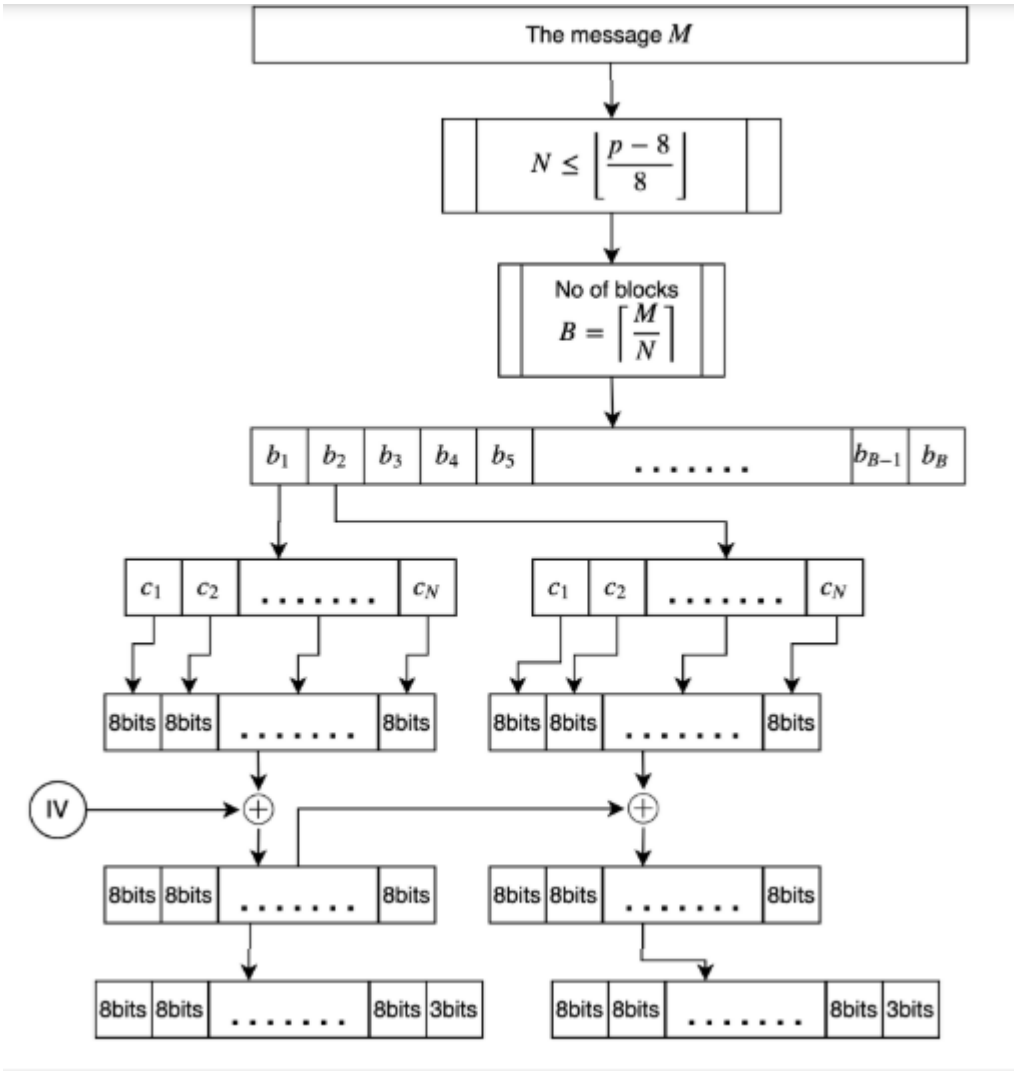


Figure 12: Message Encoding Diagram for Algorithm 11 [3]

Algorithm 11 corrects a mistake in step 2 in [3], while N is the number of characters in each block. Also, [3] misses a lot of explanations of Algorithm 11 as defining M (number of characters in original message), defining the equation used for N, and the equation used for B. Algorithm 11 in encryption/decryption model 2 includes more explanations for message encoding in [3].

3. Block Map for EC (Algorithm 12 in 2.2.5):

Mapping points to EC is used to map the encoded blocks of the message to the points which lie on the elliptic curve.

This step is very important to obtain points on EC. More details are explained in encryption/decryption model 2, in step 4.

4. Encrypting Mapped Points Using Shared Key (ksh) (Algorithm 13 in 2.2.5)

The purpose of Algorithm 13 in section 2.2.5 is to encrypt the mapped points by adding them with the Shared key ksh by using point addition equation (11). Each point is added to the shared key and that's how the encrypted coordinates are generated.

Algorithm 13 represents step 5 in the encryption/decryption model 2, where it is explained in details in the model.

5. Signing the Encrypted Points and Message Validation by the Recipient

A digital signature is a cryptographic technique, which allows one party to prove the authenticity and integrity of a message in a digital way. Digital signatures are used to ensure authenticity of the sender and integrity of the message during transit. In addition, they take away from the sender the ability to reject the delivery of the message (non-repudiation).

Encrypted points signature b sender is represented in Algorithm 5 in Section 2.2.3 in step 6. Verifying the message by the receiver is represented in Algorithm 6 in Section 2.2.3 in step 6. ECDSA is used in both Algorithm 5 and Algorithm 6 where more explanation is done in step 6 in Section 2.2.3.4.

6. Decrypted points Algorithm 14 in Section 2.2.5

Each encrypted point is subtracted from the shared group key by using equation (7). This actually decrypts the points, putting them back to their original mapped positions on the curve.

More explanation is done for Algorithm 14 in step 7 in Section 2.2.5.

7. Decoding Deciphered Points to Binary Values Algorithm 16

The pseudo code for decoding deciphered points to binary values is represented in Algorithm 16:

Algorithm 16: Decoding Mapped Points to Binary Values Algorithm [3]

Decoding Deciphered Points to Binary Values Algorithm in [3]
Input: Mapped points Output: Encoded block 1. Recipient: obtain x_i value for the mapped point; 2. Recipient: convert x_i to the binary value; 3. Recipient: remove the padding 3 bits for each block; 4. Recipient: XOR first block with IV; 5. Recipient: for each block, XOR it with previous block; 6. Recipient: repeat step 4 for all blocks; 7. Return Binary values;

Algorithm 16 is a part of The pseudo code for decoding the mapped points to the decrypted message is represented in Algorithm 15: 15 in section 2.2.5. It is all explained in step 8 in section 2.2.5.

Algorithm 16 transfers the mapped points, by using the x value of each point, and does the steps mentioned in the algorithm, to reach at the end the encoded blocks which contains binary numbers.

8. Decoding Binary Values to Text Algorithm 17

The pseudo code of decoding binary values to plaintext is represented in Algorithm 17:

Algorithm 17: Binary to Characters by Using ASCII Table Algorithm [3]

Decoding Binary Values to Text Algorithm in [3]
Input: Binary values Output: The plain text message M 1. Recipient: obtain the binary values; 2. Recipient: convert each 8 bits into its corresponding char; 3. Recipient: repeat step 3 for all blocks; 4. Recipient: for each N char aggregate to single blocks; 5. Return The message M;

Algorithm 17 is a part of Algorithm 15 in section 2.2.5. It is all explained in step 8 in section 2.2.5.

Conclusion:

SE-Enc is considered to be a good ECC scheme for security and efficiency. However, some deficiencies appeared by addressing some problems concerning this scheme. Starting with Problem 1 (in Section 2.2.5 in step 1) which is the random IV, which is introduced to be used but it is constant IV, while this problem is solved in an updated version of SE-Enc [5], but without a clear explanation on the procedure for creating this random IV. Also several questions are asked about the 3 padded bits which end up with 2 problems, which are specifying whether 3 padding bits are enough for successfully mapping for all EC points (Problem 2 in Section 2.2.5), and if these 3 padding bits have better performance (less compilation time) than 8 bits padding bits (Problem 3 in this section).

2.3.2 Singh [6] and Sengupta [4] Systems

Here Singh [6] and Sengupta [4] systems will be discussed, which are based on Elliptic Curve Cryptography (ECC) and offer improved security and performance. Even though these systems are intended for general purpose applications, we can adopt some

of their techniques to the IoT area in order to obtain an efficient and secure ECC implementations for IoT.

Singh System [6]:

The Singh system proposes an ECC scheme which is useful for the communication of images. The main aim of the Singh system is to encrypt and decrypt images using elliptic curve cryptography. The main objective is to map the points to elliptic curve formatted as (8) in a simple way. This is done by adding group of pixels representing x value and then compute y value which represented on elliptic curve. This approach is better so mapping process isn't repeated for every pixel.

Figure 13 shows the overall structure of Singh System:

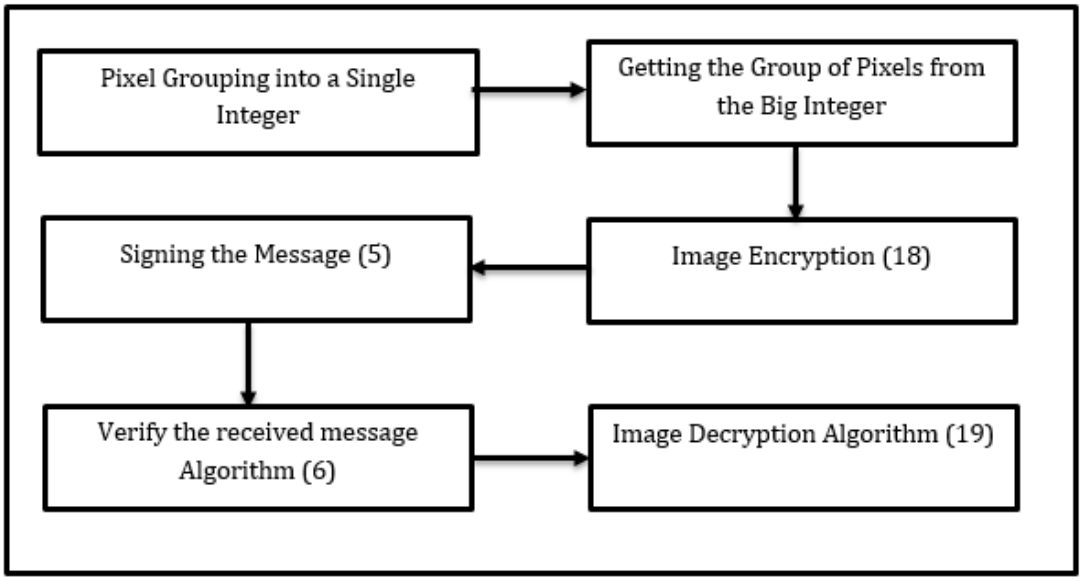


Figure 13: Overall Structure for Singh System [6]

Detailed Steps Used in Singh:

1. Pixel Grouping into a single integer

Images are represented by pixels. Normally, to encrypt image, each pixel should be encrypted. This step is working on grouping pixels in a single integer so every group of pixels will be encrypted, instead of encrypting each pixel alone. The number of the

grouped pixels is depended on the prime field of the ECC. For instance, 512 bits ECC prime field groups up to 63 pixels together. To find the single integer which represents the group of pixels, a function from Mathematica is used called FromDigits [list of pixels, b], where list of pixels represents the pixels which will be grouped and b represents the base for pixels which is 256 (max range + 1). 1 or 2 is added to each pixel to avoid the error done by FromDigits function. The value for each pixel varies between 0 and 255 (both included), because of the addition of 1 or 2 to each byte, then the range may reach 257. Based on that, the base used in FromDigits is 258.

2. Getting the group of pixels from the big integer

This step will convert back the single big integer to list of pixels in range 0 and 255, both included. To achieve this goal, IntegerDigits function form Mathematica is used, while IntegerrDigits[big integer value, 256] provides a list of pixels ranges between 0 and 255 (base -1), both included. IntegerDigits is the inverse fnction for FormDigits.

3. Algorithm 18: Image Encryption

The pseudo code of image encryption is represented in Algorithm 18:

Algorithm 18: Image Encryption Algorithm [6]

Image Encryption Algorithm in [6]
Input: Image to be encrypted Output: Cipher image 1. Get the pixel value of the image to be encrypted and randomly add 1 or 2 to each pixel. Record the number of channels present in the image. 2. Group the pixels and convert to single large integer value for each group. Number of pixel to be group using Mathematica is given by grp = Length[IntegerDigits[p, 258]] - 1 3. Pair up the result obtained from step 2 and store as 'Pm' which is the plain message input for the ECC system. 4. Select a random 'k' and compute 'kG' and 'kPb' where 'Pb' is the public key of the receiver. 5. Perform point addition of 'kPb' with each value of 'Pm' and store as 'Pc' which is the cipher text. 6. Convert the cipher text list from step 5 to value ranging from 0 to 255.

7. Pad left with 0 to each list from step 6 which have less than grp + 1 number of elements, to make each list equal in length.
8. Flatten the list from step 7, group them according to the number of image channels that we have recorded and partition them to the width of the plain image.
9. Convert the values from step 8 into cipher image.

Return cipher_image

The steps in Algorithm 18 explains the encryption of the image used in Singh system, while a step is missed between step 2 and step 3, which is mapping each integer to its point on elliptic curve.

Step 4 and step 5 in Algorithm 18 are described in section 2.2.4 in Algorithm 7.

4. Digital Signature on Cipher Image

Signing the message is related to ECDSA discussed in section 2.2.3 (Algorithm 5).

5. Verifying the Signature

Verifying the signature is related to ECDSA discussed in section 2.2.3 in Algorithm 6.

6. Algorithm 19: Image Decryption:

The pseudo code of image encryption is represented in Algorithm 19:

Algorithm 19: Image Decryption Algorithm [6]

Image Decryption Algorithm in [6]
Input: Cipher image to be decrypted Output: Plain image 1. Get the pixel value of the cipher image and group by grp + 1 number of pixels and form single big integer value for each group with base 256. Record the number of image channels of the cipher image. 2. Pair up the value obtained from step 1. 3. Perform point multiplication of 'kG' with 'nB' where 'nB' is the private key of the receiver. 4. Perform point subtraction between values from step 2 with value from step 3. 5. Get the value in the range of 0 to 255 from step 4 with base 258 and subtract random 2 from each value. 6. Group the flatten value obtained in step 5 in terms of recorded number of image channels of the cipher image and partition them to the width of the cipher image. 7. Convert the values from step 6 into plain image.

Return plain_image

The steps in Algorithm 19 explain the image decryption which is done in Singh system.

Conclusion for Singh System:

In order to guarantee the integrity and validity of the received image, the Singh system presents the implementation technique for image encryption and decryption in addition to the insertion of a digital signature to the encrypted image. The process entails putting pixels in groups and describing how many groups of pixels are possible given the ECC parameters. They paired these grouped pixel values rather of translating them to Elliptic curve coordinates, which obviates the requirement for a reference mapping table both during encryption and decryption. The approach generates a cipher image with minimal correlation even in cases where the source image has uniform pixel values. They examine the methodology as well to highlight the algorithm's power.

Sengupta System [4]:

ECC is utilized as a public-key cryptosystem for encryption and decryption. To encrypt a message, it is mapped to a distinct point on the elliptic curve using a mapping algorithm. While ECC arithmetic is computationally simpler than that of other cryptographic algorithms, converting simple messages to elliptic curve points remains a challenge. In Sengupta scheme, they examine various message mapping schemes in ECC, highlighting the flaws and vulnerabilities of each to cryptanalysis. They also outline the characteristics of an effective message mapping scheme in ECC. In the latter part of [4], they introduce a new mapping scheme that resists frequency analysis and other cryptanalytic attacks.

Figure 14 shows the steps for Sengupta system:

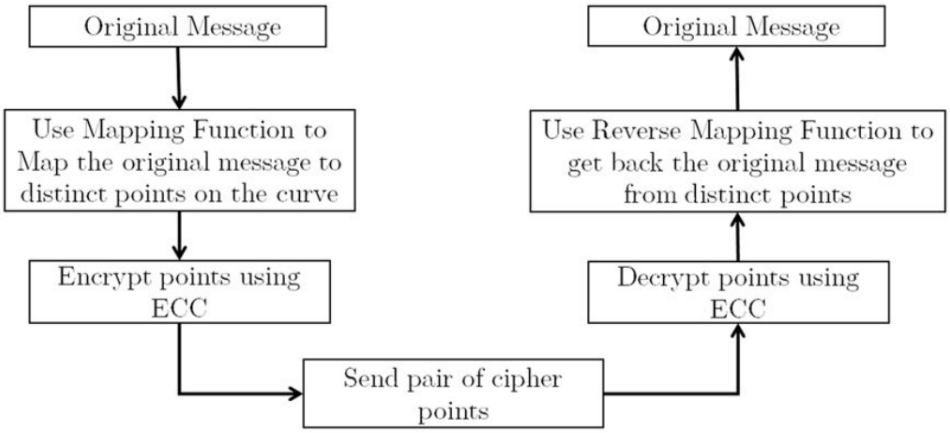


Figure 14: Steps for Sengupta System [4]

Steps for Sengupta scheme:

1. Original Message:

Original message is obtained by the sender and it should be represented as a text containing character/s.

2. Mapping Points:

This step is discussed in section 2.2.3 in Algorithm 3.

3. Encrypt Points using ECC:

This step is done exactly like the introduced section 2.2.4 in Algorithm 8.

4. Send Pair of cipher points:

This step discussed also in section 2.2.4 in Algorithm 8.

5. Decrypt Points using ECC:

This step is discussed in section 2.2.4 in Algorithm 9.

6. Reversed Mapping:

This step is discussed in section 2.2.3 in Algorithm 4.

7. Original Message:

The decrypted message in step 6 should be equal to the original message in step 1.

Conclusion:

In [4], a new message mapping scheme for elliptic curve cryptosystems is proposed, accompanied by derived guidelines for an effective mapping scheme. The proposed scheme adheres to these guidelines, demonstrating its quality as a mapping scheme. Security analysis reveals that the new scheme prevents many attacks that previous schemes discussed in [4] were vulnerable to. Additionally, the proposed scheme is faster than those previously discussed schemes in [4]. Its main advantage is that no prior information needs to be shared for the mapping or reverse mapping process.

2.3.3 Barman System [27] Description

Introduction:

The paper addresses security challenges in IoT, proposing a security framework combining ECC with DNA encoding. ECC is known for its efficiency and security, suitable for resource-constrained IoT devices. The paper details how DNA encoding can enhance ECC encryption to provide robust security for IoT environments.

Security Challenges within IoT Systems:

IoT systems face several security challenges due to their diverse applications and limited resources. These challenges include:

- Physical security of devices.
- Computational limitations preventing complex security algorithms.
- The need for secure remote management.
- Scalability issues with identifying and managing billions of devices.
- Ensuring the longevity of cryptographic algorithms beyond the lifespan of IoT devices.

Solution for IoT Security:

The paper discusses the necessity of encryption for IoT security, highlighting common algorithms such as RSA and ECC. It emphasizes the need for security at multiple layers and introduces the concept of DNA encoding to strengthen cryptographic methods.

The steps for Barman system are shown in Algorithm 20:

Algorithm 20: Plaintext Conversion to Encrypted DNA-Based ECC Ciphertext Algorithm in [27]

Plaintext Conversion to Encrypted DNA-Based ECC Ciphertext Algorithm in [27]	
Inputs: - P: Plaintext message - D: Known DNA sequence - k: Segment size for binary DNA sequence (k > 2) - ECC_params: (a, b, p) // Elliptic curve parameters - G: Base point on the elliptic curve - P_B: Public key of the receiver - n_B: Private key for receiver Steps: 1. Plaintext Conversion to Binary: a. Start with the plaintext message P. b. Convert the plaintext P into its binary representation P'. 2. DNA Sequence Conversion: a. Select a known DNA sequence D. b. Convert the DNA sequence D into its binary equivalent D' using a predefined conversion table: - Adenine (A) = 00 - Thymine (T) = 01 - Guanine (G) = 10 - Cytosine (C) = 11	4. Reconstruction and DNA Conversion: a. Concatenate the modified segments to form a new binary sequence. b. Convert this binary sequence back into a DNA sequence DN using the same binary-to-nucleotide table. 5. Decimal Conversion: a. Convert the newly formed DNA sequence DN into a decimal number N using a nucleotide-to-number conversion table: - Adenine (A) = 10 - Thymine (T) = 20 - Guanine (G) = 30 - Cytosine (C) = 40 6. Elliptic Curve Mapping: a. Map the decimal number N to a point P_m on the elliptic curve using ECC_params. 7. Elliptic Curve Encryption: a. Randomly select a private key k.

3. Binary Segmentation and Insertion:

a. Segment the binary DNA sequence D' into smaller chunks, where each segment contains k bits.

b. Insert each bit of the binary plaintext P' into the beginning of each segment of D'.

4. Reconstruction and DNA Conversion:

a. Concatenate the modified segments to form a new binary sequence.

b. Convert this binary sequence back into a DNA sequence DN using the same binary-to-nucleotide table.

b. Compute the encryption components:

i. k×G // Multiply k with the base point G

ii. P_m + k×P_B // Add the point P_m to the product of k and the receiver's public key P_B

8. Transmission:

a. Transmit the ciphertext {k×G, P_m + k×P_B} to the receiver.

Notes:

- Steps 6–8 correspond to the operations mentioned in Algorithm 3 and Algorithm 8, including more detailed explanations.

End Algorithm

For Decryption: only this equation is mentioned: $P_m + kP_B . - n_B\,(kG) = Pm + k\,(n_B)\,G - n_B\,(kG) = Pm$, where n_B is the private key of the receiver, but in [27] the decryption isn't explained. And the equation is similar to the one in Algorithm 9.

Based on that, the overall structure for the Barman System is represented in Figure 15:

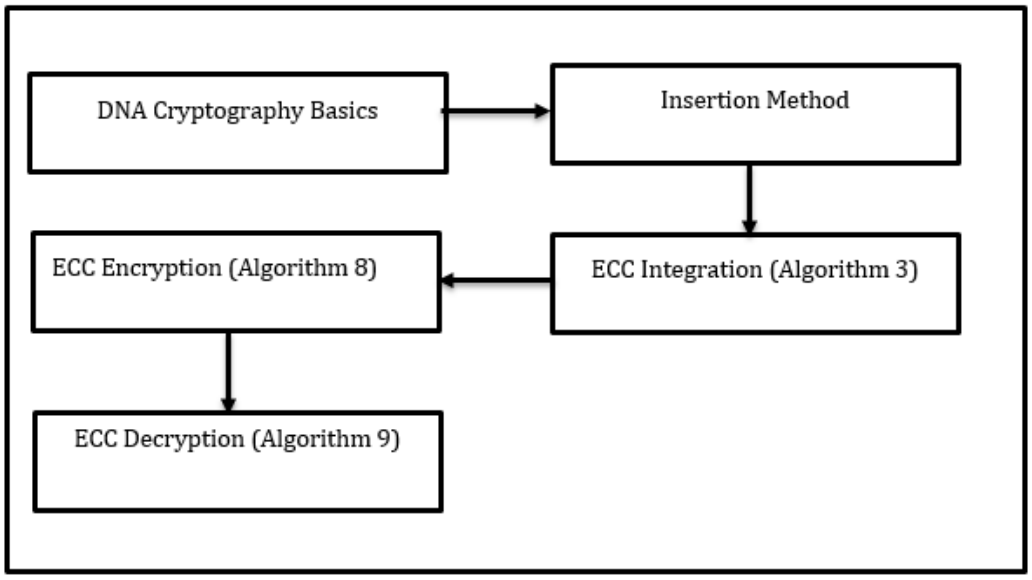


Figure 15: Barman System Overall Structure

DNA Encoded ECC for IoT Security:

DNA cryptography uses the sequences of DNA bases (A, T, G, C) as information carriers. This method involves converting plaintext into DNA sequences and using these sequences for encryption. The combination of DNA cryptography with ECC leverages the strengths of both methods to create a highly secure system.

Encoding Mechanism:

1. DNA Cryptography Basics: DNA cryptography utilizes the biological properties of DNA sequences. The four nucleotides (A, T, G, C) are encoded into binary values (A=00, T=01, G=10, C=11).

2. Insertion Method: The plaintext is first converted into binary. This binary sequence is inserted into the beginning of segments of a binary DNA sequence. The segments are then concatenated and converted back to nucleotide sequences.

3. ECC Integration: The resulting DNA sequences are converted into decimal values, which are then mapped to points on an elliptic curve using Algorithm 3. These points represent the plaintext points (P_m).

4. Encryption (using Algorithm 8): Using ECC, the plaintext points are encrypted into ciphertext points using the ECC encryption formula: $\{k \times G, P_m + k \times P_B\}$, where k is a random number, G is the base point, and P_B is the public key of the receiver.

5. Decryption (using Algorithm 9): The receiver deciphers the ciphertext points using ECC decryption, converting them back into numbers, and then decoding the DNA sequence to retrieve the original plaintext.

Conclusion:

The proposed DNA encoded ECC system provides a dual-layer security mechanism for IoT devices. It combines the efficiency of ECC with the complexity of DNA cryptography, making it secure against modern threats but having some deficiencies

while digital signature isn't used and it's exposed to CCA and CPA. The paper suggests further development and practical implementation to validate the proposed scheme.

2.3.4 Experimental Settings and Results on ECC System Security and Efficiency [3] - [4] - [5] (Results are Taken from [3], [4] and [5])

Problems Statements and Goals of [3], [4], and [5]

The listed and known experimental studies have been done to solve several problems towards secure and efficient implementation of ECC. The main problems statements and goals of the experimental studies are given below:

1. Experiment 1: Analyze the encryption operation of ECC scheme while encrypting the message twice without using random IV, and twice with using random IV, which proves that the generated ciphertext is secured against possible cryptographic attacks, like CCA and CPA.
2. Experiment 2: Evaluate the number of rounds needed for mapping number of numeric values to points on an elliptic curve, and include the successful rate of mapping the points in each round. This experiment is done on three standard elliptic curves which are secp192k1, secp224k1 and secp256k1 mentioned in Appendices D.1, D.2 and D.3 respectively.
3. Experiment 3: Evaluate the performance of the encryption of Sengupta system [4], which computes the time needed for the encryption process, while its performance is compared to another scheme discussed in [4].

Results of three known experiments from [4], [3], and [5] are analyzed in the next parts 1-3:

1. Experimental Settings and Results of Experiment 1 [5]:

Objective:

The objective of Experiment 1 is to involve encrypting (Algorithm 17 (updated in [5] to have random IV), Algorithm 12 and Algorithm 13) the same plaintext multiple times to observe the effects of using a random initial vector (IV) on the encryption output, where the output without using random IV for the same plaintext and EC parameters will be the same (the same encrypted message is always obtained when the same plain text is encrypted), while by using random IV, if the same plaintext is encrypted, then the encrypted message will be always different. The experiment is conducted in two parts: first, the same plaintext is encrypted twice without using a random initial vector, and this process is repeated twice to confirm consistency. Second, the same plaintext is encrypted twice using a random initial vector each time, and this process is also repeated twice to observe the variations introduced by the random IV. By comparing the outputs from both parts, we aim to ensure the security of the scheme (ciphertext) against Chosen Ciphertext Attacks (CCA) and Chosen Plaintext Attacks (CPA).

