Analyzing Computational Complexity of Popular Cryptographic Algorithms to Determine the Most Efficient Cryptosystem

M. Rayhan Farrukh - 13523035
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]rayhan.farrukh@gmail.com, 13523035@std.stei.itb.ac.id

***Abstract*— In the digital age, where nearly every interaction is conducted online, ensuring security of data transmitted through the internet is important. Cryptography serves as a critical tool for such tasks However, while security is essential, efficiency is equally important, to ensure scalability and usability of cryptographic algorithms. This paper evaluates the efficiency of three widely-used cryptographic algorithms: RSA, ECC, and AES, through both theoretical complexity analysis and empirical runtime testing. The results demonstrate that AES is the fastest for encryption and decryption, ECC provides a balance between efficiency and security, and RSA is the slowest but remains valuable for secure key exchange. These findings emphasize the importance of balancing performance and security when selecting cryptographic algorithms for specific applications.**

***Keywords*— AES, ECC, RSA, time complexity**

I. Introduction

In the modern age, where close to every interaction is digitalized, many information are shared around through networks between people who are countries, even continents apart. This poses a serious security risk, as a competent hacker can intercept these interactions to extract sensitive informations. For this reason there needs to be some protocols and systems to ensure security of online transactions. One such tool is cryptography.

Cryptograpy plays a critical role in ensuring data security, it is used everywhere, spanning various industries and platforms. Companies like Google and Amazon use protocols such as *TLS* and *SSL* to protect data and secure online transactions. Moreover, cryptography is implemented for messaging apps like Whatsapp, for their end-to-end encryption, and even blockchain technologies, like Bitcoin, to ensure privacy and data integrity. For this reason, it is imperative to construct robust cryptosystems that are immune to cyberattacks.

However, while security is critical, the efficiency of such cryptosystems is also an important variable to consider. As the volume of data grows, the computational cost of cryptographic processes must be optimized to ensure usability and scalability, making analysis of computational complexity a critical aspect of evaluating cryptosystems implementation.

This paper will focus on the analysis for time complexity of three widely used cryptographic algorithms: RSA, ECC, and AES. By analyzing their encryption and decryption process, this paper's study aims to provide a deeper understanding of their efficiency and suitability for various security applications.

II. Theoretical Foundation

A. Computational Complexity

Computational Complexity, or often referred to as *Algorithmic Complexity*, is a branch of theoretical computer science that focuses on the calculations of resources to solve a problem, such as the time or space (memory) required to solve a problem. It categorizes algorithms based on their growth rates as input sizes increase in order to analyze their efficiency and aptitude for the problems specified.
Key concepts in computational complexity include:

1. Growth of T(n)
   The function T(n) represents the precise growth of resources required by an algorithm as the input size n increases. For example, the T(n) of both comparisons and swaps combined for the worst case of bubble sort is as such:

$$T(n) = n(n-1) = n^2 - n \quad (1)$$

   For time complexity, T(n) is used to quantify the number of basic operations performed in the algorithm, whereas for space complexity, it quantifies the amount of memory used by the algorithm.
2. Asymptotic Notations
   Asymptotic notations represents the behaviour in complexity of algorithms as the input size n grows very large. They drop the specificity of the T(n), focusing on the dominant growing factor by taking the highest order terms and removing constanst and lower order terms. For example, the asymptotic notation for the worst case of bubble sort algorithm is $O(n^2)$.
   Asymptotic notations are divided into three:
   a. Big-O ($O(n)$): Upper Bound, represents the worst case growth rate of an algorithm.
   b. Big-Omega ($\Omega(n)$): Lower Bound, describes the best-case growth rate.

c. Big-Theta ($\theta(n)$): Tight Bound, used when Big-O = Big-Omega, providing an exact (tight-bound) growth rate for the algorithm

Asymptotic notations are used in place of T(n) to focus only on the scalability of the algorithm, ignoring implementation-specific details.

In *cryptography*, computational complexity is critical for identifying trade-offs between security and efficiency, by analyzing the resources that a specific cryptographic algorithm requires as opposed to their strength. Computational complexity is also important in calculating a cryptographic algorithm's resistance to brute-force attacks or *cryptanalytic* methods.

B. Cryptography and Cryptosystems

Cryptography is the practice of hiding or obscuring information or data in order to secure it from being read by unauthorized parties. The process of obscuring information into unreadable form is called *encryption*, whereas the process for retrieving information back from the encrypted form is called *decryption*. An example of a basic cryptographic technique is shifting letters by a certain amount to deform the text (e.g. Caesar Cipher). More advanced techniques rely on mathematical foundations to ensure security. For instance, RSA (Rivest-Shamir-Adleman) uses modular arithmetic and prime factorization, and ECC (Elliptic Curve Cryptography) leverages properties of elliptic curves to secure informations.