Experimental Setup:

Plaintext Input: The same sample of plaintext (only one sample plain text) is used in all runs in Experiment 1 for ensuring the consistency of input data and to examine the impact of encryption process on the same input by varying the parameters. The used plaintext is:

"This is a test code of Java application and encryption system. It's not an actual system in real life."

All the runs of the encryption process in Experiment 1 have the same ECC parameters (a, b and p) and the same base point G (while there exact values aren’t mentioned in [5], but probably the standard EC secp192k1 (in Appendix D) is used), while the first

2 runs doesn’t have IV, and the second 2 runs have random IV (different IV for each one because it is random).

Encryption Process: The plaintext is encrypted to ciphertext by using the ECC encryption in model. The encryption process may vary depending on the random initial vector (IV) used in Figure 18 and Figure 19.

Experiment 1 Results:

Figure 16 and Figure 17 show the encrypted points of the same plaintext without using a random initial vector which are computed twice, the first encryption run is done in Figure 16 while the second run is done in Figure 17, while both give the same encrypted points as outputs, which is as expected because random IV isn’t used.

Figure 18 and Figure 19 show the encrypted points of the same plaintext with using a random initial vector which are computed twice, the first encryption run is done in Figure 18 while the second run is done in Figure 19, while give different encrypted points as outputs, as expected and which is good so the same plaintext won’t have a unique output in this case.

Plaintext:

This is a test code of Java application and encryption system. It's not an actual system in real life

Encrypted points:

(20607074791017864591888755486294908591020435794799437684,29761832042692535779900008144331930740840315249220681118 94)

(21284171406412493778662006757228316730543974115944705937 18,82187983534761801011596735963263787141046796684158000 17)

(19521567662103091062578942696050301849395331296046968471 73,27858192322192073919523154467101088660986065935744125827 76)

(57912696496056807184280636422476693597425655654600760594 89,17061881673937273068888375976806260523482198802398547977 62)

(46202876753226439252107421672589800712848262828950808734 64,24420361806420178238978279226276650980049450903956540863 30)

Figure 16: Cipher Text Generated by the First Run of Encryption Function [5]

Figure 16 shows the encrypted points of the plaintext used (mentioned in Figure 16) and this plaintext is represented in 5 EC points, while random IV isn't used which means if running the encryption again for the same plaintext, it has to give the same encrypted points.

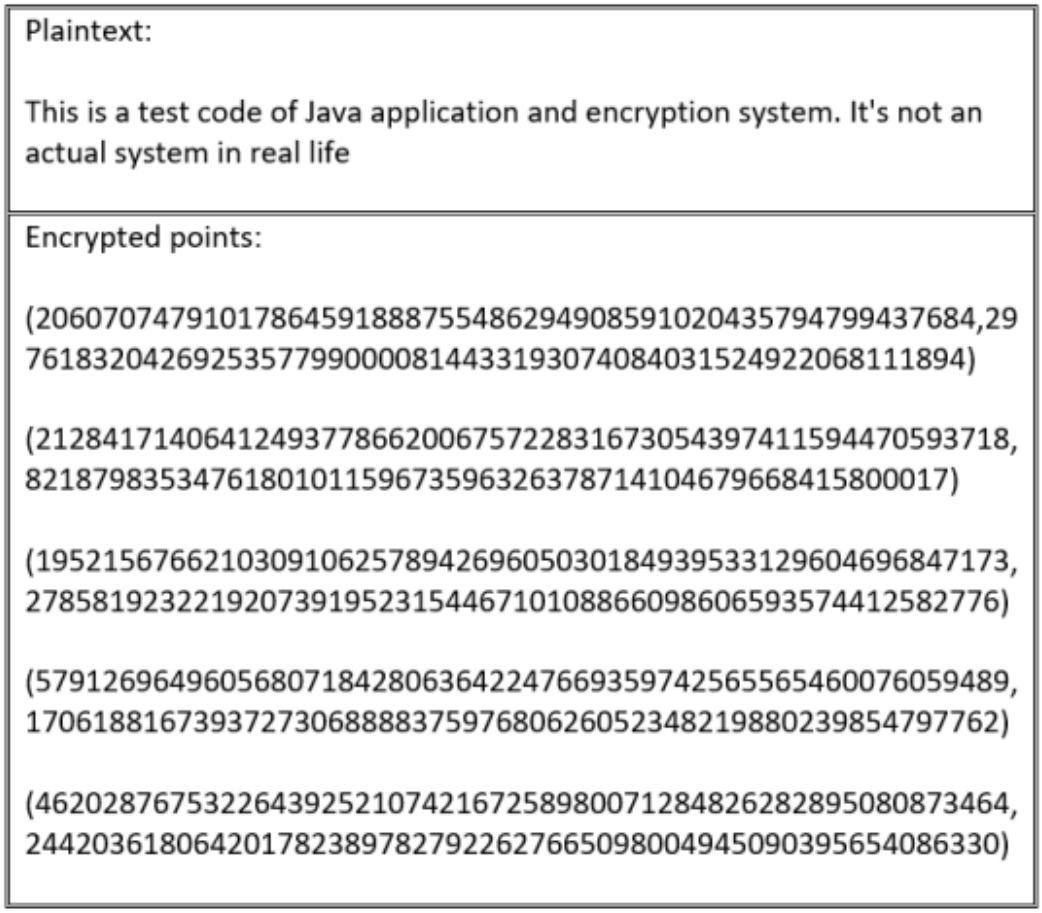


Figure 17: Cipher Text Generated by Second Run of the Encryption Function Using the Same Plaintext as in Figure 17 [5]

Figure 17 shows the encrypted points of the plaintext used (mentioned in Figure 17 and it's the same plaintext in Figure 16) and this plaintext is represented in 5 EC points, while random IV isn't used and the encrypted points are exactly the same as the ones in Figure 16. This is expected and proves that without using random IV, the encryption of the same plaintext will lead to the same encrypted points.

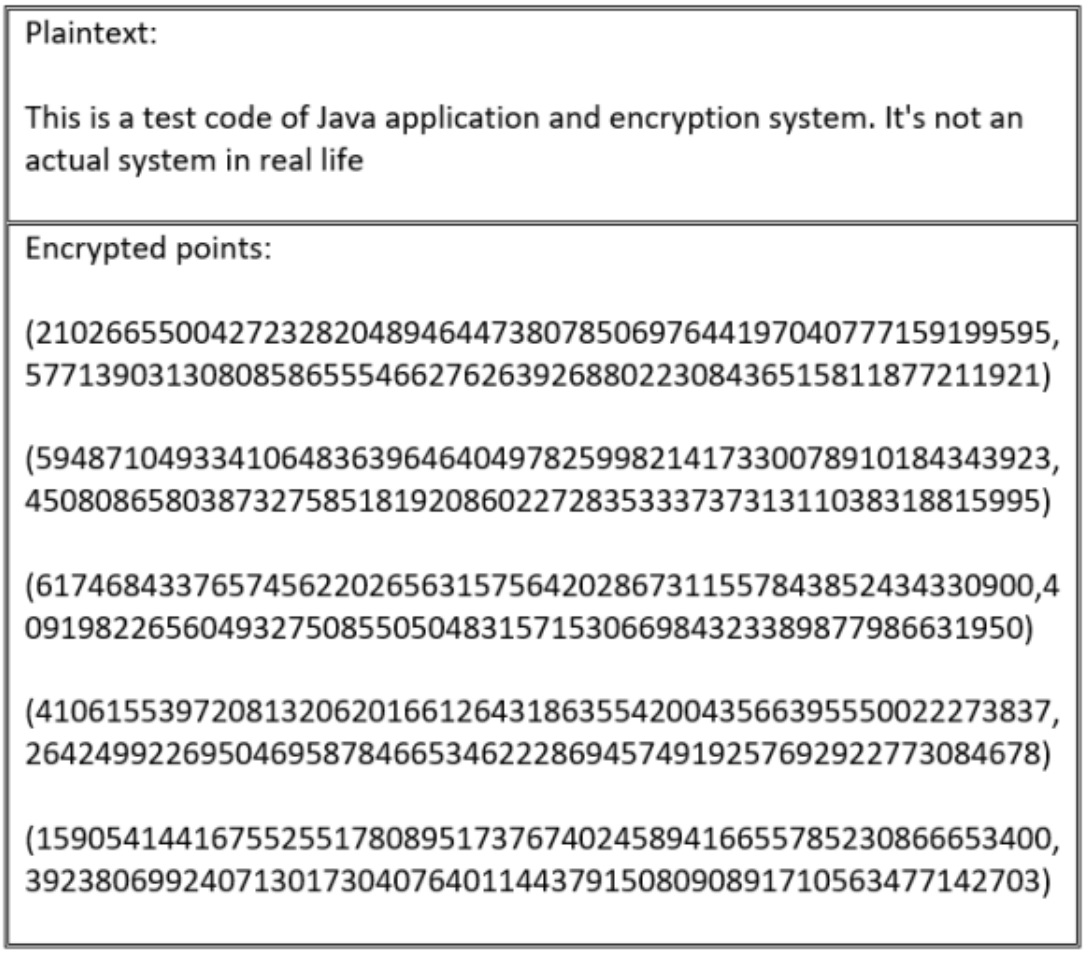


Figure 18: Cipher Text Generated by the First Encryption Function Using XOR with random IV [5]

Figure 18 shows the encrypted points of the plaintext used (mentioned in Figure 18) and this plaintext is represented in 5 EC points, while random IV is used and the encrypted points are different than the ones in Figure 16 and Figure 17. This is expected because of the using of random IV, but also it’s not enough because a new variable is used this time so it’s normal to get different encrypted points compared to the ones in Figure 16 and Figure 17, so the expectation to have different encrypted points while running the encryption again using random IV.

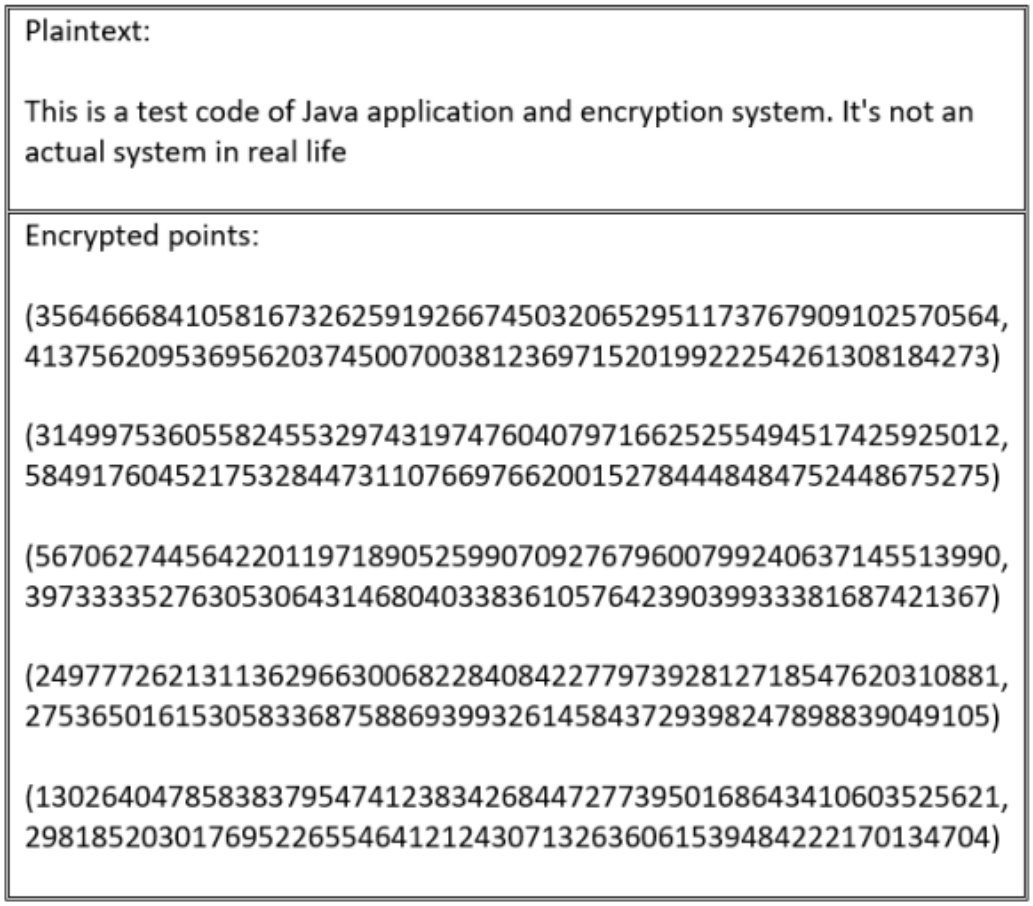


Figure 19: Cipher Text Generated by Second Run of the Encryption Function with XOR random IV Using the Same Plaintext in Figure 18 [5]

Figure 19 shows the encrypted points of the plaintext used (mentioned in Figure 19) and this plaintext is represented in 5 EC points as encrypted points after applying encoding, mapping and ECC encryption in Algorithm 11, Algorithm 12, and Algorithm 13, respectively, while random IV is used and the encrypted points are different than the ones in Figure 18 as expected. This proves that using random IV leads to different encrypted point for the same plaintext in every run of encryption. . Random IV changes which prevents the opponents from using techniques such as CPA or CCA to expose or guess the secret key. As a result, this scheme is more secure against advanced attacks.

Conclusion of Experiment 1 Results:

Experiment 1 supports the concept of non-deterministic encryption methods in ECC to prevent the vulnerabilities of predictable ciphertext generation. It ensures the variation in the encryption parameters, such as random IVs, strengthens the security. The results in Figure 16, Figure 17, Figure 18, and Figure 19 proves role of random IV in blocking such attacks like CCA and CPA, and correlate with the concepts discussed in the referenced paper [5] and emphasize the requirement of ECC in making the cryptographic systems efficient and secure for IoT. These results are acquired from SE-Enc [5] but not implemented in [3] and are very secure.

2. Experimental Settings and Results of Experiment 2 [3]

Objective:

Evaluate the successful mapping rate, which mean the number of mapped points over the number of x values (number of needed points to be mapped) ($\frac{Number\ of\ Mapped\ Points}{Total\ Number\ of\ x\ values\ Needed\ to\ be\ MApped}$), for each iteration in Algorithm 12 and number of iterations needed to map all points that are generated to the standard EC curves: secp192k1, secp224k1 and secp256k1 curves where ECC parameters are introduced in Appendix D in Figure 73, Figure 74 and Figure 75, respectively.

Methodology:

1. Elliptic Curve: The curves aimed are secp192k1, secp224k1 and secp256k1, discussed in A6.

2. Process: Make several trials to map a random set of several numeric values (x values) into the elliptic curve. The process includes the increase (increment by 1 in each round) of numerical values until the mapping is finally successful (Algorithm 12).

3. Metrics: Number of mapped points as well as the cumulative percentage of points mapped successfully out of the total number of points mapped in each round is documented.

Setup:

1. Number of Trials: One thousand numbers were chosen at random, which represents choosing any 1000 random numbers by Algorithm 12 for 1000 different x- coordinate values.

2. Random Numbers Size: 192-bit values of the curve secp192k1, 224-bit values for the curve secp224k1, and 256-bit values of the curve secp256k1.

Experiment 2 Results:

Figure 20, Figure 21 and Figure 22 illustrate the results from trials that measured the percentage of mapped points followed by each round, the number of mapped points in each rounds and the number of rounds needed to achieve a complete mapping of points to the curves secp192k1, secp224k1 and secp256k1 (Appendix D), respectively:

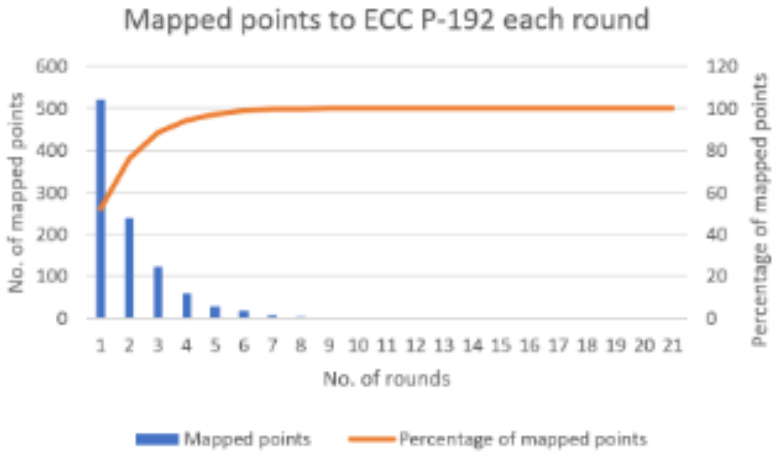


Figure 20: Number of Rounds Needed to Map Points to Secp192k1 Curve (in Appendix D) with Percentage of Mapping Points [3]

The bars in Figure 20 represents the number of mapped point in each round, while the curve represents the percentage of mapped points after each round.

Figure 20 shows 8 rounds in total to complete the mapping of points to EC with more than 50% of successful mapping points in the first round, and finishes at 100% successful mapping points at round 8. The result is good, because padding 3 bits to the right in Algorithm 17, means at most 8 rounds are allowed, which means this result is good.

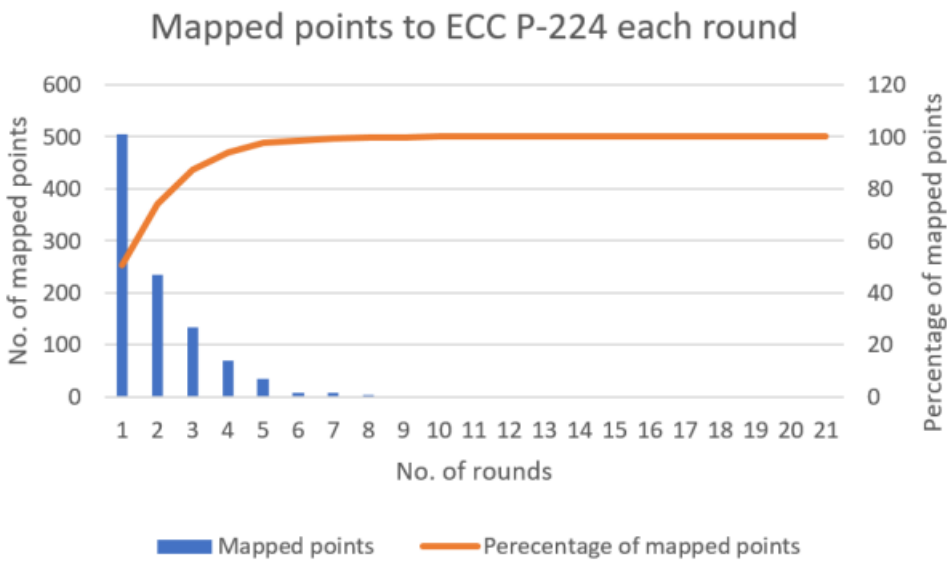


Figure 21: Number of Rounds Needed to Map Points to Secp224k1 (Appendix D) Curve with Percentage of Mapping Points [3]

The bars in Figure 21 represents the number of mapped point in each round, while the curve represents the percentage of mapped points after each round.

Figure 21 shows mainly similar results to Figure 20, where it needs 8 rounds in total to complete the mapping of points to EC with more than 50% of successful mapping points in the first round, and finishes at 100% successful mapping points at round 8. The result is good, because padding 3 bits to the right in Algorithm 11, means at most 8 rounds are allowed, which means this result is good.

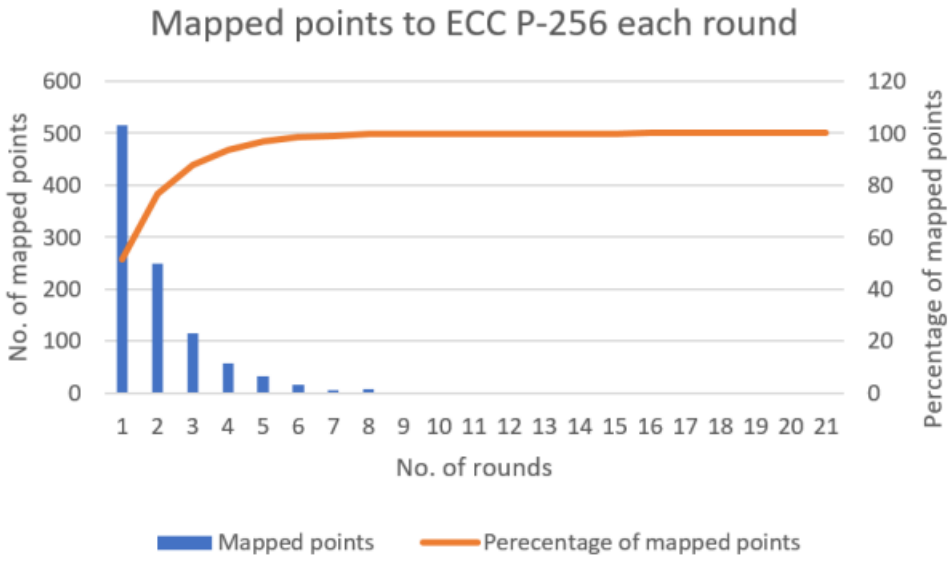


Figure 22: Number of Rounds Needed to Map Points to Secp256k1 Curve (Appendix D) with Percentage of Mapping Points [3]

The bars in Figure 22 represents the number of mapped point in each round, while the curve represents the percentage of mapped points after each round.

Figure 22 shows mainly similar results to Figure 20 and Figure 21, where it needs 8 rounds in total to complete the mapping of points to EC with more than 50% of successful mapping points in the first round, and finishes at 100% successful mapping points at round 8. The result is good, because padding 3 bits to the right in Algorithm 11, means at most 8 rounds are allowed, which means this result is good.

Conclusion of Experiment 2:

Experiment 2 shows that 8 rounds are enough to map all points to EC for all curves secp192k1, secp224k1 and secp256k1 in Figure 20, Figure 21 and Figure 22, which means 3 padding bits in Algorithm 11 are accepted based on the results, but that doesn't guarantee that 3 padding bits are enough because some other random numbers may lead to more rounds than 8, and the maximum number of rounds that allowed by the 3 padded bits is 8 rounds.

3. Experimental Settings and Results of Experiment 3 [4]

Experimental Settings:

For Experiment 3, a machine with Intel XeonE5645@2.40 GHz (6core, each core hyper threaded) and 16-GB RAM is used. The compiler is gcc version 4.4.7 (The machine is given because compilation time may have slight differences from 1 machine to another). Experiment 3 is measuring the time taken for the encryption process by Algorithm 3 and Algorithm 8, used in encryption/decryption model 1, to be done by the compiler for different size of messages, which are: 1 KB, 10 KB, 100 KB, 1 MB, and 10 MB.

In Experiment 3, two schemes are compared based on their performance (time), while the first scheme is the proposed scheme in [4], and the other one is a scheme [26] discussed in [4].

Experimental Result:

Figure 23 shows a comparison of time needed for encryption process to be compiled between Sengupta [4] and V [26] scheme discussed in [4] with respect to different sizes of message:

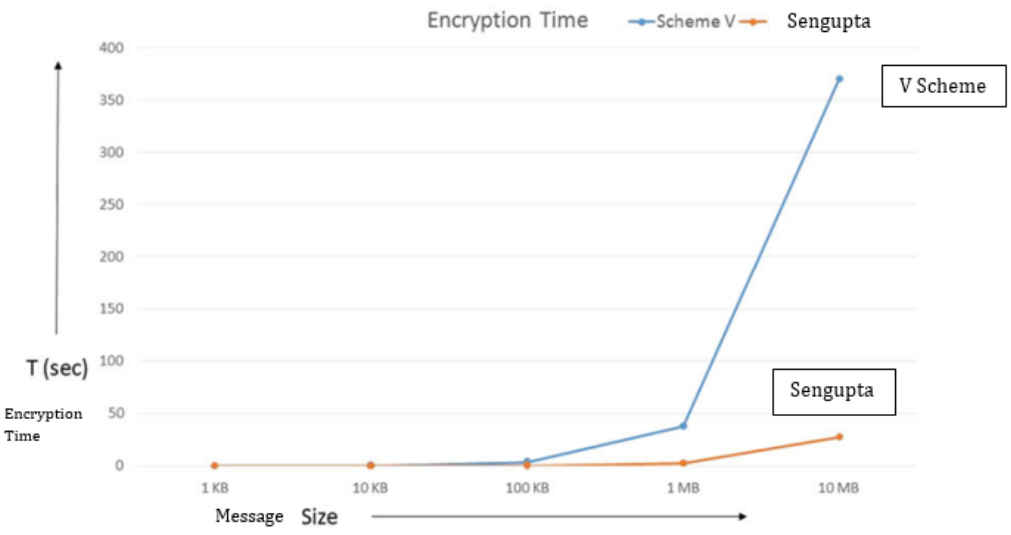


Figure 23: Encryption Performance Comparison Between Two Schemes [26] [4]

Figure 23 shows the time needed to encrypt a message (Algorithm 3 and Algorithm 8) for different size of messages of 2 schemes which are [4] and [26], where 1 KB message size and 10 KB take less than 1 second for both schemes [4] and [26].The significant different starts to appear at size 1 MB of the message, where [4] takes around 3 seconds to encrypt the message, while [26] takes around 40 seconds to run it. When substantial data is taken into account, the time required to encrypt using the [4] algorithms is significantly less than the time required by scheme V. A 10-MB message can be encrypted in 370 seconds using scheme V [26], whereas it only takes 27 seconds to encrypt the same data using scheme [4]. This significant difference can be attributed to [4]'s algorithms, which (Algorithm 3) consider M characters when mapping, which is for sure preferable than mapping each character separately.

Conclusion of Experiment 3:

Experiment 3 shows the good performance of [4] and the comparison with V (existing scheme mentioned in [4]), shows a big difference for 10-MB data. For article [3], also mapping of group of characters is discussed in Algorithm 12, sharing the same formula with [4], so do [3] has a similar performance to [4]? Problem 4 is introduced here, where checking the performance for the encryption for any scheme is a major thing to prove the good efficiency, so the running time for encryption should be similar to [4] which is considered to have good efficiency.

2.3.5 Summary of Review

SE-Enc provides a higher level of security compared to the schemes proposed by Sengupta [4] and Barman, making it the preferred choice for this study. One of the key security features of SE-Enc is the use of an IV. Even with a constant IV, SE-Enc [3] ensures robust protection against various attacks, and it discusses the implementation of random IVs, further strengthening its security mechanisms. In contrast, Sengupta's

scheme, while providing foundational cryptographic concepts, lacks the advanced security features of SE-Enc, particularly the robust IV mechanism. Barman's scheme [27] is designed for IoT security, which aligns with the goals of SE-Enc. However, SE-Enc offers a more comprehensive approach by incorporating IVs, enhancing the overall security of the encryption process.

Moreover, SE-Enc utilizes ECDSA, adding an additional layer of security through digital signatures, which is not present in Sengupta's and Barman's schemes. While Barman's article addresses IoT security and is suitable for such applications, it does not use IV or any similar idea to enhance the security. The superior security features of SE-Enc, including the use of IVs and ECDSA, make it the preferred choice for conducting this study, as it offers a more secure and resilient encryption process compared to Sengupta's and Barman's schemes.

2.4 Thesis Problem Definition

The present thesis aims to design, implement, and test a secure and efficient ECC scheme for IoT following [3], with necessary modifications to address the following problems:

1. **Block CPA and CCA**: Implement an ECC scheme that uses a random IV and trusted public keys. Adding a random IV, which is missing in [3], will help block CPA and CCA. Trusted public keys will be used to block Man-in-the-Middle (MITM) attacks (Problem 1 in 2.2.5).
2. **Determine Required Padding Bits**: Compute the number of rounds needed for mapping points to ECC to determine the appropriate number of padding bits required for efficient mapping (Problem 2 in 2.2.5).

3. **Evaluate Padding Bits Performance**: Evaluate the performance (time) of using 3 to 8 padding bits and decide whether using 3 bits is more efficient than 8 bits or not (Problem 3 in 2.3.1).
4. **Assess Encryption Time**: Evaluate the encryption time and compare it to the one in [4] (Problem 4 in 2.3.3).

Chapter 3

DEVELOPMENT OF EFFICIENT AND SECURE ECC SCHEME FROM [3]

An efficient and secured ECC scheme for IoT structure can be developed by combining the design, implementation and testing of the scheme. The design (section 3.1) of the ECC scheme involves the architecture of the network model and the optimized ECC algorithms. The implementation and testing in section 3.2 refer to the development and integration of the optimized ECC algorithms into a working system optimized for IoT. The testing also discussed in section 3.2 which evaluates the scheme developed to ensure that the algorithms are effectively encrypting and decrypting the message. This combined development of ECC scheme ensures the security and efficiency of the scheme.

3.1 Design of Secure and Efficient ECC Scheme from [3]

The overall design of the efficient and secure ECC scheme is discussed in section 3.1.1. The design phase of the efficient and secure ECC scheme for IoT will be done with much attentions to the parameters considered in section 3.1.2. In section 3.1.3 the key agreement subpart design is established. In section 3.1.4, encryption process design is discussed with its steps. In section 3.1.5, decryption process design is discussed with its steps. Finally, the discussion in section 3.1.6 involves summary of the design of all the subparts of the ECC scheme.

3.1.1 General Design of ECC Scheme

The design for an efficient and secure ECC scheme can be segmented into the four key subparts shown in this Figure 24: initialization of parameters, wherein the necessary elliptic curve parameters are set up; key agreement, in which communicating parties agree on the same key with the help of their private and public keys; and the shared secret key would act as the key to secure communication. This is then followed by a procedure for encryption, where the sender encrypts the message using the shared key. Lastly, a decryption procedure at the end will enable the receiver to decrypt the message using their private key and the shared key. Again, this well-structured approach makes it very suitable for all applications, including the resource-constrained environments like IoT, due to the attainment of security together with efficiency.

The algorithms used for this design are based on encryption/decryption model 2 in section 2.2.5, while the only major is in Key agreement represented in Algorithm 26. Figure 24 shows the design of overall subparts (3.12 - 3.15), where each subpart will be explained in details in its section, of the ECC scheme:

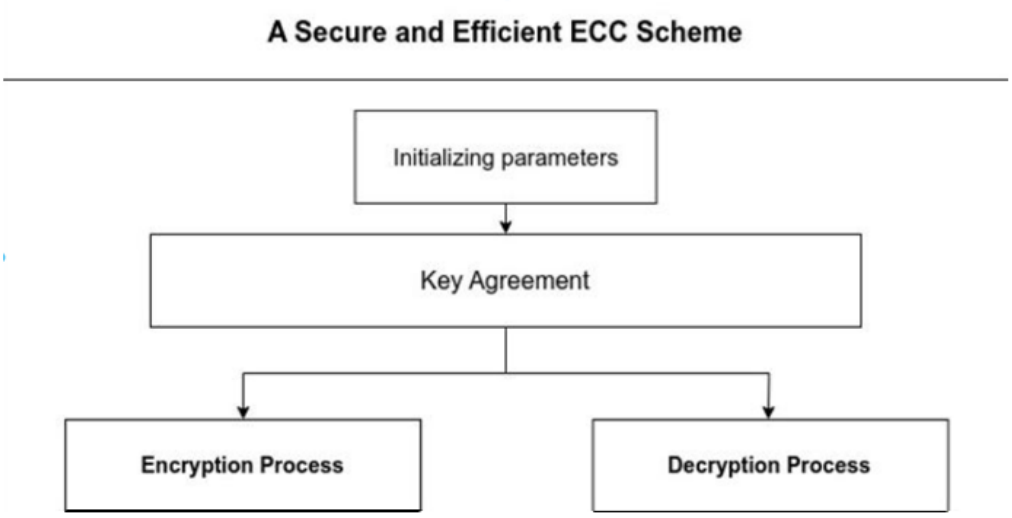


Figure 24: ECC Scheme Overall Design

3.1.2 Initializing Parameters in the Designed System

System Parameters:

The system parameters determine the configuration of the cryptographic operations within the ECC. These include:

• Prime Number (p): A prime large number that defines the field ECC operations work over. The size of the hash (like 192-bit for strong security) is the factor that governs the security level provided by the scheme.

• Elliptic Curve Coefficients (a, b): The coefficients which characterize the particular elliptic curve employed in cryptographic procedures. This equation $y^2 \, mod \, p = \, x^3 \, + \, ax \, + \, b \, mod \, p$ depends on these coefficients to characterize the curve.

• Base Point (G): A base point of elliptic curve that serves as a point of reference for the generation of public and private key pairs. The ECC scheme security and efficiency characteristics depend upon the selection of appropriate base point.

Cryptographic Keys:

The scheme's security is largely contingent on the generation, distribution, and management of cryptographic keys. The scheme's security is largely contingent on the generation, distribution, and management of cryptographic keys:

• Private Keys (d_s, d_r): Private keys with sender side and receiver, used for signature creation and decryption of incoming messages.

• Public Keys (PU_s, PU_r): Obtained from the same private keys and the base point, these ones are made public and are critical for encryption and signature verification.

• Static public keys (PUs_signed, PUr_signed): This is static public key that has been certified and digitally signed by the CA. The certification ensures that the public key is legitimate and can be trusted by the Sender and **respective private key is the** private

key corresponds to the PUs_signed/PUr_signed. Respective private key is used to sign session-specific public keys during each communication session.

Hash Function and Random Data:

• Hash Function: Used in signing the message for maintaining the integrity of the message.

• Initialization Vector (IV): A random vector used to guarantee that the same plaintext will produce different cipher texts in various encryption runs.

• Cipher Text (C_m): The encryption of the plaintext message, which will be the result of secure transmission over the network.

• Signed Messages: Digitally signed using private keys that assure the correctness and integrity of the messages.

All these parameters and components are essential for the safe and effective functioning of the suggested ECC scheme. The careful selection and controlling are the central part of system design that both security and functionality of the whole communications network depends on. Subsequent sections will also cover the architecture that makes use of these parameters and the detailed methodology for building the scheme.

Table 1 shows the parameter notations and description taken from [3] with simple modifications:

Table 1: Notation Used in the Scheme [3]

Notation	Description
d_s	Sender private key which is random number between 2 and p-1, both included
d_r	Receiver private key which is random number between 2 and p-1, both included
G	EC base point
PU_s	Sender public key = ds * G
PU_r	Receiver public key = dr * G
p	Large prime number
a, b	EC coefficients, such that 4a^3 + 27b^2 mod p ≠ 0
$y^2 \ mod \ p = x^3 + ax + b \ mod \ p$	EC map points equation
HASH	Signing message Cm hash function
ksh	Shared group key
IV	Random initial vector ((size of p) – 8 bits)
k	Random integer chosen from [2, p - 1]
C_m	Cipher text (all encrypted points)
M	Message sent (Plain text)
+	Addition operation used in ECC to encrypt mapped points with ksh using (11)
IVS	A variable calculated by IV XOR x_{ksh}, used for sending IV securely
CA_pub	Public key for Certificate Authority (CA)
PUs_signed	Sender's static public key
PUr_signed	Receiver's static public key

ECC parameters (a, b and p) and base point G are global variables so no need to include them in the input of algorithms.

3.1.3 Key Agreement Design

The key agreement phase in an ECC-based system allows two parties (the sender and receiver) to securely establish a shared secret key. This phase leverages the properties of elliptic curves to ensure that the shared key is derived securely without exposing the private keys of the parties involved. Also, a random IV is generated here and shared between the two parties. This step is modified and not mentioned in [3].

Figure 25 illustrates the key agreement process between the sender and the receiver:

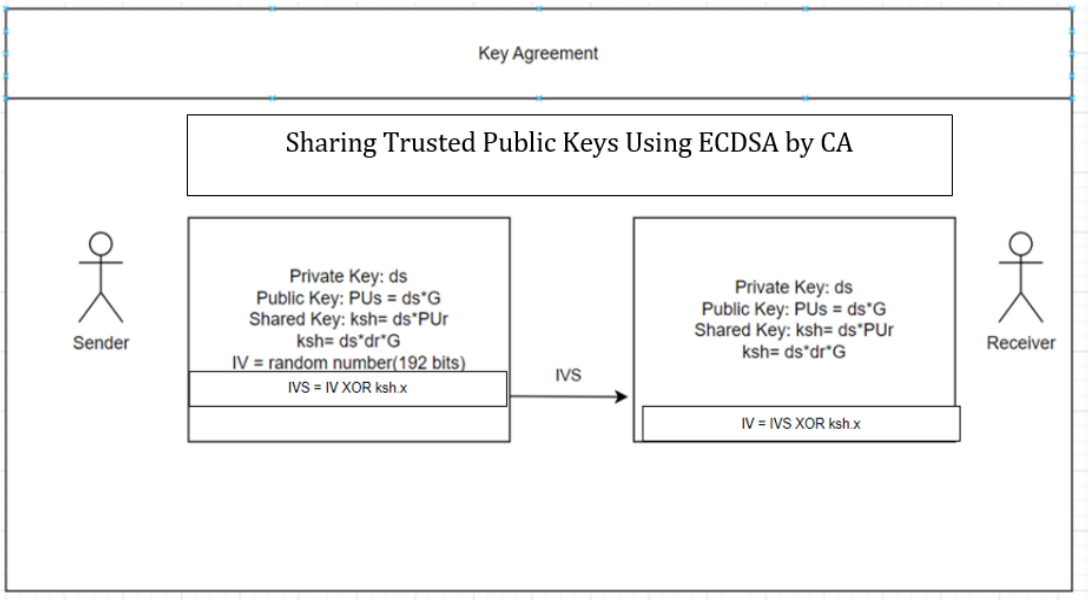


Figure 25: Key Agreement Subpart

Algorithm 21: Key Agreement (Modification of Algorithm 10 in 2.2.5)

The pseudo code for Key Agreement is represented in Algorithm 21:

Algorithm 21: Thesis's ECC Scheme Key Agreement Algorithm

Key Agreement Algorithm	
Inputs: - PUs: Sender's Public Key - ds: Sender's Private Key - PUr: Recipient's Public Key - dr: Recipient's Private Key - G: Curve Generator Point - ECC parameters: (a, b, p) - PUs_signed: Sender's static public key signed by a trusted third party (Certificate Authority) - PUr_signed: Recipient's static public key signed by a trusted third party (Certificate Authority) - CA_public_key: Public key of the Certificate Authority used to verify PUs_signed and PUr_signed Note: The CA-signed public keys PUs_signed and PUr_signed must be trusted and verified against the CA's public key before the exchange and verification phase. Algorithm:	//Explanation: /*The Signing Message function takes the public key (either PUs or PUr), the respective private key (either ds or dr), the curve generator point G, and ECC parameters (a, b, p) as inputs, and produces a signature (r, s). This signature ensures the authenticity and integrity of the public key being exchanged */ 3. Verification Phase: a. The Sender verifies the received PUr and its signature (r, s) using the Verifying Message function: - Verifying Message(PUr, (r, s), PUr_signed, G, a, b, p) using ECDSA in Algorithm 6 b. The Recipient verifies the received PUs and its signature (r, s) using the Verifying Message function:

1. Certificate Validation: a. Sender and Recipient: - Each party uses the Verify CA Signature function to validate PUs_signed and PUr_signed against the CA_public_key:
- Verifying Message(PUs_signed, (r, s), CA_public_key, G, a, b, p) using ECDSA in Algorithm 6 /* (r, s) is the signature when CA already signed public keys for sender and receiver */
- Verifying Message(PUr_signed, (r, s), CA_public_key, G, a, b, p) using ECDSA in Algorithm 6
//Explanation:
/* The Verify CA Signature function takes a signed public key (either PUs_signed or PUr_signed), the CA_public_key, the curve generator point G, and the ECC parameters (a, b, p). It verifies that the public key was signed by the CA and has not been tampered with. Successful validation confirms that the public key is authentic */
2. Public Key Exchange (No CA Involvement in Each Session):
a. The Sender signs the public key PUs using the Signing Message function:
- Signing Message(PUs, ds, G, a, b, p) using ECDSA in Algorithm 5
b. The Recipient signs the public key PUr using the Signing Message function:
- Signing Message(PUr, dr, G, a, b, p) using ECDSA in Algorithm 5
c. The Sender and Recipient then exchange PUs, PUr, and their corresponding signatures (r, s) from the above step.
- Verifying Message(PUs, (r, s), PUs_signed, G, a, b, p) using ECDSA in Algorithm 6
//Explanation:
/*The Verifying Message function takes the received public key (either PUr or PUs), the received signature (r, s), the signed public key PUs_signed or PUr_signed (previously validated against the CA's public key), the curve generator point G, and the ECC parameters (a, b, p) to validate the authenticity of the received public key. This ensures that the public keys were indeed signed by the sender's private key and verified using the corresponding signed public key */

4. Scalar Multiplication (Parallel Execution):
a. The Sender computes K_s = ds * PUr and sends the result to the Recipient.
b. The Recipient computes K_r = dr * PUs.

5. Shared Key Calculation (by Recipient):
a. If K_s == K_r:
i. The shared key ksh is set to K_s.
b. Else:
i. Output "Key exchange failed" and terminate.
6. Initialization Vector (IV) Sharing:
a. The Sender generates a random IV.
b. The Sender computes IVS = IV XOR ksh.x.
c. The Sender sends IVS to the Recipient.
d. The Recipient computes IV = IVS XOR ksh.x.

End Algorithm

This pseudo code is used to ensure secure shared secret key derivation between the sender and receiver without sharing their private keys directly, and using elliptical curves for secure key exchange. These are all done in 2.2.5 except steps related to IV. Here the trusted public keys are used by using certificate authority (CA), which is a trusted third party. The public key and private key for CA are generated by CA, and used for signing and verifying the public key for both sender and receiver as shown in Algorithm 21. Signing and verifying are done using ECDSA in Algorithms 5 and 6, respectively. Now using trusted public keys will lead to block MITM attack as proved in 3.3.

3.1.4 Encryption Process Design

The encryption process which is carried out in the ECC-based secure communication scheme entail, the transformation of plaintext to cipher mapped points in the coming steps shown in Figure 26. This process provides security in that the data becomes unknown for unwanted parties. Figure 26 shows the encryption process consists of encoding the plaintext, conversion of the numerical value to elliptic curve, encryption of the mapped points and the signing of the message. Figure 26 illustrates the steps for encryption process:

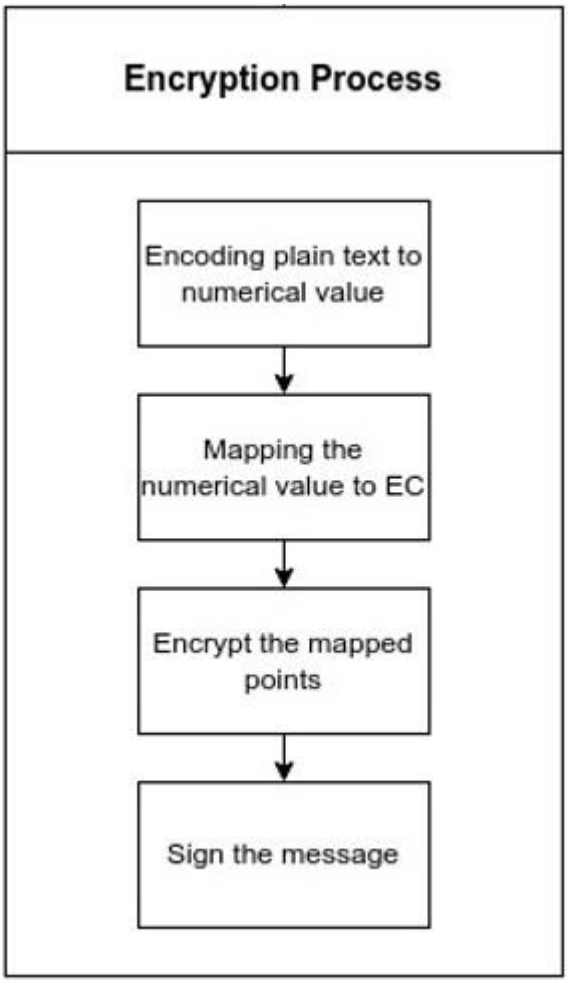


Figure 26: Encryption Process Subpart [3]

Figure 26 shows the encryption process subpart in details and steps, while in [3] it is introduced as steps in Figure 10, but here a subpart is taken to show the exact steps that are used for encryption process.

Section 3.1.3 will cover 4 steps. Starting with encoding plaintext to numerical values which will be discussed in part 1. Then, mapping the numerical value to EC will be discussed in part 2. Moving to encrypt the mapped points which will be discussed in part 3. Finally, sign the message will be discussed in part 4.

1. Encoding Plaintext to Numerical Value:

This step includes 2 main parts which represented in Algorithm 22 (which is a modification of Algorithm 11 in 2.2.5):

Algorithm 22: Encoding Plaintext to Numerical Values in Thesis's ECC Scheme Algorithm

Encoding the Message to Numerical Values Algorithm

// Input: Plaintext M, large prime number p, Random Initial Vector IV
// Output: Encoded message
 	
begin
// Step 1: Determine Block Size and Divide Message
1. Determine the block size N based on p, N = ⌊(number of bits(p) − 8) / 8.⌋
2. Calculate the number of blocks B needed for plaintext M, B = ⌈number of characters (M) / N⌉.
3. Divide M into B blocks of size N (last block size can be less than or equal to N).
4. For each block, encode the characters into ASCII values.
5. Combine the ASCII values to form the blocks B. // set of B now is obtained
6. For each block Bi in B do
6.1. Bi' = Bi XOR IV
6.2. IV = Bi'
end for
 	
// Step 7 and 8: Pad and Return Encoded Message
7. Pad 5 bits of zeros to the right and 3 bits to the left of each XORed block.
8. Return Encoded message.
End

Algorithm 22 is different than Algorithm 11 which use in its step 8 three padding bits, while here step 8 in Algorithm 25, five padding bits are used.

1.1. Converting plaintext to Encoded blocks

Converting plaintext to Encoded blocks is a base element in plaintext preparation for encryption in the ECC structure, making it to be transformed into a set of blocks. This step is crucial to make sure that the encryption stage works on small and secure data segments, compliant with the field size of the elliptic curve.

5 padding bits are used here on the right (Least Significant Bit).

1.2. Securing blocks against encryption attacks using IV

Block security is enhanced in algorithm 16 by securing blocks for resistance against encryption attacks such as Chosen Plaintext Attack and Chosen Ciphertext Attack employing Cipher Block Chaining (CBC) [3], which is an encryption mode where each plaintext block is XORed with the previous ciphertext block before being encrypted. This ensures that identical plaintext blocks produce different ciphertexts, enhancing security. The algorithm processes each block of the message with an XOR operation with an Initial Vector (IV). This stage is very important for preventing the possible attacks on encryption that are aimed at predictability and patterns inside the blocks.

The CBC and IV are used in the process of securing blocks in order to prevent identical blocks of plaintext from producing identical ciphertext blocks, a common weakness in simpler encryption methods. This approach introduces complexity to the would-be attackers in analyzing patterns in the encrypted data which strengthens the communication channel against highly sophisticated encryption attacks.

The use of the XOR operation in CBC mode is especially powerful because it introduces a measure of randomness and dependence on the previous block's encryption, making it practically impossible for an attacker to crack the encryption without the proper key. The scheme by using the consistent changing of the Initial Vector IV after each block is processed guarantees that the encryption of each block is unique. Therefore, even if the same plaintext is encrypted multiple times a strong level of security is provided.

2. Mapping the Numerical Value to Elliptic Curve (similar to Algorithm 12 in 2.2.5)

When the secured blocks are ready, Algorithm 12 is activated to carry out the mapping of these blocks onto the elliptic curve, a fundamental step in ECC encryption of information. This procedure converts the numbers obtained from the secure blocks into points on the elliptic curve using the curve (8).

This step is done to find the point on elliptic curve by using x coordinate and if this point doesn't exist, then x will be incremented by 1 and the search for the point starts again till finding the point on EC.

In ECC this mapping process is essential since the encryption and decryption operations are mathematical operations on these points. Highly efficient data mapping to elliptic curve points not only ensures the security of the encryption process but also its efficiency, which is especially important when dealing with the limited resources and high-performance requirements of IoT.

3. Encrypt the Mapped Points (refer to Algorithm 13 in 2.2.5)

In this scheme, Algorithm 13 is very important, as it deals with the encryption of mapped points using the Elliptic Curve Cryptography (ECC), making sure the integrity of the data as it is transmitted over the network. This is an essential step in turning the readable data into a safe format that can only be read by authorized parties with the appropriate shared key.

Algorithm 27 not only guarantees data confidentiality but also embeds into the framework of ECC, taking advantages of strengths of ECC providing effective security features. The encrypted points can be securely sent via public or insecure networks without exposing the original data, protecting the data from eavesdropping [28], which

is the unauthorized interception and listening to private communications or data transmissions between parties. It is a common security threat where an attacker gains access to the information being transmitted without the knowledge or consent of the participants.

4. Sign the Message

Process of Signing the Message (by Algorithm 5 in 2.2.3).

Algorithm 5 deals with the sign of the message using elliptic curve cryptography that is essential in ensuring the integrity and authenticity of the message's transmission. Here, $r = k \times G$.

More information is discussed in 2.2.3.

3.1.5 Decryption Process Design

While the encryption process in the ECC-based secure communication scheme are used to convert a message into cipher mapped points, the decryption process is designed to reverse encryption process and convert the cipher mapped points back to plaintext. This process also helps to ensure that the data and its content can be understood or read by the target recipient the check for the message integrity and authenticity. The detailed decryption process is also presented in the form of the diagram in Figure 27 that consist several steps including Verify the message, Decrypt the cipher mapped points, decode hex value, and Hex values to plaintext.

Figure 27 illustrates the decryption process:

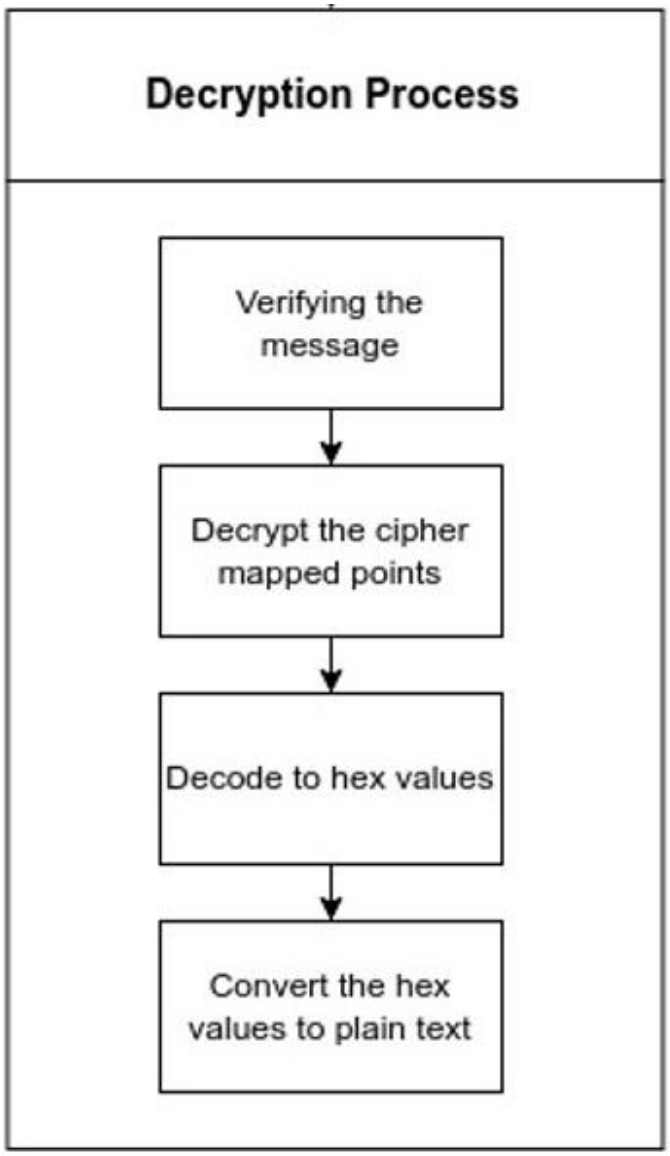


Figure 27: Decryption Process Subpart [3]

Figure 27 shows the decryption process subpart, related to Figure 10, while Figure 10 discusses the steps for the whole scheme [3], without showing the exact steps for decryption, so Figure 27 shows the steps needed for the decryption process.

Section 3.1.4 covers 4 steps. Starting with verifying the message which will be discussed in part 1. Then, decrypt the cipher mapped points will be discussed in part 2. Moving to decode to hex values which will be discussed in part 3. Finally, convert the hex values to plaintext will be discussed in part 4.

1. Verifying the Message

Recipient Verification of a Signed Message (by Algorithm 6 in 2.2.3).

The receiver of the message can check the signature (r, s) with the sender's public key. This process entails a number of steps to see to it that the signature matches the hash of the message when the message is processed with the public key and therefore, it proves that the message was not altered and it comes from the sender as claimed. Here it should be proved that r = x_1, while $(x_1,\ y_1) = u_1 \times G + u_2 \times PU$.

The signing process is an essential part of the security process in any ECC based system as it enables a way to authenticate the messages and also to ensure that they have not been altered. It should be noted that the said characteristics are of utmost importance when security and trust are the primary concerns, especially in areas of applications like IoT where devices usually inter-communicate through insecure networks.

More information is provided in section 2.2.3.

2. Decrypt the Cipher Mapped Points

Decrypting Cipher Mapped Points by Algorithm 14 in section 2.2.5.

Algorithm 14 ensures decrypting the mapped points, which is an important step here in order to proceed with the remaining steps. Algorithm 14 uses ksh for decryption which is known only by the 2 parties, which means the encrypted mapped points can only be decrypted by the receiver.