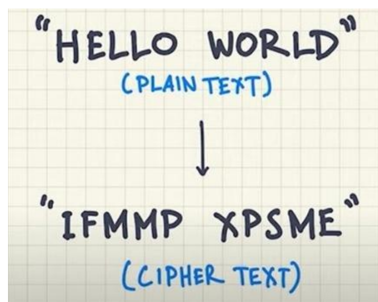

Figure 1. Example of Caesar Cipher with the text "Hello World" shifted by one letter
Source: https://www.researchgate.net/profile/Plaintext-and-the-Corresponding-Ciphertext-using-Caesar-Cipher-with-Key-1.jpg

In practice, cryptography requires a *cryptosystem* to ensure its effectiveness in securing data. A cryptosystem is a framework that combines cryptographic algorithms and protocols required to implement a secure data encryption and decryption. It tipically consists of the following components:

1. Key Generation, produces the key used in encryption and decryption
2. Encryption, converts plaintext into a ciphertext using a key
3. Decryption, converts ciphertext back into the plaintext

Cryptosystems are classified into two main types based on how encryption and decryption keys are used: *symmetric* and *asymmetric*.

1. Symmetric

Symmetric cryptosystems uses a single key for both encryption and decryption, making them efficient. However, the key has to be shared between the intended parties which requires a secure method for exchanging, this can be a challenge in large networks.

2. Asymmetric

Asymmetric cryptosystems use different keys for the encryption and decryption. The key for encryption is called the public key, as it is not required to keep this key a secret, while the key for decryption is called the private key.

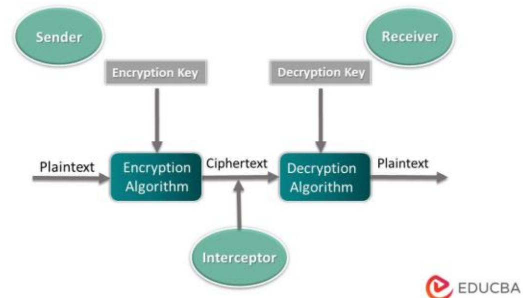

Figure 2. Illustration of a cryptosystem
Source: https://cdn.educba.com/academy/wp-content/uploads/2019/08/What-is-Cryptosystems-1.jpg

C. Rivest-Shamir-Adleman (RSA)

RSA is an asymmetric cryptosytem inroduced in 1977 by Ron Rivest, Adi Shamir and Len Adleman. RSA is the most widely used public-key cryptosystem by virtue of its simplicity and reliability to provide both data encryption and digital signatures. RSA relies on modular arithmetic and number theory. It is based on the computational difficulty of factoring a product of two large primes *p* and q, which provides security for its encryption method. The process in RSA include:

1. Key Generation

Generating the key starts with selecting two large prime numbers *p* and *q*, which will be kept private. These primes are multiplied to compute the modulus *n*:

$$n = p \times q \quad (2)$$

where *n* is a part of the public-key, which means it does not have to be kept secret. After calculating *n*, the totient $\phi(n)$ is then calculated as:

$$\phi(n) = (p-1)(q-1) \quad (3)$$

Next, a public exponent *e* is chosen such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$, to ensure that *e* is coprime with the totient. Finally the private-key, or private exponent *d* is computed as the modular multiplicative inverse of *e* modulo $\phi(n)$, as such:

$$d = e^{-1} \bmod \phi(n) \quad (4)$$

This private-key d is kept secret, and will be used for the decryption process.

2. Encryption

   To encrypt a plaintext P, it must first be converted into numerical form that satisfies $0 \leq P \leq n$. This is done by using an encoding scheme, such as ASCII or UTF-8. Once converted into numbers, the sender uses the public key to compute the ciphertext C:

$$C = M^e \bmod n \tag{5}$$

   Notice that in the equation, no private keys are used, so anyone can encrypt a message to result in the same ciphertext as the original P. This means this process *can* be brute-forced, but it will be ridiculously expensive.

3. Decryption

   To obtain the plaintext P from the received C, the recipient will use the private key d along with public key n as such:

$$P = C^d \bmod n \tag{6}$$

   The properties of modular arithmetic ensure that it is infeasible for anyone without d (and thus without knowledge of p and q) to decrypt the ciphertext.