3. Decode to Hexadecimal Values

Algorithm 23: Decoding Decrypted Points to Hexadecimal Values

The pseudo code of decoding decrypted points to hexadecimal values is represented in Algorithm 23:

Algorithm 23: Decoding Decrypted Points to Hexadecimal Values

Decoding Decrypted Points to Hexadecimal Values Algorithm
Input: Mapped points, random IV Output: Decoded block 1. Recipient: obtain xi value for the mapped point; 2. Recipient: convert xi to the binary value; 3. Recipient: remove the padding 5 bits for each block; 4. Recipient: XOR first block with IV; 5. Recipient: for each block, XOR it with the previous block; 6. Recipient: repeat step 4 for all blocks; 7. Hex values <- convert the results to hex; 8. Return Decoded Blocks

Algorithm 23 processes the part of the decoded points and converts them into hex values, while XORing by IV is also used as well. Algorithm 23 is necessary for transformation of elliptic curve coordinates into hex values that can be converted to characters.

4. Convert Hex Values to Plaintext

Algorithm 24: Converting Hexadecimal Values to Plain Text

The pseudo code of Algorithm 24 is shown below:

Algorithm 24: Converting Hexadecimal Values to Plaintext Algorithm

Converting Hexadecimal Values to Plain Text Algorithm
Input: Hex values Output: Plaintext message M 1. Get the hex values 2. for i from 0 to number_of_hex_values - 1 do 3. Convert each hex value to its corresponding 4-bit binary string 4. Combine binary strings to form the full binary representation 5. Convert each set of 8 bits to its corresponding ASCII character 6. Aggregate to form the message M 7. end for Return M

Algorithm 24 is apparently necessary for the representation of hex values to readable plaintext format that comes after their decryption and decoding in previous steps.

Algorithm 24 is the last phase of secure communication with ECC during which encrypted data travels safely, gets decrypted and converted to its original state, thereby guaranteeing that the information remains confidential and intact. The performance of this algorithm, more specifically its ability to convert binary data back into plain text accurately and fast play a crucial role in the function and security of the communication system.

3.1.6 Design Phase Summary

The design phase of this thesis has carefully defined the subparts of an ECC-based secure communication scheme designed for IoT environments. This stage has provided a strong base for the deployment of a system, which focuses on both security and efficiency.

The full design for the secure and efficient ECC scheme is shown in Figure 28:

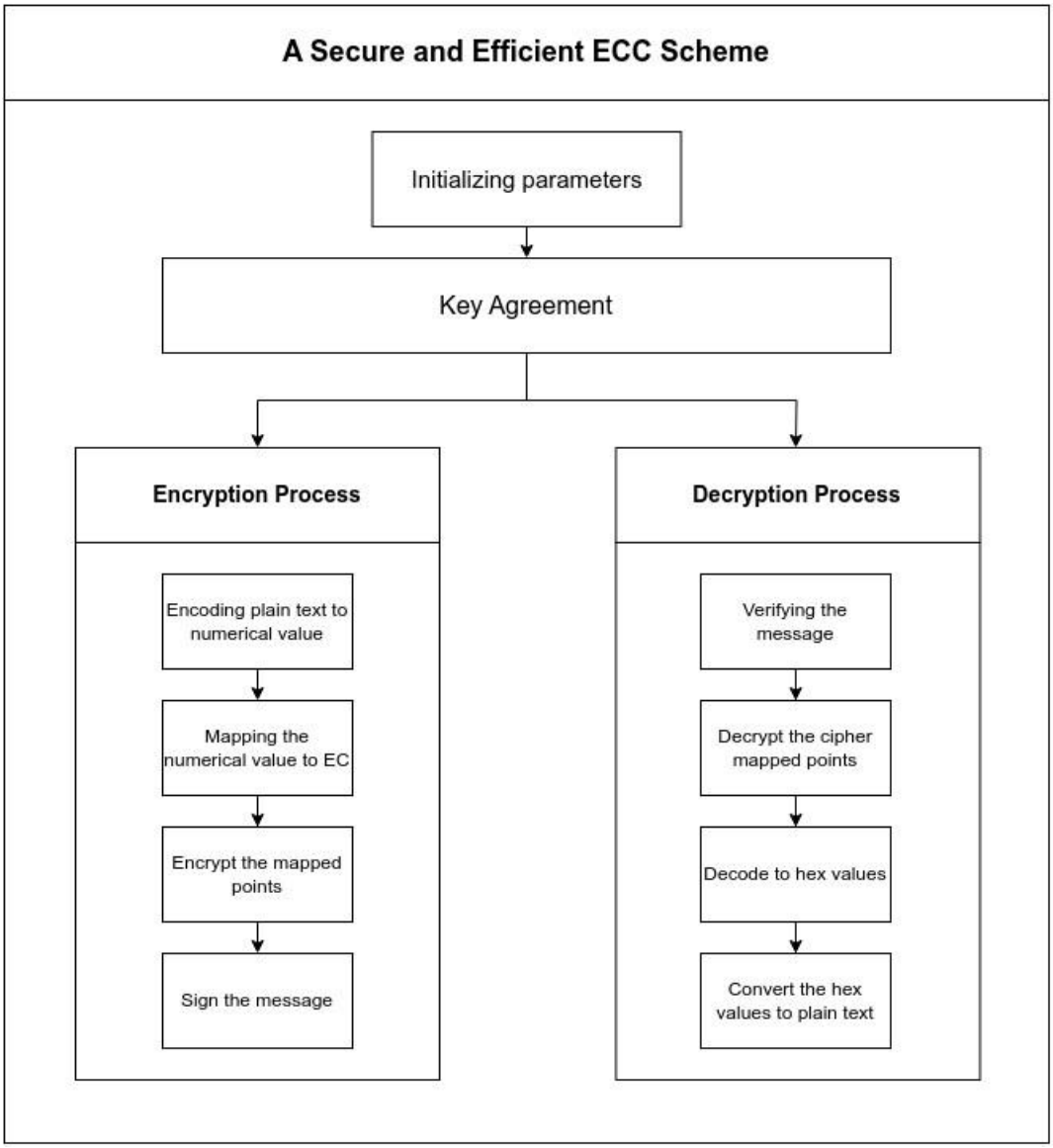


Figure 28: Connected Subparts Design for Secure and Efficient ECC Scheme [3]

Figure 28 connects all the subparts with their major steps, to show a summary for this thesis's scheme design.

The design of the secure and efficient ECC scheme can be explained by breaking down its four main parts and explaining how they are linked together. These parts are: Parameters of the designed ECC Scheme, Key Agreement, Encryption Process, and Decryption Process.

Linking the Parts:

The flowchart effectively illustrates how these parts are linked:

1. The Parameters of the Designed ECC Scheme includes all the basic details which are used for all other functionalities.

2. These parameters are utilized in the Key Agreement phase to achieve a safe and sound sharing of a symmetric key.

3. In the Encryption Process, the shared secret key is then applied to change the plaintext to the ciphertext (encoding and mapping are done before encryption), and the same is for the Decryption Process to provide secure transference of information.

4. The Decryption Process is the inverse of the Encryption Process, utilizing the shared secret key to decrypt the ciphertext back into the plaintext, thus securing the message and its contents from any misinterpretation, and at last decoding appears to have a decrypted message which contains characters.

3.2 Implementation and Testing of Secure and Efficient ECC Scheme from [3]

Implementation and testing the ECC scheme for Internet of Things focuses in section 2.3.1 on the use of appropriate programming languages, cryptographic libraries and development environments in order to produce a working and optimized system in section 3.2.1. Whereas, in section 3.2.2 it consists of developing, integrating, and testing ECC algorithms used for key agreement, encryption, decryption and digital signatures in order to achieve seamless communication and run the system. Finally, in section 3.2.3, a summary for implementation and testing the ECC scheme is discussed.

3.2.1 Tools Used for Secure and Efficient ECC Scheme

The implementing tools refer to the development environment, programming language and cryptographic libraries that the ECC scheme for IoT structure will use. These tools are considered critical in the implementation phase since they might introduce security vulnerabilities if wrong tools are selected. The development environment in part 1 refers to the integrated tool and platform where the code that implements the ECC scheme is written, tested and debugged. The programming language in part 2 refers to the programming language that should be suitable and enough expressive for the cryptographic operations to be performed. The cryptographic libraries in part 3 refer to the pre-implemented cryptographic algorithm functions that are needed in the ECC scheme and are used to ensure both security and efficiency of the cryptographic operations. The structure of these tools guarantees that the ECC scheme will be implemented in an efficient, secure way.

1. Development Environment

Visual Studio Code

Microsoft has developed Visual Studio Code which is a highly flexible and dynamic editor. Its detailed feature set, along with plugin support, makes it a top choice. The user-friendly interface of VS Code, paired with its language support, makes it an excellent platform for developing ECC-based schemes.

Key Features of Visual Studio Code:

Visual Studio Code is notable for its key features, which include:

- Cross-platform support (Windows, macOS, Linux)
- Automatically integrates Git version control into the development environment.
- A wide range of extensions and plugins are available.

- Debugging capabilities
- Integrated terminal

2. Programming Language

JavaScript

JavaScript is a high-level, interpreted programming language commonly used for web development. However, its versatility and extensive library support make it suitable for various applications, including cryptographic implementations. The integration of non-blocking I/O in JavaScript accelerates the delivery of IoT applications.

Key Features of JavaScript:

- Lightweight and fast
- Asynchronous processing capabilities
- Its technology selection is characterized by a wide range of libraries and frameworks.
- Easy integration with various platforms and technologies.

3. Libraries Used

The implementation of ECC scheme for IoT structure incorporates two open source libraries: BN.js and bcrypt.js. Part 3.1 introduces BN.js – a library that deals with arbitrary-precision arithmetic. This becomes necessary when one considers the large integers that are involved in the ECC operations and their eventual application in the cryptographic algorithms. Part 3.2 introduces bcrypt.js – a library that is used to hash and protect sensitive information (cryptographic keys and the like). This offers an additional layer of protection and security for the key. Both libraries are combined to produce an efficient and secure ECC scheme.

3.1.BN.js

BN.js is a JavaScript library for arbitrary-precision arithmetic. Evidently, the

cryptography of elliptic curves necessitates the effective management of large integers. By using BN.js, the mathematical methods developed for javascript guarantee the precision and safety of ECC implementations.

Key Features of BN.js:

- Arbitrary-precision arithmetic
- A wide range of mathematical operations is enabled, including the four fundamental operations: addition, subtraction, multiplication, and division.
- Performance optimization for large integers
- Compatibility with browser environments

3.2. bcrypt.js

bcrypt.js libraries rely on the Bcrypt algorithm to securely hash passwords. It's designed to securely store passwords using a secure and reliable scheme. The library's ECC-based scheme is secured by the cryptographic keys' integrity and confidentiality through the use of robust hashing functions.

Key Features of bcrypt.js:

- To provide security and data protection, bcrypt is used as a secure password hashing algorithm during the development life cycle.
- Salt generation for enhanced security
- Applies both asynchronous and synchronous algorithms for hashing operations without issues.
- Compatibility with browser environments

The use of tools and libraries allows us to develop a trustworthy IoT scheme with impressive performance and security. A common framework for building software combines Visual Studio Code, JavaScript, BN.js, and bcrypt. The js infrastructure is

known for being reliable and scalable, making it an ideal fit for our cryptographic system.

3.2.2 ECC Scheme Implementation and Testing

The implementation and testing for the ECC scheme involves major steps to ensure that the system runs securely and efficiently. The overall implementation structure and workflow are presented in part 1. The initial parameter which produces the necessary cryptographic keys for encryption and decryption is presented in part 2. Part 3 presents the implementation and testing for the key agreement subpart. The implementation and testing for the encryption process is discussed in part 4. While the implementation and testing for ECC decryption process is presented in part 5. Finally, the testing for the overall ECC scheme is presented in part 6.

1. Implementation Structure

The development of the secure and efficient ECC-based scheme requires organizing JavaScript files into various directories. This section provides a detailed breakdown of the project's file structure and the purpose of each file within it.

The system structure is shown in the Figure 29:

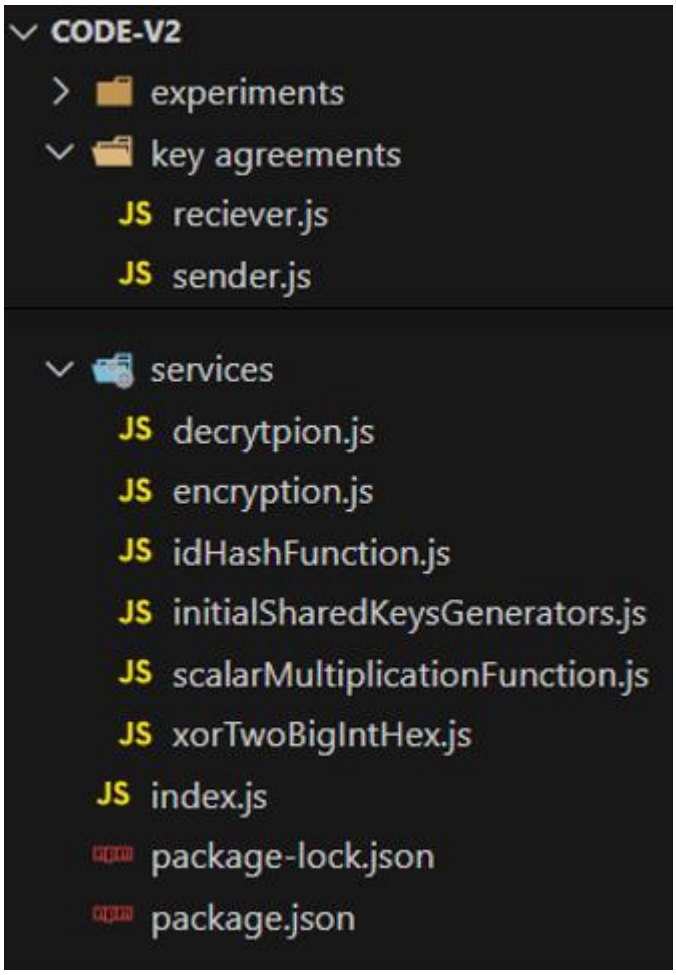


Figure 29: System Structure

Root Directory:

1. index.js: The primary route into the application is the starting point. This system file is in charge of coordinating the interaction between various modules.
2. package.json: Contains information about the project, including dependencies and scripts.
3. package-lock.json: Generated automatically, the file locks the specific version of dependencies used in the project.

Key Agreement:

1. sender.js: It consists of methods for generating shared group key for the sender by using the private key of sender, public key of receiver and base point G.
2. receiver.js: It consists of methods for generating shared group key for the receiver by using the private key of receiver, public key of sender and base point G.

Services Directory:

The utility functions and services that are essential for the system's operation are contained in the services directory.

1. decryption.js: The decryption process is handled by the functions specified in this file. The procedure includes steps for decrypting mapped points and converting them back to plain text. This function includes decoding also.

2. encryption.js: The functions in this file are designed to encrypt data. The encryption of mapped points is accomplished with the shared group key. This function includes encoding also.

3. scalarMultiplicationFunction.js: Inside this file are the functions for carrying out scalar multiplication operations on elliptic curves, a critical aspect of ECC. Point Addition and point doubling methods are also implemented inside this file (scalar multiplication uses point addition and point doubling in its steps).

4. xorTwoBigIntHex.js: The file contains functions for performing XOR operations on large integers represented in hexadecimal format. This technology is used for secure encoding and key generation.

5. idHashFunction.js: This file contains a hash function used for signing and verifying the messages.

6. initialSharedSecretKeyGeneration.js: This file is used to compute ksh for both parties.

Explanation:

1. Initialization: The index.js file initializes edge and node instances, setting up initial keys and controlling component interactivity.

2. Edge and Node Management: edgeClass.js and nodeClass.js handle the secure transmission of data by encrypting and decrypting messages.

3. Encryption and Decryption: Begins with encryption.js and decryption.js. Secure data transmission is ensured by js files through the process of encrypting and decrypting messages.

4. Utility Functions: The file called scalarMultiplicationFunction.js, and xorTwoBigIntHex.js. Essential utility functions in JS support the core cryptographic operations.

The structure has been designed to be easily updated and extended.

2 Initializing Parameters

Figure 30 shows the implementation for EC parameters initialization:

```
26  const a = new BN("2");
27  const b = new BN("7");
28  const p = new BN("6277101735386680763835789423176059013767194773182842284081");
29  const G = {
30    x: new BN("602046282375688656758213480587526111916698976636884684818"),
31    y: new BN("174050332293622031404857552280219410364023488927386650641"),
32  };
```

Figure 30: Initializing Parameters

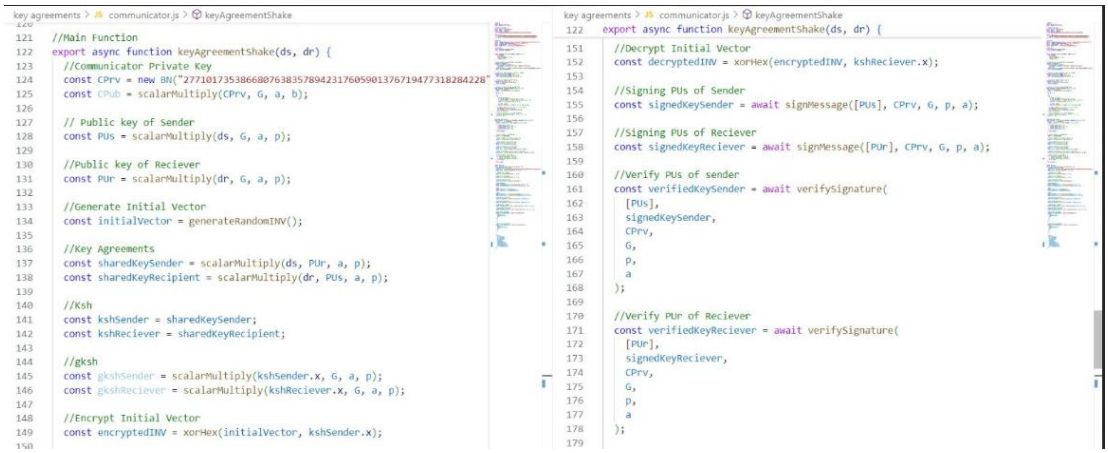
3. Key Agreement Implementation and Testing

In the key agreement (Algorithm 21), both the sender and the receiver generate key pairs of their own (private key and public key by further calculations can be determined). And both are able to compute the same shared secret key independently

using its respective private key and other party's public key. This shared key is used to encrypt further or to decrypt messages, enabling a secure communication between the sides.

In Figure 31, the implementation for the key agreement is done, while Algorithm 21 is implemented, and both sender and receiver determine their private and public keys. Also, a comparison for the shared secret key is done to ensure that both parties have the same shared secret key.

Figure 31 shows the implementation for shared key generation (ksh) for both parties:



```
key agreements > JS communicator.js > keyAgreementShake
121  //Main Function
122  export async function keyAgreementShake(ds, dr) {
123    //Communicator Private Key
124    const CPrv = new BN("27710173538668076383578942317605901376719477318284228
125    const CPub = scalarMultiply(CPrv, G, a, b);
126
127    // Public key of Sender
128    const PUs = scalarMultiply(ds, G, a, p);
129
130    //Public key of Reciever
131    const PUr = scalarMultiply(dr, G, a, p);
132
133    //Generate Initial Vector
134    const initialVector = generateRandomINV();
135
136    //Key Agreements
137    const sharedKeySender = scalarMultiply(ds, PUr, a, p);
138    const sharedKeyRecipient = scalarMultiply(dr, PUs, a, p);
139
140    //Ksh
141    const kshSender = sharedKeySender;
142    const kshReciever = sharedKeyRecipient;
143
144    //gksh
145    const gkshSender = scalarMultiply(kshSender.x, G, a, p);
146    const gkshReciever = scalarMultiply(kshReciever.x, G, a, p);
147
148    //Encrypt Initial Vector
149    const encryptedINV = xorHex(initialVector, kshSender.x);
150
```

```
key agreements > JS communicator.js > keyAgreementShake
122  export async function keyAgreementShake(ds, dr) {
151    //Decrypt Initial Vector
152    const decryptedINV = xorHex(encryptedINV, kshReciever.x);
153
154    //Signing PUs of Sender
155    const signedKeySender = await signMessage([PUs], CPrv, G, p, a);
156
157    //Signing PUs of Reciever
158    const signedKeyReciever = await signMessage([PUr], CPrv, G, p, a);
159
160    //Verify PUs of sender
161    const verifiedKeySender = await verifySignature(
162      [PUs],
163      signedKeySender,
164      CPrv,
165      G,
166      p,
167      a
168    );
169
170    //Verify PUr of Reciever
171    const verifiedKeyReciever = await verifySignature(
172      [PUr],
173      signedKeyReciever,
174      CPrv,
175      G,
176      p,
177      a
178    );
179
```

Figure 31: Key Agreement Implementation

Key agreement is implemented on both sides, but the operation is done as shown in Figure 31 in the main page. Also, the encryption of IV is shown, and decryption as well.

Figure 32 shows the implementation for useful functions for Key agreement implementation:

```
22  function pointAddition(P, Q) {
23    if (P.x === Q.x && P.y !== Q.y) return { x: 0n, y: 0n }; // P + (-P) = 0
24    if (P.x === 0n && P.y === 0n) return Q; // P + 0 = Q
25    if (Q.x === 0n && Q.y === 0n) return P; // 0 + Q = P
26
27    let m;
28    if (P.x === Q.x && P.y === Q.y) {
29      m = mod((3n * P.x * P.x + a) * inverseMod(2n * P.y, p), p);
30    } else {
31      m = mod((Q.y - P.y) * inverseMod(Q.x - P.x, p), p);
32    }
33
34    const xR = mod(m * m - P.x - Q.x, p);
35    const yR = mod(m * (P.x - xR) - P.y, p);
36
37    return { x: xR, y: yR };
38  }
39
40  export function scalarMultiplication(k, P) {
41    let Q = { x: 0n, y: 0n };
42    let N = P;
43
44    while (k > 0n) {
45      if (k & 1n) {
46        Q = pointAddition(Q, N);
47      }
48      N = pointAddition(N, N);
49      k >>= 1n;
50    }
51
52    return Q;
53  }
```

Figure 32: Scala Multiplication and Point Addition Functions Implementation

Figure 32 shows the implementation for point addition, inverse modular, and scalar multiplication which are used in key agreement process.

Key agreement implementation is manually tested to make sure it works with no issues. It consists of running the code and testing that both the sender and the receiver have the same secret key. The shown result in Figure 33 shows private, public and shared keys of both the parties, sender and receiver, which proves the verification of the key agreement process. The output is shown in Figure 33 as followed:

```
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Signature for Sender: {
  r: <BN: e1b794a9c0a911ea63464bccb256cc569acca5c1e5272282>,
  s: <BN: 32b1e483bfca1650a78f6b551a334eee371689da861abe2d>
}
Signature for Reciever: {
  r: <BN: d41d80eca334db2dcbba6bdb8623e46611aeb73ee0e2a6af>,
  s: <BN: 6279bd5d3827b32e3f264c3c5af33aceb689270b9fe88366>
}
Verification for Sender: true
Verification for Reciever: true
Shared Key for Sender: {
  x: <BN: 92514b4d7e0a9c56fb81a22d526e4708295be195865a051b>,
  y: <BN: a8d57fc884ab04e6ff8d341885bfab7fe4381dc3231bde18>
}
Shared Key for Reciever: {
  x: <BN: 92514b4d7e0a9c56fb81a22d526e4708295be195865a051b>,
  y: <BN: a8d57fc884ab04e6ff8d341885bfab7fe4381dc3231bde18>
}
```

Figure 33: Key Agreement Testing

As shown in the output above that sender has been able to generate their private and public keys. The same goes for the receiver as well. The most important part is that both parties have now the same shared secret key, validating the key agreement process.

This proves for us that the ECC key agreement implementation is correct and reliable, and allows both sides to use their private public keys to establish a shared secret key for subsequent encrypted communication.

4. Encryption Process Implementation and Testing

Part 4 will cover 4 steps. Starting with the implementation of encoding plaintext to numerical values, in addition of testing it which will be discussed in 4.A. Then, the implementation of mapping the numerical value to EC, in addition of testing it will be discussed in 4.B. Moving to the implementation of encrypting the mapped points, in

addition of testing it which will be discussed in.4.C. Finally, the implementation of signing the message, in addition of testing it will be discussed in 4.D.

4.A Implementation and Testing Encoding Plaintext to Numerical Values Step

Encoding of the plaintext presents in converting the given plaintext message into the numerical representation suitable for the operations with the elliptic curves. This step is critical for ECC since its goal is to prepare the data to undergo other encryption stages of the process.

A piece of code for encoding plaintext to numerical values is shown in Figure 34 below:

```
//Function to generate array of BigInt chained blocks
async function bigIntChainedBlocks(blocks) {
  const result = [];
  blocks.forEach((block) => {
    result.push(BigInt(new BN(block, 2).toString(10)));
  });
  return result;
}
```

Figure 34: Implementation of Encoding Plaintext to Numerical Value

Figure 34 shows the conversion from binary values blocks to decimal values, so each block will have an integer x value which is ready to be mapped to elliptic curve point in the further steps. This piece of code is a crucial step in Algorithm 22.

To verify the implementation of encoding plaintext to numerical values, a manually testing to the function by providing a sample plaintext message. The output should show the array of plain text characters, the ASCII values in the corresponding blocks of binary values, chained blocks after XOR operation with IV is done, and the numerical x values, while this step done in Figure 34. This helps ensure that the

encoding process correctly converts the plaintext into a format suitable for ECC operations.

The output of encoding plaintext to numerical values is shown in Figure 35 below:

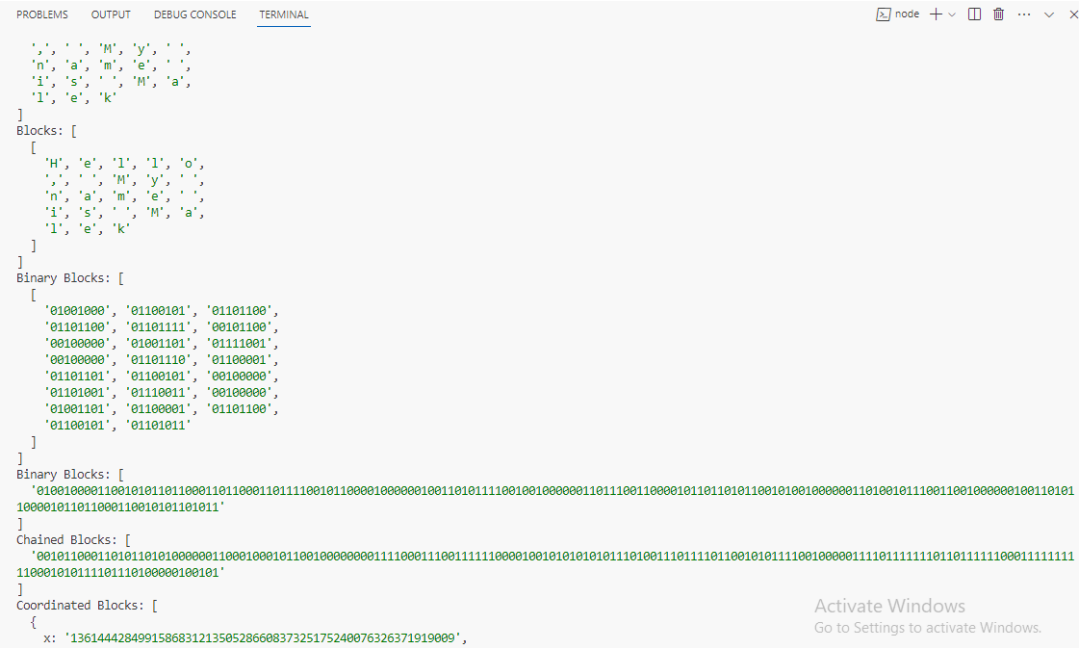


Figure 35: Encoding Plaintext to Numerical Values Testing

From the output, it is clear that the plaintext is converted successfully to numerical values represented in x values. This proves the correctness of the encoding process.

4.B Implementation and Testing Mapping the numerical value to EC Step

Mapping of a numeric value to an elliptic curve point has always been an essential step of ECC encryption. This step involves a numerical value of the plaintext which allows one to get the point (x, y) on the elliptic curve. It makes sure that the numerical values can be applied in elliptical curve operations such that things like encrypting and decrypting can be done.

Below is the implementation code for mapping a numerical value to elliptic curve, which represents Algorithm 12, is shown in Figure 36:

```
374   //Function to caclualte X and Y coordinates for each block
375   export async function generateXandYforBlocks(blocks, a, b, p) {
376     // console.log("Blocks:", blocks);
377     const result = [];
378     const arrayOfCounters = [];
379     blocks.forEach((block) => {
380       result.push(generateXandYPoints(block, a, b, p).coordinates);
381       arrayOfCounters.push(generateXandYPoints(block, a, b, p).counter);
382     });
```

Figure 36: Mapping a Numerical Value to Elliptic Curve Implementation

For the testing a sample numerical value is used to ensure that a properly set up is done for the conversion of numerical values to elliptic curve points. The expected output is the elliptic curve points coordinates (x and y). This assists in verifying that the mapping procedure is competent in the identification of proper points on the elliptic curve.

Below is the output for mapping a numerical value to elliptic curve which shown in Figure 37:

```
Coordinated Blocks: [
  {
    x: '136144428499158683121350528660837325175240076326371919009',
    y: '627710173538668076383578942317605901376719477318284228408O'
  }
```

Figure 37: Mapping a Numerical Value to Elliptic Curve Testing

In the present case, the conversion of numerical x values to the actual elliptic curve points with the (x, y) coordinates is achieved as demonstrated in Figure 37. This verification confirms the correctness of the implementation of mapping a numerical value to elliptic curve points.

4.C Implementation and Testing the Encrypting the Mapped Points

Encrypting the mapped points on the elliptic curve is a key step in ECC encryption. This step uses the ksh derived from the key agreement to transform the points into

ciphertext. The encryption ensures the data is encapsulated and not accessible to unauthorized. In this step, ksh is added to each mapped point to obtain the encrypted mapped points which are defined in Algorithm 13.

Below is the code to encrypt the mapped points shown in Figure 38:

```
429  //Function to apply points addition
430  async function encryptingBlocksPoints(blocks, ksh, p) {
431    const result = [];
432    blocks.forEach((block) => {
433      const point = { x: new BN(block.x), y: new BN(block.y) };
434      const addedPoint = addPoints(point, ksh, p);
435      result.push(addedPoint);
436    });
437    console.log("Encypted Block Points", result);
438    return result;
439  }
```

Figure 38: Encrypting Mapped Point Implementation

A manual test will be done to encrypting the mapped points function by providing the points. The output should show the encrypted mapped points so we can verify that the encryption correctly transforms the mapped points to encrypted mapped points. Below is the output for encrypting the points which is shown in Figure 39:

```
Encypted Block Points [
  {
    x: <BN: 52db5b9e287691d5853794c0876b049d177f8f958ed93fb>,
    y: <BN: 7981ca12479e0c47e3a3a6b98498092f598e941730bb9547>
  }
]
```

Figure 39: Encrypting Mapped Points Testing

Figure 39 shows the encrypted mapped points, which are 2 points, while they are represented in hex value. This verifies that the encryption correctly transforms the mapped points to encrypted mapped points.

4.D Implementation and Testing Signing the Message

Signing the message is an important step in the ECC scheme which guarantee the integrity and authenticity for the transmitted data. Signing the message is discussed in Algorithm 5 , and this process will give a digital signature for the encrypted message represented by object {r, s}.

Below the implementation code for signing the message which is shown in Figure 40:

```
462  //Function to sign encrypted message blocks
463  async function signMessage(points, ds, G, p, a) {
464    const e = hash(points);
465    const z = leftmostBits(e, p.bitLength());
466
467    let k, r, s;
468
469    do {
470      do {
471        // Step 3: Select k
472        k = new BN(crypto.randomBytes(32)).umod(p);
473
474        // Step 4: Obtain r = x mod p where x is (x, y) = k * G
475        const kG = scalarMultiply(k, G, a, p);
476
477        r = kG.x.umod(p);
478      } while (r.isZero()); // Step 5: if r == 0 go to step 3
479
480      // Step 6: Obtain s = (z + ds * r) * k^(-1) mod p
481      let kInverse = modInverse(k, p);
482      s = z.add(ds.mul(r)).mul(kInverse).umod(p);
483    } while (s.isZero());
484
485    // Step 7: The pair (r, s) is the signature
486    console.log("Signed Message:", "r:", r, "s:", s);
487    return { r, s };
488  }
```

Figure 40: Signing the Message Implementation

To verify the implementation of signing the message, a manual testing should be done. The output of this implementation should show the digital signature (r, s).

Below the output for signing the message which is shown in Figure 41:

Figure 41: Signing the Message Testing

The digital signature (r, s) is shown in Figure 41, which proves the validity of signing the message step.

5. Decryption Process Implementation and Testing

Part 5 covers 4 steps. Starting with the implementation of verifying the message, in addition of testing it which will be discussed in 5.1. Then, the implementation of decrypting the cipher mapped points, in addition of testing it will be discussed in 5.2. Moving to the implementation of decoding to hex values, in addition of testing it which will be discussed in 5.3. Finally, the implementation of converting the hex values to plaintext, in addition of testing it will be discussed in 5.4.

5.1 Implementation and Testing of Verifying the Message

Verifying the message is a very important step, which is used to verify the digital signature attached to the message. If the signature is valid, it confirms that the message is authentic.

Below the implementation code for verifying the message (Algorithm 6) which shown in Figure 42:

```
77  async function verifySignature(message, signature, ds, G, p, a) {
78    const { r, s } = signature;
79    // Step 1: Verify (r, s) are integers in [1, p - 1)]
80    if (
81      r.cmp(new BN(1)) === -1 ||
82      r.cmp(p.sub(new BN(1))) === 1 ||
83      s.cmp(new BN(1)) === -1 ||
84      s.cmp(p.sub(new BN(1))) === 1
85    ) {
86      return false;
87    }
88
89    // Step 2: Obtain e = HASH(Msent)
90    const e = hash(message);
91
92    // Step 3: Obtain z = leftmost p bits of e
93    const z = leftmostBits(e, p.bitLength());
94
95    // Step 4: Obtain u1 = es^(-1) mod p
96    const sInverse = modInverse(s, p);
97    const u1 = z.mul(sInverse).umod(p);
98
99    // Step 5: Obtain u2 = rs^(-1) mod p
100   const u2 = r.mul(sInverse).umod(p);
101   const test = z.add(r.mul(ds)).mul(sInverse).umod(p);
102
103   // Step 6: Calculate (x1, y1) = u1 * G + u2 * PUs
104   const x1y1 = scalarMultiply(u1.add(u2.mul(ds)).umod(p), G, a, p);
105
106   // Step 7: Verify r ≡ x1 mod p
107   if (r.cmp(x1y1.x.umod(p)) === 0) {
108     console.log("r:", r);
```

Figure 42: Verifying the Message Implementation

To verify the correctness for the implementation of verifying the message, a manual testing will be done by providing a sample message, its digital signature, and the sender's public key. The output should show the result of the verification process. This helps confirm that the verification process correctly determines whether the message and signature are valid.

Below the output for verifying the message which is shown in Figure 43:

```
r: <BN: 17ecbbcb7af8df634b823a2b83facc79d146dbe8159222c>
Message Verified.
```

Figure 43: Verifying the Message Testing

It shows “Message Verified”, which prove the correctness of verifying the message.

5.2 Implementation and Testing the Decryption of the Cipher Mapped Points

Decrypting the cipher points is the next step in ECC decryption where the cipher points are converted back to their original mapped points by using ksh. This ensures the encrypted data can be correctly decoded and converted back to its original number. This step is done in Algorithm 14, while encrypted mapped points are subtracted from ksh to obtain the decrypted mapped points.

Below is the code for decrypting the cipher mapped points shown in Figure 44:

```
180  //Function to Subtract Cipher Text
181  async function subtractCipherText(inputArray, ksh, p) {
182    const result = [];
183
184    inputArray.forEach((element) => {
185      const point = { x: new BN(element.x), y: new BN(element.y) };
186      const subtractedPoints = subtractPoints(point, ksh, p);
187      result.push({
188        x: subtractedPoints.x,
189        y: subtractedPoints.y,
190      });
191    });
192    console.log("Decrypted Cipher Points:", result);
193
194    return result;
195  }
```

Figure 44: Decrypting the Cipher Mapped Points Implementation

We will manually test the function by providing a sample cipher point and ksh. The output should show the decrypted mapped points so the decryption of cipher mapped points step will be verified by converting the cipher points back to its original form using ksh.

Below is the output for the decryption of the cipher mapped points which is shown in Figure 45:

```
Decrypted Cipher Points: [
  {
    x: <BN: 58d6a0622c8078e7e12aae9decaf20f7f6fc7fe2bdd04a1>,
    y: <BN: ffffffffffffffffffffffff99def836146bc9b1b4d22830>
  }
]
```

Figure 45: Decrypting the Cipher Mapped Points Testing

The decrypted cipher points are shown in Figure 45, and this proves the correctness of the decryption of the cipher points because the transformation of mapped points is done correctly.

5.3 Implementation and Testing the Decoding to Hex Values

Decoding to hex is an important step in ECC decryption, where the decrypted points are converted back to their original numerical (hex) form. This ensures the decrypted data can be converted back to the original plaintext in further step, so each decrypted point represents one hex value in each block. While this step is done Algorithm 23.

Below is the code for decoding to hex which is shown in Figure 46:

```
318  //Function to transform binary blocks to hex blocks
319  async function binaryBlocksToHexBlocks(inputArray) {
320    const result = [];
321    for (let i = 0; i < inputArray.length; i++) {
322      result.push(binaryToHex(inputArray[i]));
323    }
324    console.log("Hex Blocks",result);
325    return result;
326  }
```

Figure 46: Decoding to Hex Values Implementation

A manual test is done for the function by providing a sample decrypted point. The output should show the decrypted point and the hex values so the verification for the decoding is done by converting the decrypted point back to its original numerical form. This step is done in Algorithm 23.

Below is the output for decoding to hex is shown in Figure 47:

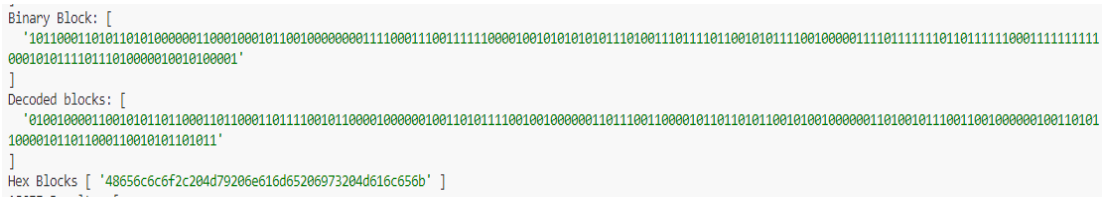


Figure 47: Decoding to Hex Values Testing

Decoded elliptic curve point is in its original hex form. Tested output is correct, since decoded points are back to numerical form.

5.4 Implementation and Testing the Converting of Hexadecimal Values to Plaintext

Converting hex values to plaintext is the last step of the ECC decryption process. This step converts the numerical values (in hexadecimal) back to the original message using ASCII table. This step belongs to Algorithm 24.

Below is the code to convert hexadecimal to plaintext which is shown in Figure 48:

```
328  //Function to transform hex strings to ascii arrays
329  async function hexStringsToAsciiArrays(hexStrings) {
330    const asciiArrays = hexStrings.map((hexString) => {
331      // Ensure the hex string length is even
332      const paddedHex = hexString.length % 2 === 0 ? hexString : "0" + hexString;
333
334      // Convert each pair of hex digits to decimal and then to ASCII character
335      const asciiArray = [];
336      for (let i = 0; i < paddedHex.length; i += 2) {
337        const hexPair = paddedHex.substr(i, 2);
338        const decimalValue = parseInt(hexPair, 16);
339        const asciiChar = String.fromCharCode(decimalValue);
340        asciiArray.push(asciiChar);
341      }
342      return asciiArray;
343    });
344    console.log("ASCII Results:", asciiArrays);
345
346    return asciiArrays;
347  }
```

Figure 48: Converting Hexadecimal Values to Plaintext Implementation

To test the hexadecimal to plaintext function we will manually test it by providing a sample hexadecimal string. The output should show the hex string and the corresponding plaintext message so the verification is done to the conversion which is correct and converts the hex values back to the original message.

Here is the output of the conversion of hexadecimal to plaintext which is shown in Figure 49

```
ASCII Results: [
  [
    'H', 'e', 'l', 'l', 'o',
    ',', ' ', 'M', 'y', ' ',
    'n', 'a', 'm', 'e', ' ',
    'i', 's', ' ', 'M', 'a',
    'l', 'e', 'k'
  ]
]
Final Message: Hello, My name is Malek
```

Figure 49: Converting Hexadecimal Values to Plaintext Testing

The hex is converted back to the plaintext message. The output is correct.

6. ECC Scheme Testing

Here the testing of the encryption and decryption will be done for the whole system. Below the testing for the ECC scheme is shown in Figure 50:

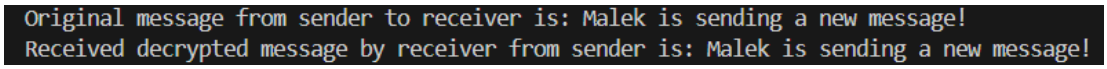


Figure 50: ECC Scheme from [3] Testing

3.2.3 Implementation and Testing Summary

The final and efficient ECC scheme for IoT involves a complete structure for initializing parameters, key agreement, encoding and mapping the plaintext, encrypting and decrypting points and signing and verifying messages. Some already existing and proven libraries as BN.js and bcrypt.js are used to implement the proposed scheme ensuring the security and efficiency of the system. This careful implementation taking into account the special constraints and requirements of the IoT scenario leads to a dependable cryptographic scheme that improves the secure communications and the data integrity for the IoT.

The testing of the ECC system confirms that all stages, from key agreement to decryption and conversion back to plaintext, work correctly, ensuring secure and accurate communication.

3.3 Security Analysis

A security analysis is done for three types of attacks, which are: MITM attack, Chosen Plaintext Attack (CPA), and Chosen Ciphertext Attack (CCA).

3.3.1 Proof for MITM Attack [28]

A Man-in-the-Middle attack is a type of cyber-attack where an attacker intercepts and potentially alters the communication between two parties who believe they are directly communicating with each other. The attacker can eavesdrop on the communication,

steal sensitive information, or inject malicious data without either party being aware of the intrusion. This type of attack exploits the lack of authentication and encryption in the communication channel, making it a significant threat to data security and privacy.

Diffie-Hellman key exchange is exposed by MITM attack, while to solve this problem, trusted public keys should be used. This trusted public keys are used using certificate authority (CA) which is a trusted third party. CA will generate its private key and public key, then the public keys for sender and receiver will be signed and verified using ECDSA in Algorithm 5 and Algorithm respectively.

Trusted public keys are used for Diffie-Hellman key exchange in Algorithm 21.

Blocking the MITM Attack:

Proof by Equations:

Let's denote the attacker as A, sender as S, and recipient as R.

With CA (Blocking MITM)

1. Signing and Verification of both public keys for sender and receiver by authorized party:

PU_s signed by CA_priv (CA private key): sender's signature (σ_s) = Signing Message (PU_s, CA_priv, G, a, b, p) using ECDSA in Algorithm 5, while σ_s (r, s).

PU_r signed by CA_priv: receiver's signature (σ_r) = Sign Message ($PU_{,r}$, CA_priv, G, a, b, p), while σ_r represents (r, s).

2. Exchange and Verification:

S sends (PU_s, σ_s) to R.

R sends (PU_r, σ_r) to S.

Both S and R verify the received signed public keys using CA_pub. Verifying Message (PU_s, σ_s, PU_s signed, G, a, b, p) using ECDSA in Algorithm 6

Verifying Message (PU_r, σ_r, PU_r signed, G, a, b, p)

3. Authentication and Integrity: The attacker tries to intercept and alter the key but in this case the verification step will lead for failing and he/she can't proceed to have the shared secret keys for both sender and receiver:

If A attempts to send PU_a instead of PU_s or PU_r, the verification will fail:

Verifying Message (PU_a, σ_s, d_a, G, a, b, p) will fail because $\sigma_a$was not signed by CA_priv for PU_a.

Key Exchange is secured now form MITM.

Conclusion

By introducing a CA to sign and verify public keys, the authenticity and integrity of the public keys are ensured. This prevents an attacker from substituting the public keys with their own. The equations demonstrate that:

1. Without CA: An attacker can substitute public keys and compute separate keys with each party.

2. With CA: The attacker's substitution attempt fails because the signatures cannot be verified against the CA's public key, blocking the MITM attack.

Thus, using a CA effectively mitigates the risk of a man-in-the-middle attack in the Diffie-Hellman key exchange.

3.3.2 Proof for CPA [24]

Proof of Security of CPA using Random IV (IV: random initial vector)

Proof:

The thesis's ECC scheme is IND-CPA secure, meaning the probability that the adversary can distinguish between ciphertexts of chosen plaintexts is negligible. The proof is based on the fact that the scheme uses a different IV for each encryption. The adversary, A, can submit as many messages as desired to the encryption oracle. For each query, A receives the corresponding ciphertext.

The adversary submits q queries to the challenger to encrypt plaintexts of their choice. The adversary then sends two messages, m_0 and m_1, to the challenger for encryption. The challenger encrypts one of the messages using a randomly chosen bit b, resulting in c = Enc(k, $m_b \oplus$ IV). The adversary's task is to guess the value of b.

Consider the following games:

- Game 0: The adversary requests encryption of a message $m_0$and receives c_0= Enc(k, $m_0 \oplus$ IV).
- Game 1: The adversary sends two messages m_0 and m_1, where m_0 is the same as in Game 0. The challenger encrypts one of them m_b, and sends c_b = Enc(k, $m_b \oplus$ IV) to the adversary.
- Game 3: The adversary attempts to distinguish between $m_0$and m_1 based on the received ciphertexts. The probability of A guessing b is negligible because the ciphertexts in both games are generated with different IVs. In this case, attacker can't guess the bit because the cioher text in each game will be different, and there is no unique ciphertext for each plaintext.

- This ensures that the messages m_0 and m_1 are not distinguishable. The ciphertext from m_0 in Game 0 is different from the ciphertext from m_0 in Game 1 due to the different IVs used in each game. Hence, the adversary's advantage Adv_IND-CPA[A, E] is 0, proving the scheme's CPA security.

3.3.3 Proof for CCA [24]

A Chosen Ciphertext Attack (CCA) is a scenario where an attacker can choose arbitrary ciphertexts and obtain their corresponding plaintexts using a decryption oracle. This allows the attacker to gather information and potentially deduce the encryption key or the encryption scheme used.

Proof of Security with Random IV (IV: random initial vector) Using Equations

Proof:

The proof for Adv_IND-CCA[A, E] starts by noting the adversary's ability to access both the encryption and decryption oracle. The adversary, A, can submit chosen ciphertexts to the decryption oracle and chosen plaintexts to the encryption oracle. The encryption process is defined as:

c = Enc(k, m, IV_i)

The adversary submits two distinct messages, m_0 and m_1, to the encryption oracle and receives the corresponding ciphertexts c_0 and c_1:

c_0 =Enc(k, $m_0 \oplus IV_0$)

c_1 =Enc(k, $m_1 \oplus IV_1$)

In the challenge phase, the adversary chooses m_0 and m_1 again, and the challenger selects a random bit b. The challenger then encrypts m_b and returns the ciphertext c:

c = Enc(k, m_b, IV_i)

To distinguish between m_0 and m_1, the adversary modifies the ciphertext c by XORing it with a random $IV_i$and submits it to the decryption oracle:

$c' = c \oplus IV_i$The decryption oracle decrypts c' to obtain:

Dec(k, c') = Dec(k, c $\oplus$ IV_i)

Since the decryption process relies on the modified IV:

Dec(k, Enc(k, $m_b \oplus IV_b$) $\oplus$ IV_i)

The adversary can't then compare this result with the known plaintexts m_0 and m_1 to determine b because the ciphertext is always changing and no unique ciphertext for every plaintext.

This demonstrates that thesis's ECC scheme is secure under IND-CCA when a random IV is used. Therefore, using a random IV is essential to maintaining the security of the encryption scheme under chosen ciphertext attacks.

Conclusion

The use of random IVs in the encryption process significantly enhances the security of the scheme against chosen ciphertext attacks (CCA). By ensuring that each ciphertext block is unique and indistinguishable, the proposed method provides a robust defense against adversaries attempting to exploit decryption oracles.

Chapter 4

EXPERIMENTAL SETTINGS AND RESULTS

This chapter discusses the experimental results of the ECC scheme designed for IoT and its comparison with the existing experimental results which will be discussed in section 2.3.3. Section 4.1, shows the experimental results obtained for measuring the encryption operation. Section 4.2, shows the experimental results of the iterative mapping of values to points on an elliptic curve. Section 4.3, shows the experimental results of the encoding/decoding operation for three systems. The results are discussed in greater detail and the advantages and efficient aspects of the ECC scheme developed in this thesis for IoT devices are presented compared to an existing well established and proven benchmark presented in 2.3.3.

4.1 Experimental Settings and Results for Experiment 1 for Security

Objective:

The objective of Experiment 1 is to involve encrypting (Algorithm 17 (updated in [5] to have random IV), Algorithm 12 and Algorithm 13) the same plaintext multiple times to observe the effects of using a random initial vector (IV) on the encryption output, where the output without using random IV for the same plaintext and EC parameters will be the same (the same encrypted message is always obtained when the same plain text is encrypted), while by using random IV, if the same plaintext is encrypted, then the encrypted message will be always different. The experiment is conducted in two parts: first, the same plaintext is encrypted twice without using a random initial vector, and this process is repeated twice to confirm consistency. Second, the same plaintext

is encrypted twice using a random initial vector each time, and this process is also repeated twice to observe the variations introduced by the random IV. By comparing the outputs from both parts, we aim to ensure the security of the scheme (ciphertext) against Chosen Ciphertext Attacks (CCA) and Chosen Plaintext Attacks (CPA).

Setup:

Plain Text: Same plain text is used for all scenarios of Experiment 4 in order to ensure consistency in input data and to facilitate consistent measurement of the effects of the encryption mechanism (same plain text used in Experiment 1 in 2.3.3 as well). The plain text is:

"This is a test code of Java application and encryption system. It's not an actual system in real life."

Encryption: Plaintext is encrypted through the ECC curve with and without varying initial vectors (IV) to cover different scenarios of the crypto system.

ECC parameters used are shown in Figure 30.

Experimental Results:

Figure 51: Represents the encrypted points resulting from the first encryption run of above plaintext (ECC parameters are a, b, base point G, and p).

Figure 52: Represents the second encryption run of same plain text with same conditions (same parameters) as Figure 81, to observe any variance.

Figure 53: Represents the cipher text output when encryption run is done with an additional random initial vector (IV).

Figure 54: Represents the cipher text generated by second encryption run using the same plain text with a random value of IV which for sure different compared to Figure 53.

Figure 51: Encrypted Points Using ECC Encryption

Figure 51 shows the encrypted points of the plaintext used (mentioned in Figure 51) and this plaintext is represented in 5 EC points, while random IV isn't used which means if running the encryption again for the same plaintext, it has to give the same encrypted points.

Figure 52: Second Encryption Run for the Same Plaintext and Obtaining Encrypted Points

Figure 52 shows the encrypted points of the plaintext used (mentioned in Figure 52 and it's the same plaintext in Figure 51) and this plaintext is represented in 5 EC points, while random IV isn't used and the encrypted points are exactly the same as the ones in Figure 51. This is expected and proves that without using random IV, the encryption of the same plaintext will lead to the same encrypted points.

Figure 53: Encrypted Points Using Random Initial Vector

Figure 53 shows the encrypted points of the plaintext used (mentioned in Figure 53) and this plaintext is represented in 5 EC points, while random IV is used and the encrypted points are different than the ones in Figure 51 and Figure 52. This is expected because of the using of random IV, but also it's not enough because a new variable is used this time so it's normal to get different encrypted points compared to the ones in Figure 51 and Figure 52, so the expectation to have different encrypted points while running the encryption again using random IV.