D. Elliptic Curves Cryptography (ECC)

Elliptic Curves Cryptography is a modern public key cryptosystem that leverages the mathematical properties of elliptic curves, defined over finite fields. These properties provides efficient operations, smaller key sizes, and strong security based on the computational infeasibility of the *Elliptic Curve Discrete Logarithm Problem* (ECDLP)[7].

ECC is based on the equation of an elliptic curve, typically written as:

$$y^2 = x^3 + ax + b \bmod p \tag{7}$$

where a and b are constants, an p is a prime number defining the finite field. For the curve to be usable in ECC, it needs to be *non-singular* so that *group law* applies to it. A curve is non-singular if it satisfies the folowing equation:

$$4a^3 + 27b^2 \neq 0 \bmod p \tag{8}$$

As mentioned previously, the security of ECC comes from ECDLP, which is computationally difficult to solve. This problem involves finding k such that:

$$Q = kP \tag{9}$$

where Q and P are points on the curve, and k is a scalar multiplier.

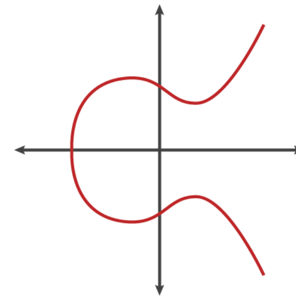

Figure 3. An elliptic curve, typically used for ECC Source: https://cf-assets.www.cloudflare.com/zkvhlag9/image00.png

The cryptographic process in ECC involves:

1. Key Generation

   Key generation in ECC is done by selecting a base point G called the generator, and a randomly chosen private key d, such that $1 \leq d < n$, where n is the order of the curve. The public key of ECC is calculated according to (9), with the generator as P and the private key as k, and the resulting public key Q is a point along the curve.

2. Encryption

   To encrypt a message, the sender first converts the plaintext into a point P on the curve. The ciphertext is divided into two points, obtained using the following equations:

$$C_1 = kG,\ C_2 = M + kQ \tag{10}$$

   Where k is a random integer chosen for the encryption process. The value of k will not be needed in the decryption process, although ensuring that k is truly random will make the ciphertext more secure.

3. Decryption

   In order to decryp the ciphertext, the recipient uses their private key d to get the plaintext M, according to (10) with the following process.
   Since $Q = dG$, and $C_1 = kG$, then we can write:

$$kQ = kdG = d(kG)$$
$$kQ = dC_1$$

   after acquiring kQ we can then subtract it from C_2 to obtain the plaintext M:

$$C_2 = M + kQ$$
$$M = C_2 - kQ$$
$$M = C_2 - dC_1 \tag{11}$$

The most basic application of ECC is the Elliptic Curve Diffie-Helman (ECDH). It is a protocol for establishing a shared secret over an insecure channel. Each party generates a private key, then derive the public key using scalar multiplication on the elliptic curve. The shared secret will then be derived after the public keys are exchanged. This secret is used in obtaining the encryption key.

E. Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES), also known as *Rijndael*, is a symmetric cryptoraphic algorithm adopted as a federal standard by the U.S. National Institute of Standards and Technology (NIST) in 2001 due to its efficiency and security.

AES operates on data blocks of fixed size (128 bits) and supports key sizes of 128, 192, and 256 bits. AES uses 4x4 matrices of bytes, called the state, as representation for the data blocks. If the data block is larger than 128 bit, the data will be separated into different 4x4 matrices, where each cells of these matrices contains 1 byte of data. AES performs a *Substitution-Permutation Network* (SPN)[8] to transform plaintext into ciphertext through a series of rounds. The number of rounds depent on the size of the key:

- 10 rounds for 128-bit keys
- 12 rounds for 192-bit keys
- 14 rounds for 256-bit keys

Each round consists of the following processes:

1. SubBytes
   Each byte in the block is substituted using a fixed 16x16 substitution lookup table containing non-linear transformation of each possible byte value. This ensures that non-linearity, creating resistance to linear and differential cryptanalysis
2. ShiftRows
   In this step, each row is shifted left by its row index minus one. This operation creates diffusion by rearranging the data within the matrix.
3. MixColumns
   MixColumns aplies a linear transformation to each column of the state matrix using matrix multiplication over a finite field (256 bits). Each column is multiplied by a fixed matrix:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

Figure 4. Matrix used for MixColumns multiplication
Source: https://www.reddit.com/media?uFhow-exactly-does-mix-columns-work-in-aes-png

   The result is a new column with the input bytes mixed, creating stronger interdependence. This step is not included in the final round of enncryption to simplify the decryption process
4. AddRoundKey
   In this step, the state matrix is combined with a round key using XOR. Each byte of the state is XORed with the corresponding byte of the round key. The original key used will be the initial key XORed directly with the plaintext. This step is repeated in every round so that every round has their own unique key for the encryption. The process of generating unique key for each round is called *key expansion.*

The above rounds processes are repeated according to the encryption key size, to transform plaintext into cipher text in the encryption process. For decryption, the steps are reversed using their respective inverse functions to restore the original plaintext.