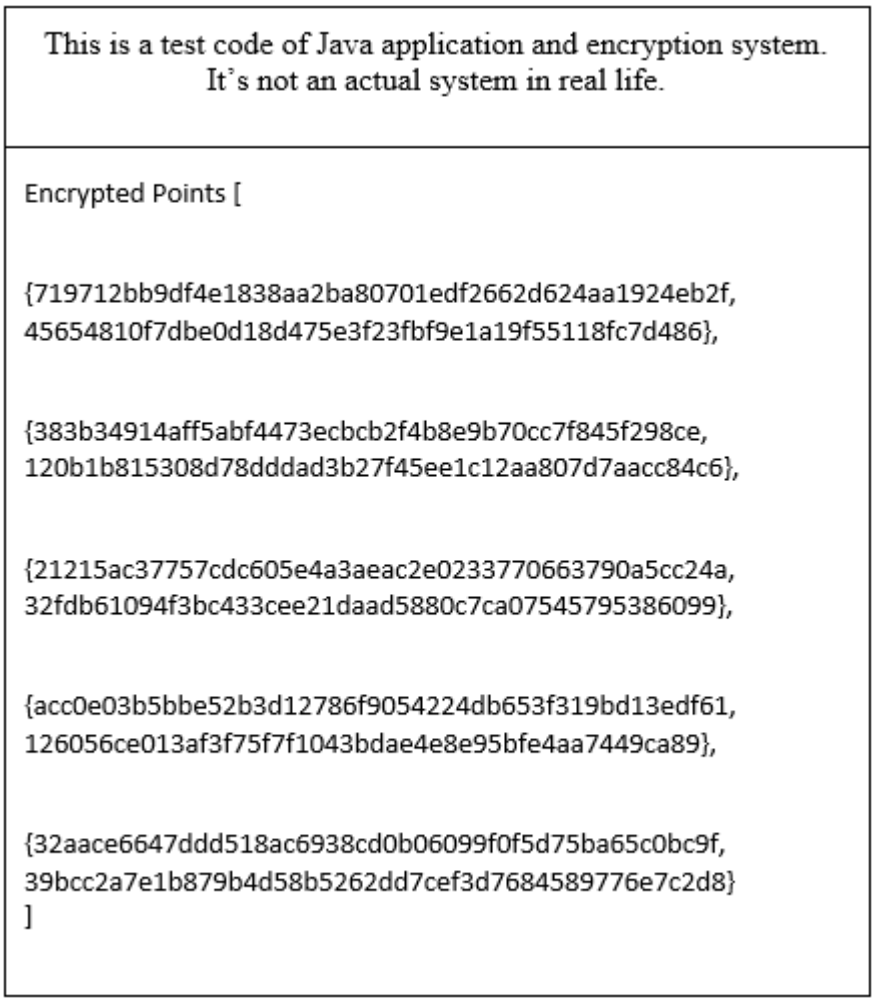


Figure 54: Second Encryption Run for the Same Plaintext Using Random Initial Vector

Figure 54 shows the encrypted points of the plaintext used (mentioned in Figure 54) and this plaintext is represented in 5 EC points, while random IV is used and the encrypted points are different than the ones in Figure 53 as expected. This proves that using random IV leads to different encrypted point for the same plaintext in every run of encryption. . Random IV changes which prevents the opponents from using techniques such as CPA or CCA to expose or guess the secret key. As a result, this scheme is more secure against advanced attacks.

Experiment 1 Results Analysis:

From Experiment 1, it is observed that the random initial vectors (IV) effectively contribute to the security of ECC encryption. The observations from the experiment are summarized below:

Cipher text Variability: Re-encryption of same plain text with same conditions (Figure 51 and Figure 52) produced the same results, which indicates that the ECC is deterministic without IV.

IV provides additional security: When encryption occurs with an initial vector (IV) (Figure 53 and Figure 54), resulted in different cipher text for each instance of encryption. This cipher text variability prevents the attacker to detect any pattern and hence provides security against CPA and CCA.

Comparison with Existing Experimental Results:

Existing Experiment Results: The existing experiments in Experiment 1 in 2.3.3 in Figure 16, Figure 17, Figure 18 and Figure 19 also observed the similar effect of using dynamic initial vectors in preventing the predictable cipher text generation. The existing experiments observed that each instance of encryption should result in different cipher text which in turns prevents the attacker to detect any pattern and hence provides security against CPA and CCA. My Results: My experimental results suggest the same observations. The IV when used in my ECC scheme resulted in non-deterministic encryption output, and hence ascertained the robustness of the additional security measures proposed in the existing experiment. This comparison ensures that my ECC scheme adequately addresses the issues posed by the CPA and CCA attacks, since encrypting the same plaintext with different initial vectors generates different cipher texts.

So my experiment has the same approach compared to Experiment 1 in 2.3.3.

Conclusion of Experiment 1:

The results from my experiment support the use of non-deterministic encryption methods in ECC to avoid vulnerabilities associated with predictable cipher text generation. By dynamically varying initial vectors, my ECC scheme demonstrates enhanced security, making it robust against sophisticated attacks such as CPA and CCA. These findings are consistent with known experimental results, underscoring the importance of implementing dynamic encryption parameters in securing IoT.

4.2 Experimental Settings and Results for Experiment 2 for Mapping Points Successfully

Objective:

To evaluate the successful mapping rate, which mean the number of mapped points over the number of x values (number of needed points to be mapped) ($\frac{Number\ of\ Mapped\ Points}{Total\ Number\ of\ x\ values\ Needed\ to\ be\ MApped}$), for each iteration in Algorithm 12 and number of iterations needed to map all points that are generated to the standard EC curves: secp192k1, secp224k1 and secp256k1 curves where ECC parameters are introduced in Appendix D in Figure 73, Figure 74 and Figure 75, respectively.

Methodology:

Curve 1: secp192k1 in Appendix D which is a recommended elliptic curve over prime field (F_p, where p's size is 192-bit) with 5 padding bits (used in this research's ECC scheme).

Curve 2: secp192k1 in Appendix D which is a recommended elliptic curve over prime field (F_p, where p's size is 192-bit) with 3 padding bits (used in [3]).

Curve 3: secp192k1 in Appendix D which is a recommended elliptic curve over prime field (F_p, where p's size is 192-bit) with 8 padding bits (used in [4]).

Curve 4: secp224k1 in Appendix D which is a recommended elliptic curve over prime field (F_p, where p's size is 224-bit) with 5 padding bits (used in this research's ECC scheme).

Curve 5: secp224k1 in Appendix D which is a recommended elliptic curve over prime field (F_p, where p's size is 224-bit) with 3 padding bits (used in [3]).

Curve 6: secp224k1 in Appendix D which is a recommended elliptic curve over prime field (F_p, where p's size is 224-bit) with 8 padding bits (used in [4]).

Curve 7: secp256k1 in Appendix D which is a recommended elliptic curve over prime field (F_p, where p's size is 256-bit) with 5 padding bits (used in this research's ECC scheme).

Curve 8: secp256k1 in Appendix D which is a recommended elliptic curve over prime field (F_p, where p's size is 256-bit) with 3 padding bits (used in [3]).

Curve 9: secp256k1 in Appendix D which is a recommended elliptic curve over prime field (F_p, where p's size is 256-bit) with 8 padding bits (used in [4]).

Procedure:

Take a random generated number and try to map the number to points on the elliptic curve. If no valid point exists for the value, increase the value by 1 and try again until it is a valid point or the round reaches maximum number of iterations.

Setup:

Number of trials: 1000 random numbers were tested (for each curve).

Size of the Random Numbers: 192-bit random number size of secp192k1, 224-bit random number size of secp224k1 and 256-bit random number size of secp256k1.

Experiment 2 Results:

The result of Experiment 2 done on Curve 1 is displayed in Figure 55 below where x-axis is the round and the number of points mapped is shown as well as the cumulative percentage:

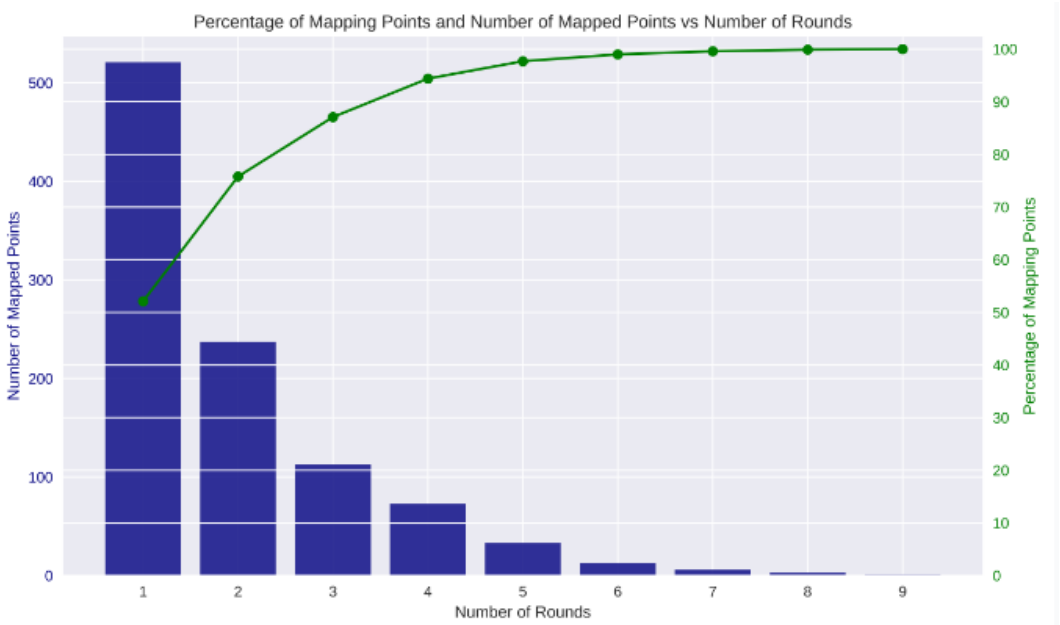


Figure 55: Number of Mapped Points and Percentage of Mapping Points by Round for Curve 1

Figure 55 shows the number of rounds needed to map all the point on Curve 1 and the total number of points mapped after each round, while 9 rounds are taken to map all the points on Curve 1. Round 1 shows around 50% mapped points and it keep going to reach 100% at round 9. Five padding bits are used here (Algorithm 27), which means 32 rounds are accepted at most. So this result is expected and good, because five padding bits can handle this number of rounds. Even though the number of rounds are less than 32 in the result in Figure 55, that doesn't guarantee that five padding bits are enough because maybe other random numbers give more than 32 rounds.

The result of Experiment 2 done on Curve 2 is displayed in Figure 56 below where x-axis is the round and the number of points mapped is shown as well as the cumulative percentage:

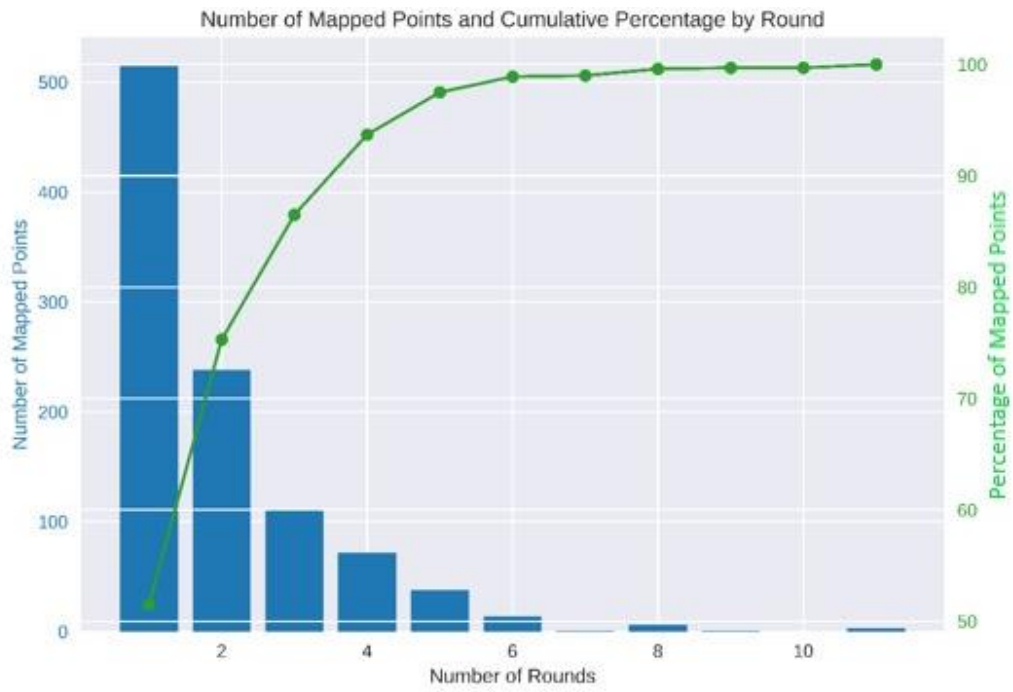


Figure 56: Number of Mapped Points and Percentage of Mapping Points by Round for Curve 2

Figure 56 shows the number of rounds needed to map all the point on Curve 2 and the total number of points mapped after each round, while 11 rounds are taken to map all the points on Curve 2. Round 1 shows around 50% mapped points and it keep going to reach 100% at round 11. Three padding bits are used here (Algorithm 11), which means 8 rounds are accepted at most. So this result isn't as expected and bad, because three padding bits can't handle this number of rounds. This result proves that three padding bits aren't enough and can't be used in [3].

The result of Experiment 2 done on Curve 3 is displayed in Figure 57 below where x-axis is the round and the number of points mapped is shown as well as the cumulative percentage:

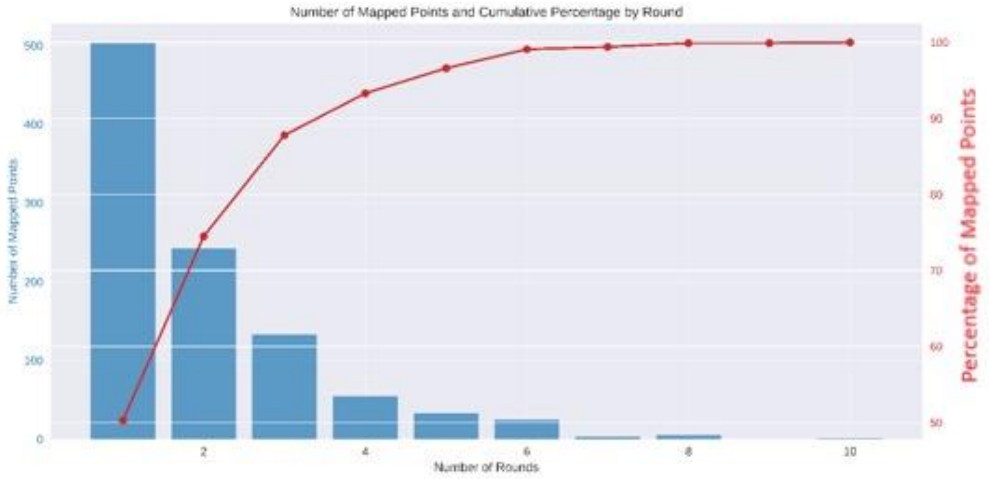


Figure 57: Number of Mapped Points and Cumulative Percentage by Round for Curve 3

Figure 57 shows the number of rounds needed to map all the point on Curve 3 and the total number of points mapped after each round, while 10 rounds are taken to map all the points on Curve 3. Round 1 shows around 50% mapped points and it keep going to reach 100% at round 10. Eight padding bits are used here (Algorithm 3), which means 256 rounds are accepted at most. So this result is expected and good, because eight padding bits can handle this number of rounds. Even though the number of rounds are less than 256 in the result in Figure 57, that doesn't guarantee that eight padding bits are enough because maybe other random numbers give more than 256 rounds.

The result of Experiment 2 done on Curve 4 is displayed in Figure 58 below where x-axis is the round and the number of points mapped is shown as well as the cumulative percentage:

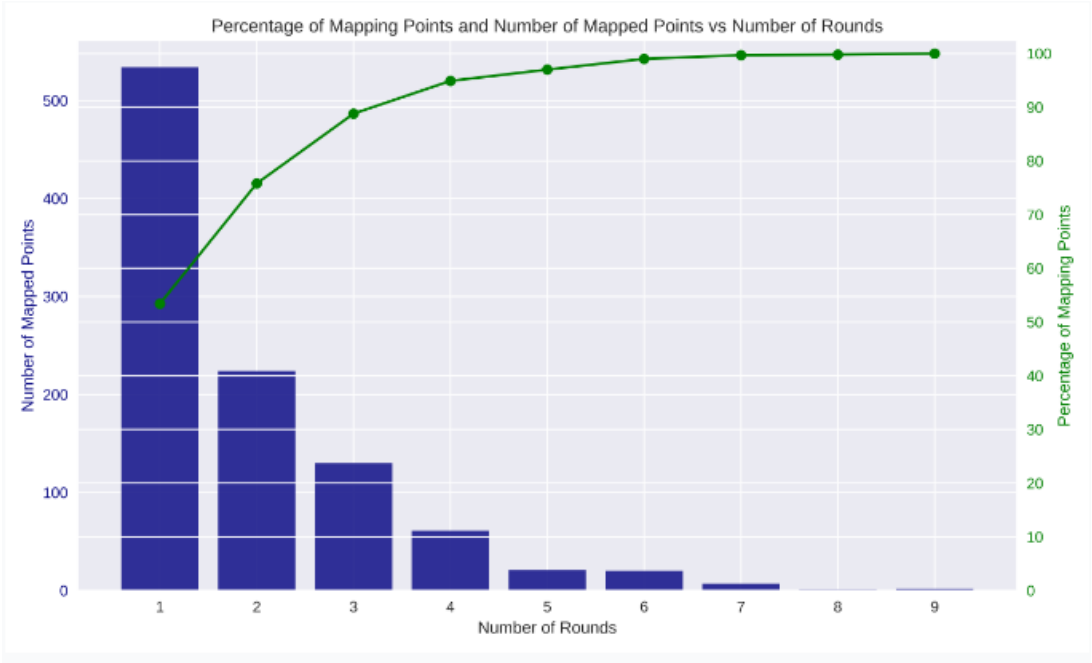


Figure 58: Number of Mapped Points and Percentage of Mapping Points by Round for Curve 4

Figure 58 shows the number of rounds needed to map all the point on Curve 4 and the total number of points mapped after each round, while 9 rounds are taken to map all the points on Curve 4. Round 1 shows around 50% mapped points and it keep going to reach 100% at round 9. Five padding bits are used here (Algorithm 27), which means 32 rounds are accepted at most. So this result is expected and good, because five padding bits can handle this number of rounds. Even though the number of rounds are less than 32 in the result in Figure 58, that doesn't guarantee that five padding bits are enough because maybe other random numbers give more than 32 rounds.

The result of Experiment 2 done on Curve 5 is displayed in Figure 59 below where x-axis is the round and the number of points mapped is shown as well as the cumulative percentage:

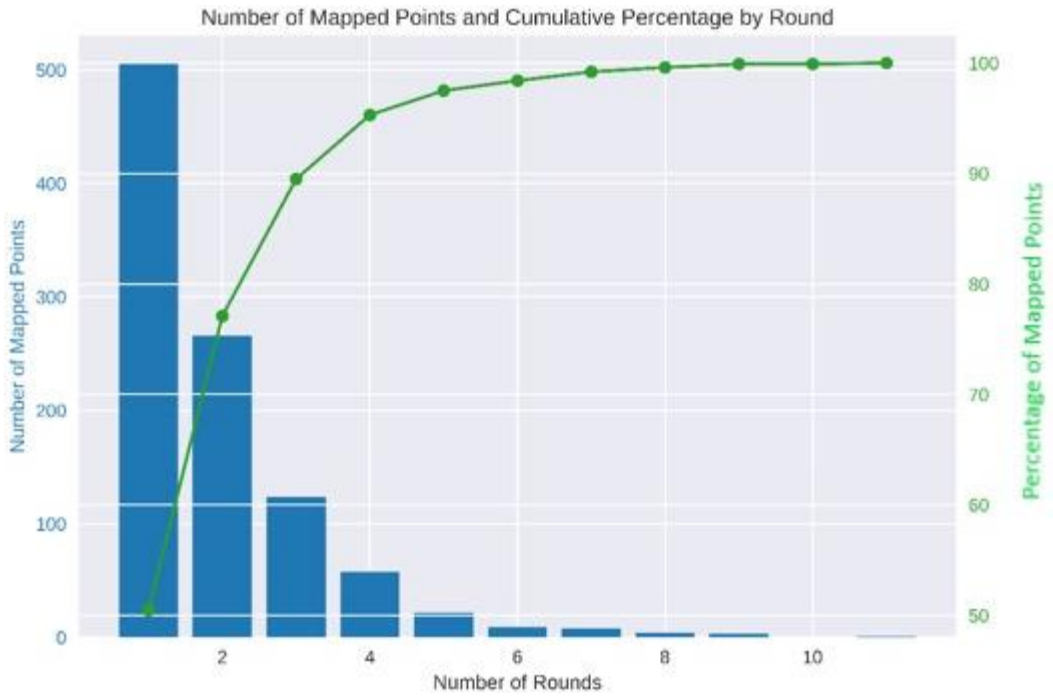


Figure 59: Number of Mapped Points and Percentage of Mapping Points by Round for Curve 5

Figure 59 shows the number of rounds needed to map all the point on Curve 5 and the total number of points mapped after each round, while 11 rounds are taken to map all the points on Curve 5. Round 1 shows around 50% mapped points and it keep going to reach 100% at round 11. Three padding bits are used here (Algorithm 11), which means 8 rounds are accepted at most. So this result isn't as expected and bad, because three padding bits can't handle this number of rounds. This result proves that three padding bits aren't enough and can't be used in [3].

The result of Experiment 2 done on Curve 6 is displayed in Figure 60 below where x-axis is the round and the number of points mapped is shown as well as the cumulative percentage:

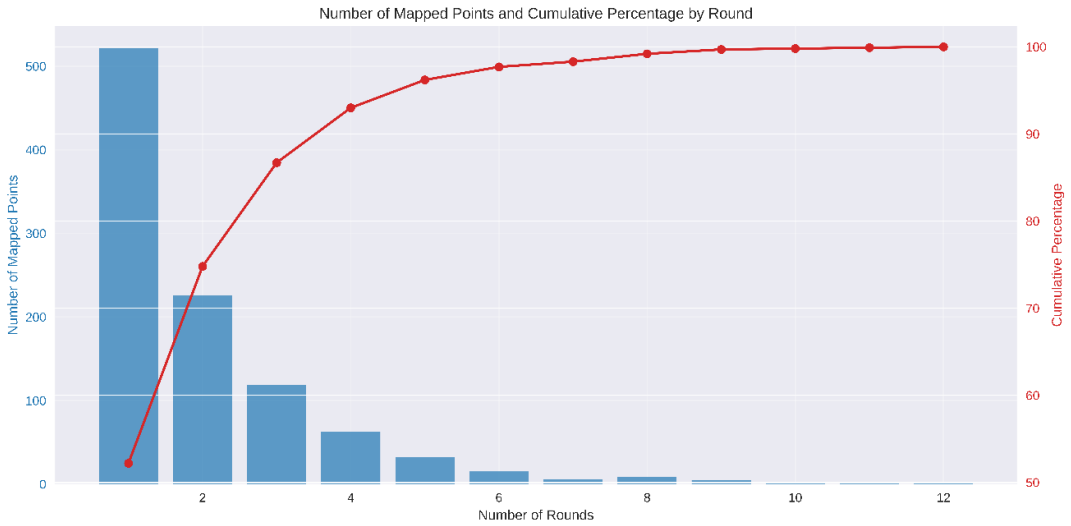


Figure 60: Number of Mapped Points and Percentage of Mapping Points by Round for Curve 6

Figure 60 shows the number of rounds needed to map all the point on Curve 6 and the total number of points mapped after each round, while 12 rounds are taken to map all the points on Curve 6. Round 1 shows around 50% mapped points and it keep going to reach 100% at round 12. Eight padding bits are used here (Algorithm 3), which means 256 rounds are accepted at most. So this result is expected and good, because eight padding bits can handle this number of rounds. Even though the number of rounds are less than 256 in the result in Figure 60, that doesn't guarantee that eight padding bits are enough because maybe other random numbers give more than 256 rounds.

The result of Experiment 2 done on Curve 7 is displayed in Figure 61 below where x-axis is the round and the number of points mapped is shown as well as the cumulative percentage:

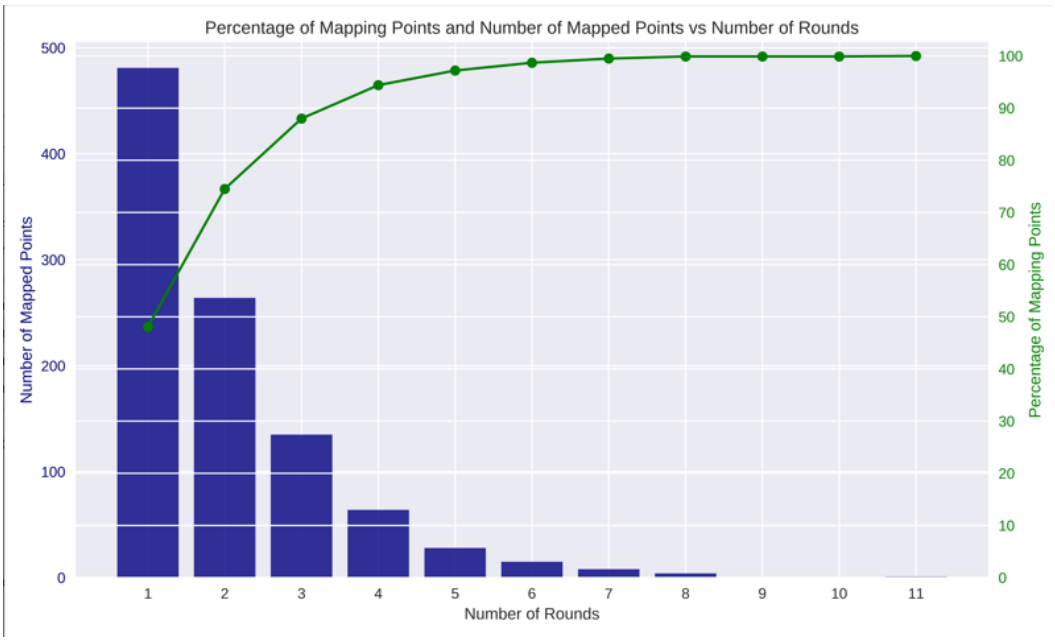


Figure 61: Number of Mapped Points and Percentage of Mapping Points by Round for Curve 7

Figure 61 shows the number of rounds needed to map all the point on Curve 7 and the total number of points mapped after each round, while 11 rounds are taken to map all the points on Curve 7. Round 1 shows around 50% mapped points and it keep going to reach 100% at round 11. Five padding bits are used here (Algorithm 27), which means 32 rounds are accepted at most. So this result is expected and good, because five padding bits can handle this number of rounds. Even though the number of rounds are less than 32 in the result in Figure 61, that doesn't guarantee that five padding bits are enough because maybe other random numbers give more than 32 rounds.

The result of Experiment 2 done on Curve 8 is displayed in Figure 62 below where x-axis is the round and the number of points mapped is shown as well as the cumulative percentage:

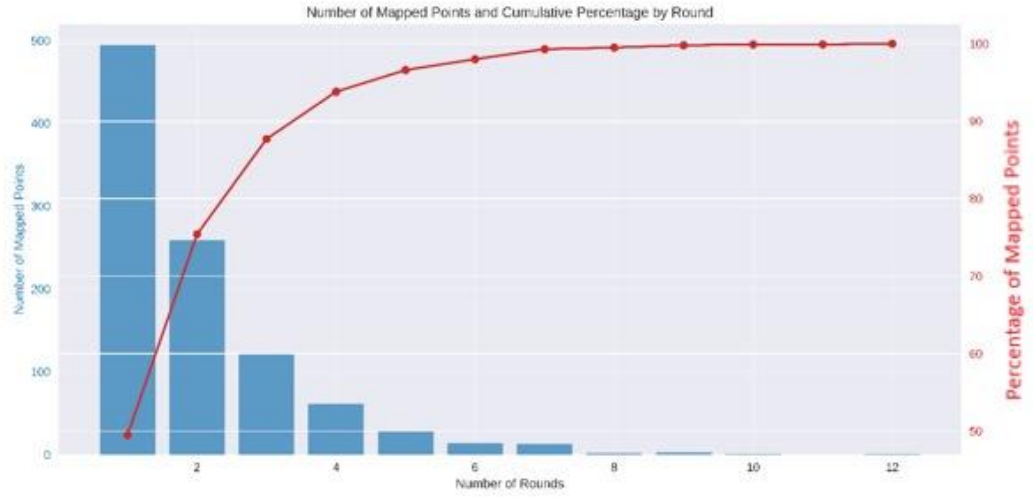


Figure 62: Number of Mapped Points and Percentage of Mapping Points by Round for Curve 8

Figure 62 shows the number of rounds needed to map all the point on Curve 8 and the total number of points mapped after each round, while 12 rounds are taken to map all the points on Curve 8. Round 1 shows around 50% mapped points and it keep going to reach 100% at round 12. Three padding bits are used here (Algorithm 11), which means 8 rounds are accepted at most. So this result isn't as expected and bad, because three padding bits can't handle this number of rounds. This result proves that three padding bits aren't enough and can't be used in [3].

The result of Experiment 2 done on Curve 9 is displayed in Figure 63 below where x-axis is the round and the number of points mapped is shown as well as the cumulative percentage:

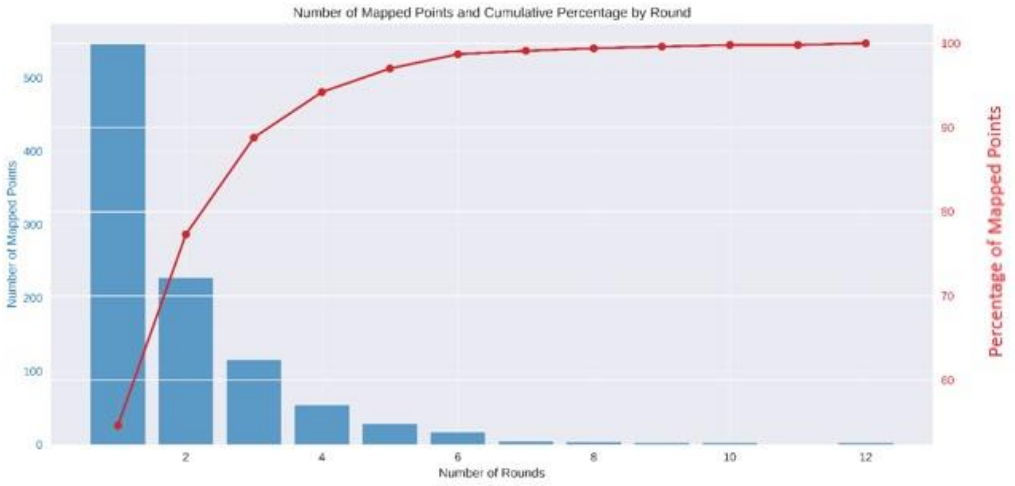


Figure 63: Number of Mapped Points and Percentage of Mapping Points by Round for Curve 9

Figure 63 shows the number of rounds needed to map all the point on Curve 9 and the total number of points mapped after each round, while 12 rounds are taken to map all the points on Curve 9. Round 1 shows around 50% mapped points and it keep going to reach 100% at round 12. Eight padding bits are used here (Algorithm 3), which means 256 rounds are accepted at most. So this result is expected and good, because eight padding bits can handle this number of rounds. Even though the number of rounds are less than 256 in the result in Figure 63, that doesn't guarantee that eight padding bits are enough because maybe other random numbers give more than 256 rounds.

Experiment 2 Results Analysis:

All the Curves (Curve 1 till Curve 9) in Figure 80 till Figure 88 mapped approximately 50% of the total number of points in the first round, but not all the curves have the same number of rounds. Also, high successful mapping rate appears in the beginning rounds (rounds 1, round 2 and round 3) for all the curves.

Curve 1, Curve 4 and Curve 7, where they are related to 5 padding bits in this research's ECC scheme, are represented in Figure 55, Figure 58 and Figure 61

respectively, show 8 rounds for Curve 1, 9 rounds for Curve 4 and 11 rounds for Curve 7 to map all the points on the EC.

Curve 2, Curve 5 and Curve 8, where they are related to 3 padding bits in [3], are represented in Figure 56, Figure 59 and Figure 62 respectively, show 11 rounds for Curve 2 and Curve 5, and 12 rounds for Curve 8 to map all the point on the EC.

Curve 3, Curve 6 and Curve 9, where they are related to 8 padding bits in [4], are represented in Figure 57, Figure 60 and Figure 63 respectively, show 10 rounds for Curve 3 and 12 rounds for Curve 6 and Curve 9, to map all points on the EC.

Conclusion for Experiment 2:

Experiment 2 shows, in Figure 80 till Figure 88, that the number of rounds at most appears to be 12 for [3] and [4] and 11 for this research's ECC scheme, which are small numbers to prove that high efficiency of mapping for the 3 schemes (this research's ECC scheme, [4] and [3]), however 12 rounds aren't covered by the padded 3 bits (max 8 rounds are accepted for 3 bits) used in [3] leading to wrong decryption output. Based on that, 3 padding bits may lead for wrong decryption so using it will cause problems. On the other hand, the results of Experiment 3 accept using 5 or 8 padding bits, but it can't guarantee using them because maybe some values will lead for higher rounds. 5 padding bits accepts 32 rounds at most, while 8 padding bits accepts 256 rounds at most, so both 32 rounds and 256 rounds are more than the double of the highest number of rounds obtained, so using either 5 or 8 padding bits won't cause any problem based on Experiment 2 but this result isn't guaranteed.

4.3 Experimental Settings and Results for Experiment 3 for Testing the Performance (Time) of the Encryption of this Research's ECC Scheme

Objective:

The purpose of Experiment 3 is to compute the encryption time for Thesis's ECC Scheme and compare it to the one in Sengupta system [4].

Setup:

Tested Message size: The message size used are: 1 KB, 10 KB, 100 KB, 1 MB and 10 MB, where random generator messages of the mentioned size where used.

Tools and Environment: 11th Gen Intel(R) Core (TM) i7-1165G7 @ 2.80 GHz (4-core, each core hyper-threaded) and 12 RAM is used to compute the performance of the encryption in this research's ECC scheme.

Experiment 3 Results:

Experiment 3 results are represented by Figure 64 below:

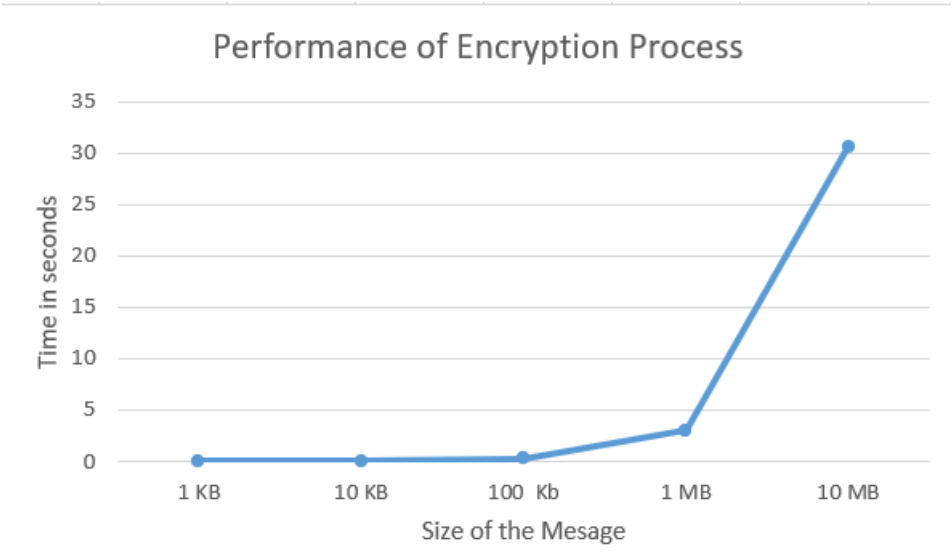


Figure 64: Performance of the Encryption Process in the Research's ECC Scheme

The x-axis represents the size of the message and the y-axis represents time needed in seconds to encrypt the message.

Figure 64 shows the time needed to encrypt a message having specific size. The time needed to encrypt 1 KB message, 10 KB, and 100 KB message is less than 1 second, while for 1 MB message is about 3 seconds, and for 10 MB message is 31 seconds. The result in Figure 64 is as expected and it is good.

Analysis of the Experiment 3 Results:

The research's encryption process takes around 31 seconds to encrypt the message of size 10 MB, which is slightly less compared to the encryption process in [4], which takes 27 seconds.

Conclusion for Experiment 3:

The results from Experiment 3 confirm in Figure 64 that this thesis's ECC scheme (which is based on [3]) has good efficiency which is almost the same as Sengupta [4] which its main purpose to improve the efficiency. The gap between the research's scheme and [4] is 4 seconds which may vary depending on the messages used, because mapping time varies from1 input to another and this small-time difference may reach 4 seconds or more while the message size is increasing.

4.4 Experimental Settings and Results for Experiment 4 for Performance (Time) of Mapping Points by Using Different Number of Padding Bits

Objective:

The purpose of Experiment 4 is to check whether using 3 padding bits, 5 padding bits and 8 padding bits will affect the performance (time) for running the point searching function used in Figure 72.

Setup:

Message Size: 1000 random number are used for each trial. The size of each random number used is 187 bits for 3 padding bits, 189 bits for 5 padding bits and 192 bits for 8 padding bits. Although all of them are dealing with 192 bis, but while saying padding 3 bits to 184 bits (23 characters * 8 bits = 184), it means the 187 bits will be mixture between 0 and 1 and the last 5 bits on left for sure will be zero. The same way is appeared for 5 padding bits and 8 padding bits, so this is done to have a fair comparison.

Device used: The same device mentioned in 4.3 is used to compute all performances (time)

Curve 1 (5 padding bits used), Curve 2 (3 padding bits used), Curve 3 (8 padding bits used), Curve 4 (5 padding bits used), Curve 5 (3 padding bits used), Curve 6 (8 padding bits used), Curve 7 (5 padding bits used), Curve 8 (3 padding bits used) and Curve 9 (8 padding bits used), which are mentioned in 4.1, are used to be tested.

Experiment 4 Results:

Figure 65 shows the performance for each curve, where group 1 includes: Curve 2, Curve 1, and Curve 3 (in this order), group 2 includes: Curve 5, Curve 4, and Curve 6 (in this order) and group 3 includes: Curve 8, Curve 7, and Curve 9 (in this order). Figure 65 is given below:

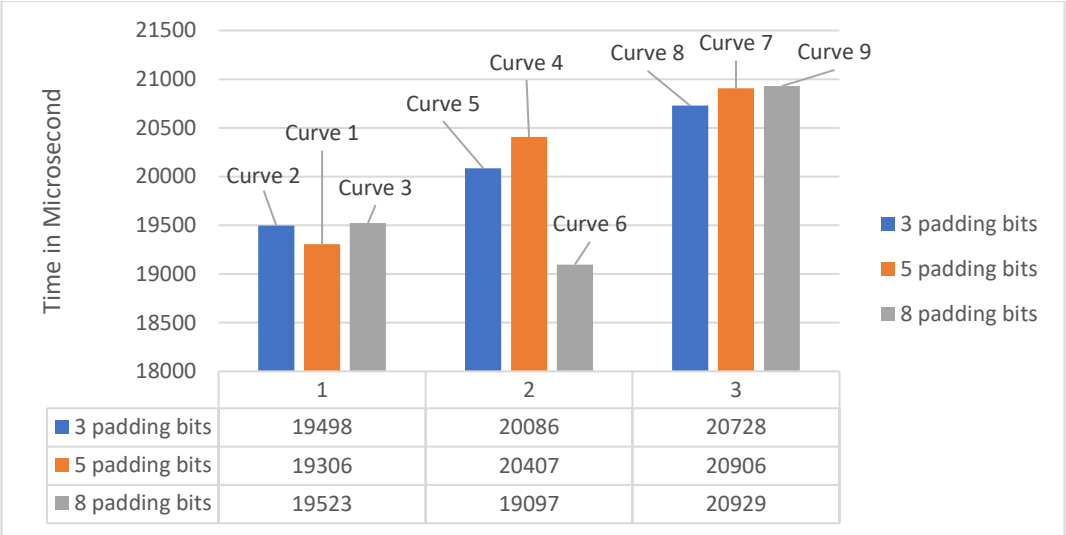


Figure 65: Performance (Time) Testing Results

Figure 65 shows in group 1, the performance of 5 padding bits is the highest with 19306 microseconds. In group 2, the performance of 8 padding bits is the highest with 19097 microseconds, while in group 3, the performance of 3 padding bits is the highest with 20728 microseconds. And in all curves, slight difference performance appears. So based on the results in Figure 65, the number of padding bits won't affect the performance which is good because it let us to select 8 padding bits which has lower risks (256 rounds are allowed) compared to 3 padding bits and 5 padding.

Conclusion of Experiment 4:

The results of Experiment 4 in Figure 65 shows that the size of padding bits doesn't affect the performance (time of running), which means the best option is to select 8 padding bits which allow the highest number of rounds, and this option won't affect the performance of the ECC scheme.

4.5 Summary for the Experimental Results

All the addressed problems in 2.4 are discussed and showed in the Experiments' conclusions and a summary for the results is shown in Table 2.

Table 2 summarize all the results for the 4 schemes [3], [4], [5], and my scheme:

Table 2: Results Summary

Scheme	AE Scheme Blocking CCA and CPA	Rounds Needed to Map All Points Successfully Based on Experiment 2 in Section 4.2	Encryption Process Performance (Time) for 10 MB message Size in section 2.3.3	Best Size of Padding Bits to be Selected in Section 4.4
SE-Enc [3], [5]	No – Yes	12	NA	No
Sengupta	No	12	27 seconds	Yes
Thesis ECC Scheme	Yes	11	31 seconds	No

This table compares the features and performance of three cryptographic schemes based on security analysis in 3.3 and experimental results (4.1 – 4.4): SE-Enc, Sengupta, and the Thesis ECC Scheme. The SE-Enc scheme, as referenced in [3] and [5], blocks Chosen Ciphertext Attacks (CCA) and Chosen Plaintext Attacks (CPA) in [5] only, requiring 12 rounds to map all points. However, its data on encryption process performance for a 10 MB message is not available, and it does not optimize the best size of padding bits. The Sengupta scheme does not block CCA and CPA attacks but requires 12 rounds to map all points and takes 27 seconds to encrypt a 10 MB message. It is optimized for the best size of padding bits. The Thesis ECC Scheme fully blocks CCA and CPA attacks, needing only 11 rounds to map all points. It takes 31 seconds to encrypt a 10 MB message but is not optimized for the best size of padding bits.

Chapter 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

This thesis presents the design, implementation, testing, and evaluation of an efficient and secure ECC system for the IoT building upon the SE-Enc system [3]. While SE-Enc has been effective in prior research, it has notable security deficiencies, particularly concerning CPA, CCA, and MITM attacks.

To address these issues, the ECC scheme in this thesis introduces several key modifications. Firstly, a random IV is used instead of a constant IV to block CPA and CCA. Secondly, trusted public keys are incorporated into the Diffie-Hellman key exchange protocol to mitigate MITM attacks. Lastly, the padding bits used in the mapping function have been increased from three to five to ensure correct encryption and decryption, as three bits are insufficient for mapping points effectively.

The security of the proposed ECC scheme is analyzed in 3.3, demonstrating its robustness against MITM, CCA, and CPA attacks. Experiment 2 results in 4.2 confirms the insufficiency of using only three padding bits on the least significant bit (LSB), highlighting the need for more padding bits like five or eight bits. The encryption time achieved by thesis's ECC scheme is considered as good as that of Sengupta's scheme [4], which is supported by the results of Experiment 4.3, which demonstrate that the encryption times of both schemes are nearly identical. Overall,

the experimental results and security analysis validate that the modified ECC scheme provides a secure and efficient solution suitable for IoT applications.

5.2 Future Work

There are many interesting areas that should be work on to enhance the developed ECC scheme for IoT in the future. This includes investigating and developing advanced key management protocols that can handle frequent changes in IoT networks. Next, testing the ECC scheme in real IoT scenarios like edge computation and further optimize it for practical usages. Also, exploring the integration of our ECC scheme with other security protocols and frameworks to provide a multi-layered security solution for IoT systems. Finally, it is important to further enhance the resilience.

REFERENCES

[1] S. Soumyalatha and G. Hegde, "Study of IoT: Understanding IoT Architecture, Applications, Issues and Challenges," presented at the International Conference on Innovations in Computing & Networking (ICICN16), Bangalore, India, May 2016.

[2] A. Ali, M. A. Rashid, and M. A. Rahman, "Security in Internet of Things: Issues, Challenges, and Solutions," in Internet of Things and Connected Technologies (ICIoTCT 2019), Lecture Notes in Networks and Systems, vol. 38, R. Ghosh, Ed. Cham: Springer, 2019, pp. 583-589. doi: 10.1007/978-3-319-99007-1_38.

[3] H. N. Almogren and A. S. Almogren, "SE-Enc: A Secure and Efficient Encoding Scheme Using Elliptic Curve Cryptography," IEEE Access, vol. 7, pp. 175865-175878, 2019. doi: 10.1109/ACCESS.2019.2957943.

[4] A. Sengupta and U. K. Ray, ‘‘Message mapping and reverse mappingin elliptic curve cryptosystem,’’ Secur. Commun. Netw., vol. 9, no. 18, pp. 5363–5375, 2016.

[5] Hisham AlMajed and Ahmad AlMogren, "A Secure and Efficient ECC-Based Scheme for Edge Computing and Internet of Things," Sensors, vol. 20, no. 6158, 2020, doi:10.3390/s20216158.

[6] L. D. Singh and K. M. Singh, ‘‘Image encryption using elliptic curve cryptography,’’ Procedia Comput. Sci., vol. 54, pp. 472–481, Aug. 2015.

[7] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications," IEEE Communications Surveys and Tutorials, vol. 17, no. 4, pp. 2347-2376, Fourthquarter 2015. DOI: 10.1109/COMST.2015.2444095.

[8] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, "Middleware for Internet of Things: A Survey," IEEE Internet of Things Journal, vol. 3, no. 1, pp. 70-95, Feb. 2016, doi: 10.1109/JIOT.2015.2479195.

[9] M. Soori, B. Arezoo, and R. Dastres, "Internet of things for smart factories in industry 4.0, a review," IoT and CPS Security, vol. 4, no. 6, Apr. 2023. doi: 10.1016/j.iotcps.2023.04.006

[10] Novikov, S. V., and A. A. Sazonov. “Application of the Open Operating System ‘MindSphere’ in Digital Transformation of High-Tech Enterprises”. Economics Journal, vol. 1, no. 1, Dec. 2019, pp. 20-26, doi:10.46502/issn.2711-2454/2019.1.03.

[11] P. Rajak, A. Ganguly, S. Adhikary, and S. Bhattacharya, "Internet of Things and smart sensors in agriculture: Scopes and challenges," Journal of Agriculture and Food Research, vol. 14, p. 100776, Dec. 2023. doi: 10.1016/j.jafr.2023.100776

[12] R. Aiyshwariya Devi and A. R. Arunachalam, "Enhancement of IoT device security using an Improved Elliptic Curve Cryptography algorithm and malware detection utilizing deep LSTM," *High-Confidence Computing*, vol. 3,

no. 2, p. 100117, June 2023. Available: https://doi.org/10.1016/j.hcc.2023.100117.

[13] S. K. Singh, "An Efficient and Secure Protocol for Ensuring Data Storage Security in Cloud Computing Using ECC," *International Journal of Advanced Research in Computer and Communication Engineering (IJARCCE)*, Jan. 2016, doi: 10.17148/IJARCCE.2016.5730

[14] M. Bafandehkar, S. M. Yasin, R. Mahmod, and Z. M. Hanapi, "Comparison of ECC and RSA Algorithm in Resource Constrained Devices," in *2013 International Conference on IT Convergence and Security (ICITCS)*, Macao, China, 2013, pp. 1-5. doi: 10.1109/ICITCS.2013.6717816.

[15] Q. Chang, Y. Zhang, and L. Qin, "A Node Authentication Protocol based on ECC in WSN", International Conference on Computer Design and Applications (ICCDA), 2010, vol. 2, pp. 606-609.

[16] J. Zhang, J. Cui, H. Zhong, Z. Chen, and L. Liu, "PA-CRT: Chinese Remainder Theorem Based Conditional Privacy-Preserving Authentication Scheme in Vehicular Ad-Hoc Networks," IEEE Transactions on Dependable and Secure Computing, vol. 18, no. 2, pp. 722-735, Mar.-Apr. 2021. doi: 10.1109/TDSC.2019.2904274.

[17] I. Blake, G. Seroussi, and N. Smart, Elliptic Curves in Cryptography. Cambridge: Cambridge University Press, 1999, doi: 10.1017/CBO9781107360211.

[18] D. Phiamphu and P. Saha, "Redesigned the Architecture of Extended-Euclidean Algorithm for Modular Multiplicative Inverse and Jacobi Symbol," in Proceedings of the 2018 2nd International Conference on Trends in Electronics and Informatics (ICOEI), Tirunelveli, India, 2018, pp. 722-726. doi: 10.1109/ICOEI.2018.8553922.

[19] H. Darrel, V. Scott, M, Alfred (2004). Guide to Elliptic Curve Cryptography. Springer Professional Computing. New York: Springer-Verlag. doi:10.1007/b97644. ISBN 0-387-95273-X. S2CID 720546.

[20] Corbellini, A. "Elliptic Curve Calculator," Andrea Corbellini, [Online]. Available: https://andrea.corbellini.name/ecc/interactive/modk-mul.html.

[21] Johnson, D., Menezes, A., and Vanstone, S., "The Elliptic Curve Digital Signature Algorithm (ECDSA)," International Journal of Information Security, vol. 1, no. 1, pp. 36-63, Aug. 2001. doi: 10.1007/s102070100002.

[22] Software signing: National Institute of Standards and Technology (NIST), "Digital Signature Standard (DSS)," FIPS PUB 186-4, Jul. 2013. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

[23] Qing Chang, "A Node Authentication Protocol based on ECC in WSN", International Conference On Computer Design And Applications (ICCDA), 2010, vol. 2, pp. 606-609, doi: 10.1109/ICCDA.2010.5541288.

[24] D. Boneh, and V. Shoup, "A Graduate Course in Applied Cryptography," Draft Version 0.5, pp. 343-345, 2020. Available: https://toc.cryptobook.us/

[25] J. Hoffstein, J. Pipher, and J. H. Silverman, *An Introduction to Mathematical Cryptography*. New York, NY, USA: Springer, 2008.

[26] B. Padma, D. Chandravathi, R. P. Prapoorna, “Encoding and decoding of a message in the implementation of elliptic curve cryptography using Koblitz’s method”. International Journal on Computer Science and Engineering 2010; 2(5): 1904–1907.

[27] P. Barman and B. Saha, DNA Encoded Elliptic Curve Cryptography System for IoT Security (March 19, 2019). International Journal of Computational Intelligence & IoT, Vol. 2, No. 2, 2019, Available at SSRN: https://ssrn.com/abstract=3355530.

[28] W. Stallings, "Cryptography and Network Security: Principles and Practice," 7th ed., Pearson, 2017.