III. Methodology

The cryptographic algorithms analyzed in this paper are RSA, ECC and AES. The analysis is conducted through both theoretical calculations and empirical testing to evaluate the efficiency of each algorithms.

The theoretical calculations will focus on the core mathematical operations underlying the algorithm. For RSA, the focus will be on the time complexity of modular exponentiation, as well as the computational difficulty of prime factorization. For ECC, the analysis centers on scalar multiplication over elliptic curves and its reliance on the difficulty of ECDLP. As for AES, each key expansion process and the Substitution-Permutation Network's complexity are evaluated to assess performance of the algorithm. Overall, the focus will be on the key generation fo these algorithms as it is the most impactful factor of performance.

Empirical testing is done by implementing these algorithms in Python using the *Pycryptodome*, *tinyec,* and *sympy* libraries. Python is chosen for its simplicity of implementing these cryptographic algorithms, although with a trade-off in execution speed. The tests will measure encryption and decryption times for varying key sizes. Additionally, brute-force resistance is simulated on small data and key sizes to estimate the effort required to break each algorithm. All tests will be conducted on a Lenovo Legion 5i Pro Gen 7, equipped with an Intel i7 12700H processor and Nvidia GeForce RTX 3060 GPU.

IV. Implementation and Complexity Analysis

A. Generating Large Numbers

The first step in each algorithms used is to generate large numbers that serves as the basis for the key generation. The size of these numbers will correspond to the size of the key. In this paper, the key sizes used is 128, 192, and 256 bit. The numbers generated will not have the same characteristics for every algorithms, as they each have different requirements, as RSA and ECC require prime numbers. Consequently, RSA and ECC relies on primality testing to ensure the numbers generated are prime. A common approach for this is to use the *Miller-Rabin primality test*. The Miller-Rabin primality test is a probabilistic algorithm used to determine whether a number is likely a prime.

```
def modular_exponentiation(base, exp, mod):
    result = 1
    while exp > 0:
        if exp % 2 == 1:
            result = (result * base) % mod
        base = (base * base) % mod
        exp //= 2
    return result
```

Figure 5. Implementation of modular exponentiation
Source: Author's documentation

```python
def is_prime(n):
    """ Miller-Rabin primality test """
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False

    r, d = 0, n - 1
    while d % 2 == 0:
        r += 1
        d //= 2

    a = random.randint(2, n - 2)
    x = pow(a, d, n)
    if x != 1 and x != n - 1:
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True
```

Figure 6. Implementation of Miller-Rabin primality test
Source: Author's documentation

The implementation for Miller-Rabin test is as follows. Where n is the candidate number being tested:

1) Decompose $n - 1$ into $d * 2^r$, where d is odd.
2) Randomly select a base a in the range $[2, n - 2]$.
3) Compute $x = a^d \bmod n$. If $x = 1$ or $x = n - 1$, the candidate passes this round.
4) If not, repeatedly square x (up to $r - 1$ times) and check if $x = n - 1$. If it still does not satisfy, then the candidate is composite.

This implementation leverages modular exponentiation (step 3) for efficient computation.

The complexity for this algorithm is dominated by modular exponentiation, computed using repeated squaring. A single modular exponentiation has complexity:

$$O(k^2 \log k) \quad (12)$$

where k is the bit length of n.

The Miller-Rabin primality test is only required to obtain primes for RSA and ECC. As for AES, the process is much more straightforward, as it does not require prime numbers. Instead, AES simply requires a random number within the desired bit sizes. This eliminates primality testing, resulting in a time complexity:

$$O(k)$$

B. RSA

In general, RSA can be divided into three important processes, key generation, encryption, and decryption. Among the three, RSA's complexity is most dominantly influenced by the key generation, where as encryption and decryption have roughly the same complexity, with both of them relying on modular exponentiation. Due to this fact, the efficiency of RSA mostly depends on the bit size of the keys, not the data size. The complexity. The implementation and complexity analysis for RSA is like so:

1. Key Generation

```python
def generate_rsa_keys(bits=128):
    p = generate_prime(bits // 2)
    q = generate_prime(bits // 2)

    n, phi_n = modulus_and_phi(p,q)

    e = 65537
    d = mod_inverse(e, phi_n)

    return (e,n), (d,n)
```

Figure 7. Implementation of RSA key generation
Source: Author's documentation

The key generation process starts with generating two large prime numbers, each of the numbers will be $\frac{k}{2}$ bits in size where k is the size of the key. The process of randomly generating these numbers has the complexity of $O(k)$. Each generated number is a candidate for primality testing, which according to (12) have the complexity of $O(k^2 \log k)$. If the numbers generated is not a prime number then repeat iteration of the primality test is required on multiple candidates, this results in the complexity:

$$O(k \cdot k^2 \log k) = O(k^3 \log k)$$

Once p and q are generated, we calculate the public key n and its totient $\phi(n)$, both of these only involve arithmetic multiplication which has a complexity of $O(k \log k)$. Afterwards we obtain the private key d using modular inverse of e mod $\phi(n)$. The implementation uses *Extended Eucledian Algorithm* for this which require $O(k)$ division steps, with each step performing either a division or multiplication of k-bit numbers. The complexity of modular inverse calculation is $O(k^2)$, assuming no logarithmic overhead. Overall, the complexity of key generation is:

$$O(k^3 + k^2 + k^3 \log k) = O(k^3 \log k) \quad (13)$$

2. Encryption and Decryption

```python
def encrypt(plaintext, public_key):
    e, n = public_key
    ciphertext = [str(mod_exp(ord(char), e, n)) for char in plaintext]
    ciphertext = ' '.join(ciphertext)
    return ciphertext

def decrypt(ciphertext, private_key):
    d, n = private_key
    ciphertext_list = list(map(int, ciphertext.split()))
    plaintext_list = [chr(mod_exp(char, d, n)) for char in ciphertext_list]
    plaintext = ''.join(plaintext_list)
    return plaintext
```

Figure 8. Implementation of RSA Encryption & Decryption
Source: Author's documentation

Both encryption and decryption rely on modular exponentiation applied to each character of the plaintext/ciphertext. Modular exponentiation computes $x^y \bmod n$, where x is the character's numeric value, y is the key, n is the totient of the public key. The complexity of modular exponentiation is (12), and for modular

exponentiation on a text of length m the complexity becomes:

$$O(m \cdot k^2 \log k) \quad (14)$$

Encryption typically uses a small, fixed public exponent (e = 65537), where as decyption uses a large private exponent with k bits size. This causes a difference in performance, but alas the asymptotic complexity remains the same.

In conclusion, the overall complexity of RSA is a combination of key generation and encryption-decryption as such:

$$O(n^3 \log n + m \cdot n^2 \log n) \quad (15)$$

here, key generation will dominate for small messages ($m \ll n$), while encryption-decryption dominates for large messages. This reflects the trade-offs between key size n and message length m.

C. ECC

Much like RSA, ECC consists of three processes, key generation, encryption, and decryption. For these processes, the computational complexity is most significantly affected by scalar multiplication, which is the foundation of all ECC operations. Scalar multiplication is utilizes in all three processes, which makes the complexity distributed relatively evenly for all three process. Implementation and complexity analysis for ECC is as follows:

1. Operations

Elliptic curves operations will be done using an elliptic curve object, as shown below in Fig. 9:

```
class EllipticCurve:
    def __init__(self, a=a, b=b, p=p):
        """
        Elliptic curve: y^2 = x^3 + ax + b (mod p).
        """
        self.a = a
        self.b = b
        self.p = p

    def is_on_curve(self, x, y):
        return (y**2 - (x**3 + self.a * x + self.b)) % self.p == 0
```

Figure 9. Implementation of Elliptic Curve object
Source: Author's documentation

The elliptic curve used is NIST P-128, P-192, and P-256, the values of the elements of these curves can is not shown for the sake of brevity (refer to the appendix for the source code). This curve is used as it is widely chosen for its strong 128-bit security, excellent performance and it is standardized in cryptographic protocols.