APPENDECIES

Appendix A: Experiments Raw Data Outputs and Source Code

1. Experiment 1 Raw Data Outputs

Figure 66 shows Experiment 1 outputs:

Figure 66: Experiment 1 Data Outputs

2. Experiment 2 Raw Data Outputs:

Figure 67, Figure 68, and Figure 69 show the raw data outputs for Experiment 2:

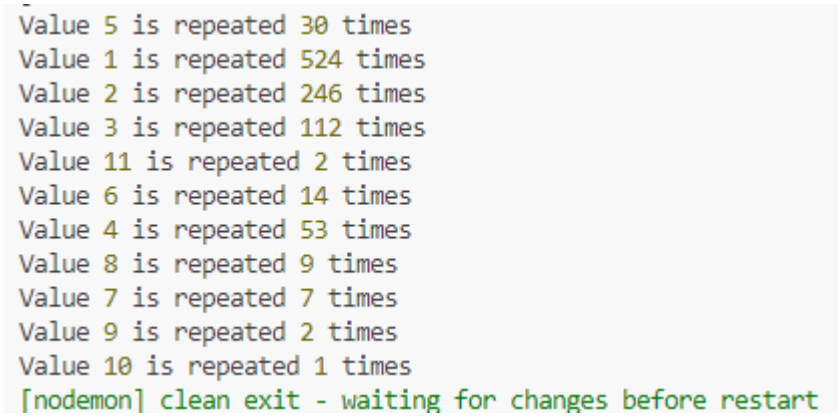


Figure 67: Experiment 2 Output for Curve 1

```
Value 3 is repeated 126 times
Value 1 is repeated 513 times
Value 2 is repeated 249 times
Value 5 is repeated 27 times
Value 4 is repeated 60 times
Value 6 is repeated 11 times
Value 7 is repeated 8 times
Value 9 is repeated 2 times
Value 8 is repeated 3 times
Value 12 is repeated 1 times
```

Figure 68: Experiment 2 Result for Curve 2

```
Value 3 is repeated 145 times
Value 2 is repeated 237 times
Value 1 is repeated 492 times
Value 14 is repeated 1 times
Value 6 is repeated 12 times
Value 4 is repeated 68 times
Value 8 is repeated 6 times
Value 5 is repeated 30 times
Value 7 is repeated 4 times
Value 11 is repeated 1 times
Value 9 is repeated 3 times
Value 10 is repeated 1 times
```

Figure 69: Experiment 2 Results for Curve 3

3. Experiment 3 Raw Data Outputs

Figure 70 show the raw data output of Experiment 3:

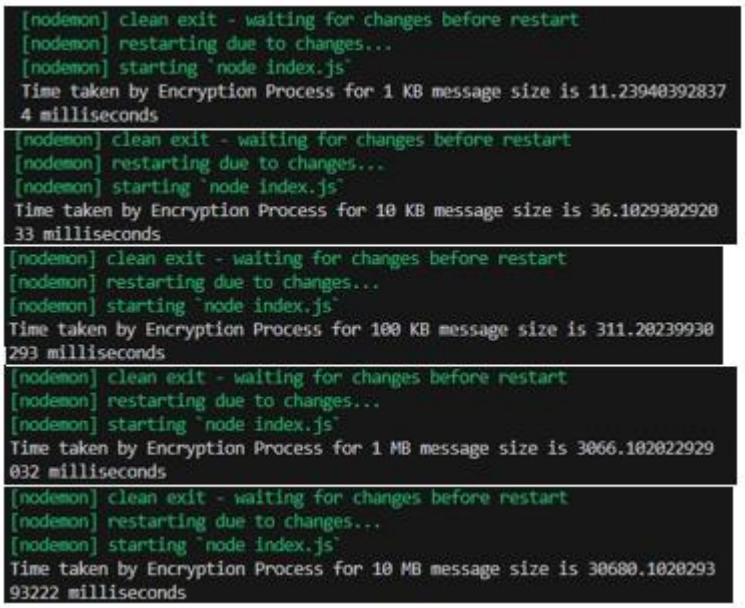


Figure 70: Raw Data Output of Experiment 3

4. Experiment 4 Raw Data Outputs

Figure 71 shows the raw data output of Experiment 4:

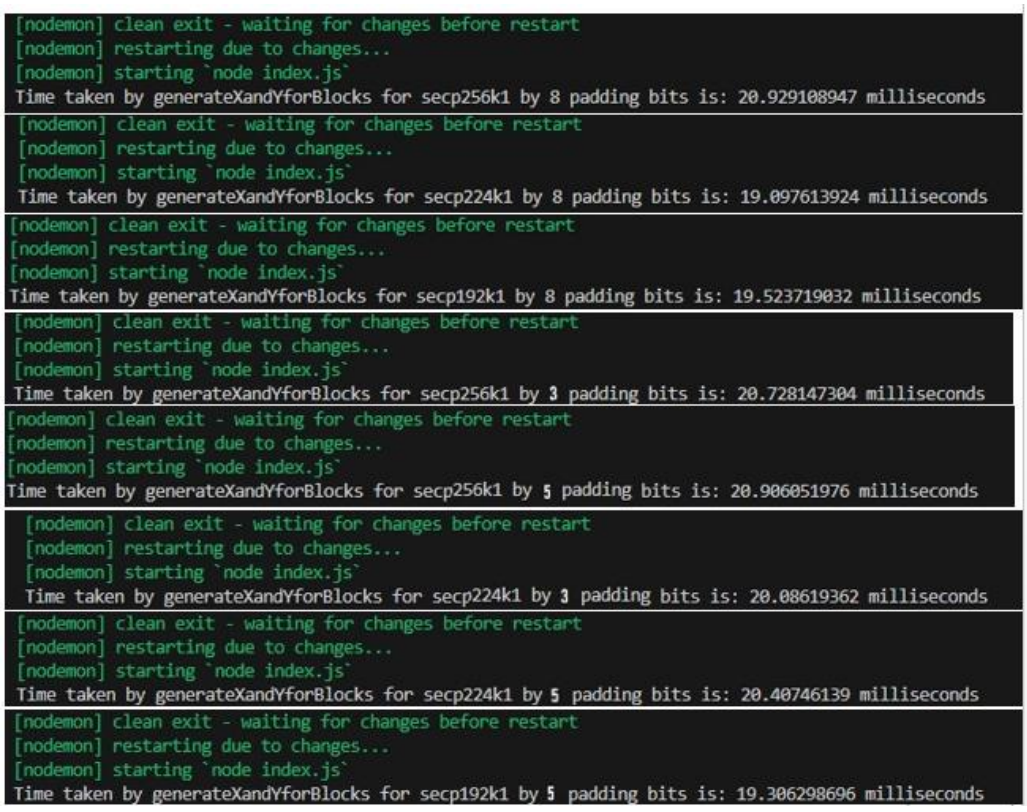


Figure 71: Raw Data Outputs of Experiment 4

5. ECC Scheme Source Code

ECC scheme source code is provided in the following link: https://drive.google.com/drive/folders/1dbeYRvGt1lfxA9IFyiUEj1x2JUoFayUd?usp=sharing

Appendix B: Calculations for Example 14 – Example 16

1. Calculations of Q_A = 6 × (5, 1) in Step 2 of Algorithm 7 in Example 14

Steps for Scalar Multiplication to compute Q_A = 6 × (5, 1) on curve (16) by using algorithm 2:

Step 1: Compute 2 × G = (x_{2G}, y_{2G}) = 2 × (5, 1) using (12) for point doubling

1. Calculate λ using (12):

$$\lambda = \frac{3x_G^2 + a}{2y_G} \; mod \; 17$$

x_G= 5, y_G= 1, a = 2

$$\lambda = \frac{3 \times 5^2 + 2}{2 \times 1} \; mod \; 17 = \frac{3 \times 25 + 2}{2} \; mod \; 17 = \frac{75 + 2}{2} \; mod \; 17$$

$$= \frac{77}{2} \; mod \; 17$$

To compute $\frac{77}{2} \; mod \; 17$, we need the modular inverse of 2 mod 17, which is 9, calculated using the Extended Euclidean Algorithm [17], which is explained in Algorithm 1, and similar calculations are done in Example 5.

λ= 77 * 9 mod 17 = 693 mod 17 = = 693 - 40 * 17 = 693 - 680 = 13

2. Calculate x_{2G} on curve (16) using (12):

$$x_{2G} = \lambda^2 - 2x_G \; mod \; 17 = 13^2 - 2 * 5 \; mod \; 17 = 169 - 10 \; mod \; 17$$

$$= 159 \; mod \; 17$$

x_{2G} =159 - 9 * 17 = 159 - 153 = 6

3. Calculate y_{2G} on curve (16) using (12):

$$y_{2G} = \lambda(x_G - x_{2G}) - y_G \; mod \; 17 = 13(5 - 6) - 1 \; mod \; 17$$

$$= 13 * (-1) - 1 \; mod \; 17$$

$$y_{2G} = -13 - 1 \; mod \; 17 = -14 \bmod 17 = 3$$

So, 2 × G = (6, 3).

Step 2: Compute $4 \times G = (x_{4G}, y_{4G})$ = 2 × 2×G using (12) for point doubling

1. Calculate λ using (12):

$$\lambda = \frac{3x_{2G}^2 + a}{2y_{2G}} \ mod\ 17$$

$x_{2G=6},\ y_{2G=3}$

$$\lambda = \frac{3 * 6^2 + 2}{2 * 3} \ mod\ 17 = \frac{3 * 36 + 2}{6} \ mod\ 17 = \frac{110}{6} \ mod\ 17$$

To compute $\frac{110}{6} \ mod\ 17$, we need the modular inverse of 6 mod 17, which is 3, and it is calculated using the Extended Euclidean Algorithm [17], which is explained in Algorithm 1, and similar calculations are done in Example 5.

Thus, λ = 110 * 3 mod 17= 330 mod 17 = 7

2. Calculate x_{4G} on curve (16) using (12):

$$x_{4G} = \lambda^2 - 2x_{2G} \ mod\ 17 = 7^2 - 2 * 6 \ mod\ 17 = 49 - 12 \ mod\ 17$$

$$= 37 \ mod\ 17$$

x_{4G} =37 - 2 * 17 = 37 - 34 = 3

3. Calculate y_{4G} on curve (16) using (12):

$$y_{4G} = \lambda\left(x_{2G\ -\ x_{4G}}\right) - y_{2G} \ mod\ 17 = 7(6 - 3) - 3 \ mod\ 17 = 7 * 3 - 3 \ mod\ 17$$

$$y_{4G} = 21 - 3 \ mod\ 17 = 18 \ mod\ 17 = 1$$

So, $4 \times G$ = (3, 1).

Step 3: Compute $6G \times = (x_{6G}, y_{6G})$ = 4×G + 2×G using (11) for point addition (12)

1. Calculate λ using (11):

$$\lambda = \frac{y_{2G} - y_{4G}}{x_{2G} - x_{4G}} \ mod\ 17$$

$4 \times G$ = (3, 1) and $2 \times G$ = (6, 3)

$$\lambda = \frac{3 - 1}{6 - 3} \ mod\ 17 = \frac{2}{3} \ mod\ 17$$

To compute $\frac{2}{3}\ mod\ 17$, we need the modular inverse of 3 mod 17, which is 6, and it is calculated using the Extended Euclidean Algorithm [17], which is explained in Algorithm 1, and similar calculations are done in Example 5.

Therefore, λ = 2 * 6 mod 17 = 12 mod 17= 12

2. Calculate x_{6G} on curve (16) using (11):

$x_{6G} = \lambda^2 - x_{4G} - x_{2G}\ mod\ 17 = 12^2 - 3 - 6\ mod\ 17 = 144 - 9\ mod\ 17$

$x_{6G} = 135\ mod\ 17 = 135 - 7 \times 17 = 135 - 119 = 16$

3. Calculate y_{6G} on curve (16) using (11):

$y_{6G} = \lambda(x_{4G} - x_{6G}) - y_{4G}\ mod\ 17 = 12(3 - 16) - 1\ mod\ 17$

$= 12(-13) - 1\ mod\ 17$

$y_{6G} = -156 - 1\ mod\ 17 = -157\ mod\ 17 = 13$

So, $6 \times G$ = (16, 13).

2. Calculations of k×Q_A= 2 × (16, 13) in Step 3 of Algorithm 8 in Example 15

Steps for Scalar Multiplication for k×Q_A= 2 × (16, 13) on curve (16) using Algorithm 2:

Step 1: Compute 2 × (16, 13) on curve (16) using (12)

$Q_A = (x_{Q_A}, y_{Q_A})$ = (16, 13)

$k \times Q_A = (x_{kQ_A}, y_{kQ_A})$ = 2 * (16, 13)

1. Calculate λ using (12):

$$\lambda = \frac{3x_{QA}^2 + a}{2y_{QA}}\ mod\ 17$$

x_{QA} = 16, y_{QA} = 13, a = 2

$$\lambda = \frac{3 * 16^2 + 2}{2 * 13}\ mod\ 17 = \frac{3 * 256 + 2}{26}\ mod\ 17 = \frac{768 + 2}{26}\ mod\ 17$$

$$= \frac{770}{26}\ mod\ 17$$

To compute $\frac{770}{26}\ mod\ 17$, we need the modular inverse of 26 mod 17, which is 2, calculated using the Extended Euclidean Algorithm [17], which is explained in Algorithm 1, and similar calculations are done in Example 5.

λ = 770 * 2 mod 17 = 1540 mod 17 = 1540 - 90 * 17 = 1540 - 1530 = 10

2. Calculate x_{kQ_A} on curve (16) using (12):

$$x_{kQ_A} = \lambda^2 - 2x_{Q_A}\ mod\ 17 = 10^2 - 2 \times 16\ mod\ 17 = 100 - 32\ mod\ 17$$

$$= 68\ mod\ 17$$

$$x_{kQ_A} = 68 - 4 * 17 = 68 - 68 = 0$$

3. Calculate y_{kQ_A} on curve (16) using (12):

$$y_{kQ_A} = \lambda(x_{QA} - x_{kQ_A}) - y_{Q_A}\ mod\ 17 = 10(16 - 0) - 13\ mod\ 17$$

$$y_{kQ_A} = 160 - 13\ mod\ 17 = 147\ mod\ 17 = 147 - 8 * 17 = 147 - 136 = 11$$

So, $k \times Q_A$= (0, 11).

3. Calculations of $M + k \times Q_A$= (9, 1) + (0, 11) in Step 4 of Algorithm 8 in Example 15

Steps of Point Addition for $M + k \times Q_A$= (9, 1) + (0, 11) on curve (16) using (11)

Step: Compute M + $k \times Q_A$ using (11)

M = (x_M, y_M) = (9, 1)

$k \times Q_A$ = (x_{kQ_A}, y_{kQ_A}) = (0, 11)

R = (x_R, y_R) = M + $k \times Q_A$

1. Calculate λ using (11):

$$\lambda = \frac{y_{kQ_A} - y_M}{x_{kQ_A} - x_M}\ mod\ 17$$

P = (9, 1) and $k \times Q_A$ = (0, 11)

$\lambda = \frac{11-1}{0-9}\ mod\ 17 = \frac{10}{-9}\ mod\ 17 = \frac{10}{8}\ mod\ 17$, (-9 mod 17 = 8 mod 17)

To compute $\frac{10}{8}\ mod\ 17$, the modular inverse of 8 mod 17 should be computed, which is 15, , calculated using the Extended Euclidean Algorithm [17], which is explained in Algorithm 1, and similar calculations are done in Example 5.

Thus, λ = 10 * 15 mod 17 = 150 mod 17 = 150 - 8 * 17 = 150 - 136 = 14

2. Calculate x_R on curve (16) using (11):

$$x_R = \lambda^2 - x_M - x_{kQ_A}\ mod\ 17 = 14^2 - 9 - 0\ mod\ 17 = 196 - 9\ mod\ 17$$

$$= 187\ mod\ 17$$

$$x_R = 187\ -\ 11\ *\ 17\ =\ 187\ -\ 187\ =\ 0$$

3. Calculate y_R on curve (16) using (11)

$$y_R =\ \lambda(x_M\ -\ x_R) - y_M\ mod\ 17 =\ 14(9-0) - 1\ mod\ 17 =\ 14(9) - 1\ mod\ 17$$

$$y_R =\ 126 - 1\ mod\ 17 =\ 125\ mod\ 17 =\ 125\ -\ 7\ *\ 17\ =\ 125\ -\ 119\ =\ 6$$

So, R = (9, 1) + (0, 11) = (0, 6).

4. Calculations of $d_A \times P$ = 6× (6, 3) in Step 1 by Algorithm 9 in Example 16

Steps of Scalar Multiplication of $d_A \times P$ = 6× (6, 3) on curve (16) using Algorithm 2.

P = (x_P, y_P) = (6, 3)

Step 1: Compute $2 \times P = (x_{2P}, y_{2P})$ = P + P = (2 × P) on curve (16) using (12) (Point Doubling)

1. Calculate λ using (12):

$$\lambda = \frac{3x_P^2\ +a}{2y_P}\ mod\ 17$$

x_P= 6, y_P= 3, a = 2

$$\lambda = \frac{3 * 6^2\ +2}{2 * 3}\ mod\ 17 =\ \frac{3 * 36\ +2}{6}\ mod\ 17 =\ \frac{108\ +2}{2}\ mod\ 17 =\ \frac{110}{2}\ mod\ 17$$

To compute $\frac{110}{2}\ mod\ 17$, we need the modular inverse of 6 mod 17, which is 3, calculated using the Extended Euclidean Algorithm [17], which is explained in Algorithm 1, and similar calculations are done in Example 5.

Then, λ = 110 * 3 mod 17 = 330 mod 17 = 330 - 19 * 17 = 330 - 323 = 7

2. Calculate x_{2P} on curve (16) using (12):

$x_{2P} = \lambda^2 - 2x_P \ mod\ 17 = 7^2 - 2 * 6 \ mod\ 17 = 49 - 12 \ mod\ 17 \ = 37 \ mod\ 17$

x_{2P} = 37 - 2 * 17 = 37 - 34 = 3

3. Calculate y_{2P} on curve (16) using (12):

$y_{2P} = \ \lambda\left(x_{P\ -\ x_{2P}}\right) - y_P \ mod\ 17 = \ 7(6 - 3) - 3 \ mod\ 17 = \ 7 * (3) - 3 \ mod\ 17$

$y_{2P} = \ 21 - 3 \ mod\ 17 = \ 18 \bmod 17 \ = \ 1$

So, $2 \times P$ = (3, 1).

Step 2: Compute $4 \times P = (x_{4P}, y_{4P}) = 2 \times P + 2 \times P$ $(2 \times 2 \times P)$ on curve (16) using (12) (Point Doubling)

1. Calculate λ using (12):

$$\lambda \ = \ \frac{3x_{2P}^2 + a}{2y_{2P}} \ mod\ 17$$

$x_{2P} = 3, \ y_{2P} = 1$

$$\lambda = \frac{3 * 3^2 \ + 2}{2 * 1} \ mod\ 17 = \frac{3 * 9 + 2}{2} \ mod\ 17 = \frac{29}{2} \ mod\ 17$$

To compute $\frac{29}{2} \ mod\ 17$, we need the modular inverse of 2 mod 17, which is 9, calculated using the Extended Euclidean Algorithm [17], which is explained in Algorithm 1, and similar calculations are done in Example 5.

Thus, λ = 29 * 9 mod 17 = 261 mod 17 = 261 - 15 * 17 = 261 - 255 = 6

2. Calculate x_{4P} on curve (16) using (12):

$x_{4P} = \lambda^2 - 2x_{2P} \ mod\ 17 = 6^2 - 2 * 3 \ \ mod\ 17 \ = 36 - 6 \ mod\ 17 = 30 \ \ mod\ 17$

$= 13$

3. Calculate y_{4P} on curve (16) using (12):

$y_{4P} = \ \lambda\left(x_{2P\ -\ x_{4P}}\right) - y_{2P} \ mod\ 17 = 6(3 - 13) - 1 \ mod\ 17$

$= \ 6 * (-10) - 1 \ mod\ 17$

$y_{4P} = -61 \, mod \, 17 = -61 + 4 * 17 = -61 + 68 = 7$

So, $4 \times P$ = (13, 7).

Step 3: Compute $6 \times P = (x_{6P}, y_{6P}) = 4 \times P + 2 \times P$ on curve (16) using (11) (Point Addition)

1. Calculate λ using (11):

$$\lambda = \frac{y_{2P} - y_{4P}}{x_{2P} - x_{4P}} \, mod \, 17$$

$4 \times P$ = (13, 7) and $2 \times P$ = (3, 1)

$$\lambda = \frac{1 - 7}{3 - 13} \, mod \, 17 = \frac{6}{10} \, mod \, 17$$

To compute $\frac{6}{10} \, mod \, 17$, we need the modular inverse of 10 mod 17, which is 12, calculated using the Extended Euclidean Algorithm [17], which is explained in Algorithm 1, and similar calculations are done in Example 5.

Thus, λ = 6 * 12 mod 17= 72 mod 17 = 72 - 4 * 17 = 72 - 68 = 4

2. Calculate x_{6P} using (11):

$$x_{6P} = \lambda^2 - x_{4P} - x_{2P} \, mod \, 17 = 4^2 - 13 - 3 \, mod \, 17 = 16 - 16 \, mod \, 17$$

$$= 0 \, mod \, 17 = 0$$

3. Calculate y_{6P} using (11):

$$y_{6P} = \lambda(x_{4P} - x_{6P}) - y_{4P} \, mod \, 17 = 4(13 - 0) - 7 \, mod \, 17 = 52 - 7 \, mod \, 17$$

$$y_{6P} = 45 \, mod \, 17 = 11$$

So, $6 \times P$ = (0, 11).

5. Calculations of C $- \, d_A \times$ P in Step 2 by Algorithm 9 in Example 16

Steps for Point Subtraction of C - d_A × P using (7)

Step 1: Find the negation of (0, 11)

The negation of a point (x, y) on an elliptic curve is (x, -y mod p). Negation of (0, 11):

11 mod 17 = 6

so, the negation of (0, 11) is (0, 6).

Step 2: Compute (0, 6) - (0, 11) on curve (16) using (12)

(0, 6) - (0, 11) = (0, 6) + (0, 6).

Since both points are identical, we need to use the point doubling in (12) to compute $2 \times (0, 6)$.

Let $2 \times P = (x_{2P}, y_{2P})$.

Step 3: Point Doubling Calculation using (12)

For point P = (x_P, y_P) = (0, 6) on the elliptic curve (16), the point doubling steps are:

1. Compute λ using (12):

$$\lambda = \frac{3x_P^2 + a}{2y_P} \; mod \; 17$$

x_P= 0, y_P= 6, a = 2

$$\lambda = \frac{3 \times 0^2 + 2}{2 \times 6} \; mod \; 17 = \frac{2}{12} \; mod \; 17$$

To compute $\frac{2}{12} \; mod \; 17$, we need the modular inverse of 12 mod 17, which is 10, calculated using the Extended Euclidean Algorithm [17], which is explained in Algorithm 1, and similar calculations are done in Example 5.

Thus, λ = 2 * 10 mod 17 = 20 mod 17 = 20 - 17 = 3

2. Calculate x_{2P} on curve (16) using (12):

$$x_{2P} = \lambda^2 - 2x_P \; mod \; 17 = 3^2 - 2 * 0 \; mod \; 17 = 9 \; mod \; 17 = 9$$

1. Calculate y_{2P} on curve (16) using (12):

$$y_{2P} = \lambda(x_P - x_{2P}) - y_P \ mod\ 17 = 3(0-9) - 6\ mod\ 17$$

$$= 3 * (-9) - 6\ mod\ 17$$

$$y_{2P} = -27 - 6\ mod\ 17 = -33 \bmod 17 = 1$$

Result:

So, $2 \times P = (0, 6) + (0, 6) = (9, 1)$.

The point subtraction (0, 6) - (0, 11) on curve (16) results in the point (9, 1).

6. Elliptic Curve Y-Coordinate Search

Figure 72 shows the code which is used for searching for y coordinate for particular x value, and returning a point on EC:

```
288  //Function to generate X & Y points for each block
289  export function generateXandYPoints(xOriginal, a, b, p) {
290    // Convert input strings to BigInt
291    let x = xOriginal;
292    let counter = 0;
293    a = a;
294    b = b;
295    p = p;
296
297    const pMinusOneOverTwo = (p - 1n) / 2n;
298    const pPlusOneOverFour = (p + 1n) / 4n;
299    while (true) {
300      counter += 1;
301      // Calculate symbol for the current x
302      let ySquared = (modPow(x, 3n, p) + ((a * x) % p) + b) % p;
303      let symbol = modPow(ySquared, pMinusOneOverTwo, p);
304
305      // Check if symbol matches the target
306      if (symbol === 1n) {
307        // Calculate both possible y values using the square root of y^2 (mod p)
308        let y1 = modPow(ySquared, pPlusOneOverFour, p);
309        let y2 = p - y1; // Second solution, as y1 and y2 are reflections across the x-axis
310
311        // Print the higher value of y
312        let higherY = y1 > y2 ? y1 : y2;
313        // console.log(`For x = ${xOriginal}, y = ${higherY}`);
314        return {
315          coordinates: { x: x.toString(), y: higherY.toString() },
316          counter: counter,
317        };
318      }
319
```

Figure 72: EC Point Searching Code

Appendix C: Conversions of Data (Binary, Decimal, and Hexadecimal)

1. Conversion of Data from Decimal to Binary:

The pseudo code for decimal to binary conversion is represented in Algorithm 25:

Algorithm 25: Decimal to Binary Conversion Algorithm

Converting Decimal to Binary Algorithm
function decimal_to_binary(decimal_number): 1. if decimal_number == 0: return "0" 2. binary_string = "" 3. while decimal_number > 0: 3.1 remainder = decimal_number % 2 3.2 binary_string = str(remainder) + binary_string 3.3 decimal_number = decimal_number // 2 4. return binary_string

2. Conversion of Data from Binary to Decimal

The pseudo code for binary to decimal conversion is represented in Algorithm 26:

Algorithm 26: Binary to Decimal Conversion Algorithm

Converting Binary to Decimal Algorithm
function binary_to_decimal(binary_string): 1. decimal_number = 0 2. length = len(binary_string) 3. for i from 0 to length-1: 3.1 bit = int(binary_string[length - 1 - i]) 3.2 decimal_number += bit * (2 ** i) 4. return decimal_number

3. Conversion of Data from Hexadecimal to Binary

The pseudo code for hexadecimal to binary conversion is represented in Algorithm 27:

Algorithm 27: Hexadecimal to Binary Conversion Algorithm

Converting Hexadecimal to Binary Algorithm

```
function hex_to_binary(hexadecimal_string):
    // Step 1: Define a map from hexadecimal digits to binary strings
    hex_to_bin_map = {
        '0': '0000', '1': '0001', '2': '0010', '3': '0011',
        '4': '0100', '5': '0101', '6': '0110', '7': '0111',
        '8': '1000', '9': '1001', 'A': '1010', 'B': '1011',
        'C': '1100', 'D': '1101', 'E': '1110', 'F': '1111'
    }

    // Step 2: Initialize an empty string to store the binary representation
    binary_string = ""

    // Step 3: Loop through each digit in the hexadecimal string
    for digit in hexadecimal_string:
        // Step 4: Convert the digit to its uppercase form
        upper_digit = digit.upper() // digit.upper() converts the lowercase letter to upper
// case letter

        // Step 5: Append the corresponding binary string from the map to the binary
// string
        binary_string += hex_to_bin_map[upper_digit]

    // Step 6: Return the binary string
    return binary_string
```

4. Conversion of Data from Binary to Hexadecimal

The pseudo code for binary to hexadecimal conversion is represented in Algorithm 28:

Algorithm 28: Binary to Hexadecimal Conversion Algorithm

Converting Binary to Hexadecimal Algorithm

```
function hex_to_binary(hexadecimal_string):
    // Step 1: Define a map from hexadecimal digits to binary strings
    hex_to_bin_map = {
        '0': '0000', '1': '0001', '2': '0010', '3': '0011',
        '4': '0100', '5': '0101', '6': '0110', '7': '0111',
        '8': '1000', '9': '1001', 'A': '1010', 'B': '1011',
        'C': '1100', 'D': '1101', 'E': '1110', 'F': '1111'
    }

    // Step 2: Initialize an empty string to store the binary representation
    binary_string = ""

    // Step 3: Loop through each digit in the hexadecimal string
    for digit in hexadecimal_string:
        // Step 4: Convert the digit to its uppercase form
        upper_digit = digit.upper() // digit.upper() converts the lowercase letter to upper case letter

        // Step 5: Append the corresponding binary string from the map to the binary string
        binary_string += hex_to_bin_map[upper_digit]

    // Step 6: Return the binary string
    return binary_string
```

Appendix D: EC Standard Curves

1. Secp192k1

Secp192k1 is a standard EC which has the following parameters represented in Figure 73:

Parameters

Name	Value
p	0xfffffffffffffffffffffffffffffffffffffffeffffee37
a	0x000000000000000000000000000000000000000000000000
b	0x000000000000000000000000000000000000000000000003
G	(0xdb4ff10ec057e9ae26b07d0280b7f4341da5d1b1eae06c7d, 0x9b2f2f6d9c5628a7844163d015be86344082aa88d95e2f9d)
n	0xfffffffffffffffffffffffe26f2fc170f69466a74defd8d
h	0x1

Figure 73: Secp192k1 EC Parameters

2. Secp224k1

Secp224k1 is a standard EC which has the following parameters represented in Figure 74:

Parameters

Name	Value
p	0xfffffffffffffffffffffffffffffffffffffffffffffffeffffe56d
a	0x00000000000000000000000000000000000000000000000000000000
b	0x00000000000000000000000000000000000000000000000000000005
G	(0xa1455b334df099df30fc28a169a467e9e47075a90f7e650eb6b7a45c, 0x7e089fed7fba344282cafbd6f7e319f7c0b0bd59e2ca4bdb556d61a5)
n	0x10000000000000000000000001dce8d2ec6184caf0a971769fb1f7
h	0x01

Figure 74: Secp224k1 EC Parameters

3. Secp256k1

Secp256k1 is a standard EC which has the following parameters represented in Figure 75:

Parameters

Name	Value
p	0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f
a	0x0000000000000000000000000000000000000000000000000000000000000000
b	0x0000000000000000000000000000000000000000000000000000000000000007
G	(0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798, 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8)
n	0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
h	0x1

Figure 75: Secp256k1 EC Parameters