The operations for ECC include point addition and point doubling, which will be used in the scalar multiplication process. For the sake of brevity, only scalar multiplication's complexity will be detailed here. Fig 10. Shows the code for scalar multiplication:

```
def scalar_multiplication(k, P, curve):
    """
    k * P on the elliptic curve using the double-and-add algorithm.
    """
    if k == 0 or P is None:  return None

    result = None  # Start with the point at infinity
    addend = P

    while k:
        if k & 1:  # if bit of k == 1
            result = point_addition(result, addend, curve)
        addend = point_doubling(addend, curve)
        k >>= 1  # divide by 2

    return result
```

Figure 10. Implementation of scalar multiplication
Source: Author's documentation

Scalar multiplication's complexity is dominated by point addition and point doubling. Both of these operations have roughly the same complexity as they are mostly doing the same operations with slight differences to account addition of a point with itself in doubling. The point addition process follows these steps:

1) Compute the slope (m), which uses subtractions, modular inversion and modular multiplication.
2) Compute the new x coordinate, this uses subtractions and modular squaring
3) Compute the new y coordinate, this step uses subtractions and modular multiplication.

The steps are slightly different for point doubling, but the complexity is more or less the same. In total, these steps combine has the total complexity of:

$$O(k^2 \log k)$$

For scalar multiplcation, it uses the *double-and-add* algorithm, which involves iterating the bits of k through $\log_2 k$ iterations, and since scalar multiplication is dominated by point addition and doubling, the total complexity becomes:

$$O(\log_2 k \cdot k^2 \log^2 k) = O(k^2 \log^2 k) \quad (16)$$

2. Key generation

The first step in key generation is to choose a point G on the curve as a generator point, which in this case G is the NIST standard point, different for each key sizes. The value of these points can be seen within the source code (refer to the appendix).

```
G = (
    48439561293906451759052585252797914202762949526041747995844080717082404635286,
    36134250956749795798585127919587881956611106672985015071877198253568414405109
)

def generate_key(curve, G=G):
    # Private key
    d = random.randint(1, curve.p - 1)

    # Public key
    Q = scalar_multiplication(d, G, curve)

    return d, Q
```

Figure 11. Implementation of key generation for ECC
Source: Author's documentation

There are two keys generated, the private key, and the public key. The private key is chosen from the range $[1, p - 1]$, where p is the prime modulus of the curve. Usually calculating p is done randomly generating it, and have it go through primality testing, but since we are using the predefined NIST standard curve, there is no need to generate p. This makes the complexity for private key generation significantly faster with the complexity of $O(1)$.

This private key is then used to calculate the public key by multiplying it with the generator point G. This step uses scalar multiplication which has complexity of (16). So overall, the computational complexity of key generation of ECC is the same as that of scalar multiplication if the prime p is predetermined. Otherwise, ECC key generation is dominated by Miller-Rabin primality test which has the complexity of (13).

3. Encryption and Decryption

Much like RSA, the encryption and decryption in ECC have similar complexities with slight differences.

```python
def encrypt_ecc(plaintext, public_key, curve, G):
    k = random.randint(1, curve.p - 1)
    C1 = scalar_multiplication(k, G, curve)
    kQ = scalar_multiplication(k, public_key, curve)

    shared_x = kQ[0]  # Use the x-coordinate of S as the shared secret
    ciphertext_list = [str(ord(M) + shared_x) for M in plaintext]
    ciphertext = ' '.join(ciphertext_list)  # C2
    return C1, ciphertext

def decrypt_ecc(ciphertext, R, private_key, curve):
    kQ = scalar_multiplication(private_key, R, curve)

    shared_x = kQ[0]  # Use the x-coordinate of S as the shared secret
    ciphertext_list = list(map(int, ciphertext.split()))
    plaintext_list = [chr(char - shared_x) for char in ciphertext_list]
    plaintext = ''.join(plaintext_list)
    return plaintext
```

Figure 12. Implementation of ECC encryption and decryption
Source: Author's documentation

The steps for encryption are as follows:

1) Pick a random scalar k as ephemeral key
2) Compute $C_1 = kG$ using scalar multiplication
3) Compute kQ using scalar multiplication
4) Obtain C_2 as the ciphertext by adding the x-coordinate of kQ, this is done to every character in the plaintext

The complexity of this process is equal to the complexity of scalar multiplication (16). As for decryption, we only need to compute kQ using private key d and part of the ciphertext C_1, then we obtain the plaintext using (11). Encryption and decryption has similar complexity, although encryption takes slightly more time because scalar multiplication is done twice, which makes it slightly more computationally expensive than decryption.

Overall, the total complexity of ECC is dominated by scalar multiplication, this reflects the process as each step in ECC require scalar multiplication. The complexity of ECC is then:

$$O(n^2 \log^2 n) \qquad (17)$$

D. AES

AES works differently than the two previous algorithms discussed. Unlike RSA and ECC, AES is a symmetric key algorithm designed for high speed data encryption. The core operations underlying AES is the Substitution-Permutation Network (SPN), which contributes the most in complexity due to its repetition during rounds calculation in both encryption and decryption process. Also unlike RSA and ECC, AES's key generation, called key expansion, is done on every round of AES. This process, along with other operations, is implemented as detailed below.

1. Substitution-Permutation Network

The heart of AES, SPN, along with its inverses, takes up most of the operations in data encryption and decryption, which means time complexity of SPN determines time complexity of AES.

```python
def sub_bytes(state):
    for i in range(len(state)):
        for j in range(len(state[i])):
            state[i][j] = S_BOX[state[i][j]]
    return state

def shift_rows(state):
    for i in range(1, len(state)):
        state[i] = state[i][i:] + state[i][:i]
    return state

def mix_columns(state):
    for i in range(4):
        col = state[i]
        state[i] = [
            galois_mult(col[0], 2) ^ galois_mult(col[1], 3) ^ col[2] ^ col[3],
            col[0] ^ galois_mult(col[1], 2) ^ galois_mult(col[2], 3) ^ col[3],
            col[0] ^ col[1] ^ galois_mult(col[2], 2) ^ galois_mult(col[3], 3),
            galois_mult(col[0], 3) ^ col[1] ^ col[2] ^ galois_mult(col[3], 2),
        ]
    return state

def add_round_key(state, round_key):
    for i in range(4):
        for j in range(4):
            state[i][j] ^= round_key[i][j]
    return state
```

Figure 13. Implementation of SPN
Source: Author's documentation

Fig. 13 shows the implementation of normal SPN used in encryption, it constitutes four operations, SubBytes, ShiftRows, MixColumns, and RoundKey addition. For decryption, the SPN used is the inverses of these operation, which has the same time complexity, meaning encryption and decryption has the same time complexity. For the sake of brevity, the implementation for the inverses will not be shown here, readers' may refer to the appendix for further exploration. The complexity for each of these operations is described below, where n is the size of the state matrix

1) SubBytes, matrix element substitution using the S-box lookup table. Complexity: $O(n)$
2) ShiftRows, simple shifting of rows. Complexity: $O(n)$
3) MixColumns, this operation uses Galois multiplication for the matrix multiplication. The complexity of Galois matrix multiplication is: $O(m \cdot n)$, where m is the cost of Galois multiplication. In AES, the state has fixed size where m is only 1, so complexity simplifies to $O(n)$
4) AddRoundKey, each byte of the state is XORed with the round key. Complexity: $O(n)$

From the details above, we can conclude the total complexity of SPN as follows, where Nr is the number of rounds:

$$O(n \cdot Nr) \tag{18}$$

2. Key Expansion

Key expansion is used in deriving a series of round keys from the initial secret randomly generated key. The process involves splitting the key into 4-byte words, rotation and substitution with the S-box, and multiple XOR operations. The implementation is detailed below.

```
1  def key_expansion(key, key_size=128):
2      Nk = key_size // 32
3      Nr = {128: 10, 192: 12, 256: 14}[key_size]
4      Nb = 4  # State Columns
5
6      expanded_key = [key[i:i+4] for i in range(0, len(key), 4)]
7
8      for i in range(Nk, Nb * (Nr + 1)):
9          temp = expanded_key[i - 1]
10         if i % Nk == 0:
11             temp = sub_word(rot_word(temp))
12             temp[0] ^= RCON[(i // Nk) - 1]
13         elif Nk > 6 and i % Nk == 4:
14             temp = sub_word(temp)
15
16         expanded_key.append([x ^ y for x, y in zip(expanded_key[i - Nk], temp)])
17     return [expanded_key[i:i+Nb] for i in range(0, len(expanded_key), Nb)]
```

Figure 14. Implementation of AES key expansion
Source: Author's documentation

The processes involved in key expansion and their time complexity is detailed below:

1) Splitting the key, dividing the key requires simple slicing operations. Complexity: $O(Nk)$, where $Nk =$ key_size/32
2) Word rotation and substitution, rotates the bytes of a 4 byte word with cyclic left shift, and substituting the words using S-box lookup table. The complexity is $O(1) + O(4)$
3) XOR first byte of transformed word with round constants. Complexity: $O(1)$
4) Expanding the key, for each word, compute:
   $$\text{temp} = \text{prev_word} \oplus Nkth \text{ prev_word}$$
   Complexity for this is $O(Nk + Nb \cdot Nr)$, where *Nr* is the number of rounds and *Nb* is the number of columns per round

The total complexity for key expansion is consistent with the final step:

$$O(Nk + Nb \cdot Nr) \tag{19}$$

The encryption and decryption process in AES is simply the combination of SPN and key expansion, so for the sake of brevity, the implementation will not be shown in this paper, readers may explore the source code attached in the appendix to see it.

The total complexity of AES is a combination of SPN and key expansion's time complexity done on a number of state matrices (if the data is longer than 16 bytes). For multiple matrices, AES processes $\frac{D}{n}$ matrices sequentially, where *n* is the block size (128 bits). For *D*-bits sized data, the complexity becomes:

$$O\left(\frac{D}{n} \cdot n \cdot Nr\right) = O(D \cdot Nr) \tag{19}$$

and so the total time complexity of AES is as follows:

$$O(D \cdot Nr + Nk + Nb \cdot Nr) \approx O(D \cdot Nr) \tag{20}$$

here, $D \cdot Nr$ dominates for large data sizes.

V. Tests and Results

For better demonstration, the implementation will be tested empirically on different key sizes for each algorithms. The tests will evaluate the runtime of key generation, encryption and decryption. The encryption is done on the following 2256-bit text:

"*Life is brilliant. Beautiful. It enchants us, to the point of obsession. Some are true to their purpose, though they are but shells, flesh and mind. One man lost his own body, but lingered on, as a head. Others chase the charms of love, however elusive. What is it that drives you*?"

The result of the tests are shown in tables below.

Table I. Test results of RSA implementation (in seconds)

Key size	Key Gen.	Encryption	Decryption	Total
128-bit	9.98×10^{-4}	9.98×10^{-4}	1.18×10^{-2}	1.38×10^{-2}
192-bit	1.47×10^{-3}	1.05×10^{-3}	2.22×10^{-2}	2.52×10^{-2}
256-bit	2.01×10^{-3}	9.99×10^{-4}	4.15×10^{-2}	4.45×10^{-2}

Table II. Test results of ECC implementation (in seconds)

Key size	Key Gen.	Encryption	Decryption	Total
128-bit	1.02×10^{-3}	2.00×10^{-3}	1.00×10^{-3}	4.02×10^{-3}
192-bit	2.00×10^{-3}	5.00×10^{-3}	2.02×10^{-3}	9.02×10^{-3}
256-bit	5.00×10^{-3}	1.00×10^{-2}	5.01×10^{-3}	2.00×10^{-2}

Table III. Test results of AES implementation (in seconds)

Key size	Encryption	Decryption	Total
128-bit	5.01×10^{-4}	9.00×10^{-4}	1.40×10^{-3}
192-bit	6.00×10^{-4}	1.20×10^{-4}	1.80×10^{-3}
256-bit	8.00×10^{-4}	1.40×10^{-4}	2.20×10^{-3}

The runtime data of implementation aligns with the theoretical complexities calculated previously. AES performs the fastest encryption and decryption, and require no key generation. Its lightweight symmetric operation causes AES to be very efficient that even larger key sizes only result in slight increases in runtime. This aligns with the complexity as calculated in (20).

For ECC, the tests results indicate that it is more computationally expensive compared to AES, but still edging RSA's speed. This aligns with the calculated complexity as shown in (17). Meanwhile RSA has the highest complexity as derived In (15), resulting in the slowest total runtime, as well as exhibiting significant growth as the key size increases.

Overall, from the results of implementation testing it is found that among the three cryptographic algorithms, AES runs the fastest across all key sizes tested, followed by ECC, and lastly RSA.

VI. Conclusion

Results of both theoretical complexity analysis and empirical runtime tests points out the efficiency of each cryptographic algorithms being tested, which is RSA, ECC, and AES. From complexity analysis of the implementation, AES was calculated

to have a complexity of $O(D \cdot Nr)$ where D is the size of data being encrypted or decrypted in bits, and Nr is the number of rounds for the corresponding key sizes. Empirical testing supports this, with total runtime being significantly faster than ECC and RSA, especially for larger key sizes. ECC was found to have a complexity of $O(n^2 \log^2 n)$ where n is the key size, making it slower than AES but faster than RSA as can also be seen from test results. RSA has the highest complexity of $O(n^3 \log n + m \cdot n^2 \log n)$, where n is the key size and m is the data length, making it the slowest algorithm between the three.

These results mirrors the efficiencies of each algorithm, but it does not reflects their effectiveness. AES is significantly faster than the other algorithms, making it the best choice for encryption of large data. However this does not mean that AES is more secure than ECC or RSA. ECC and RSA, which are asymmetric cryptographic algorithms, offer strong security for key exchange and authentication. Each algorithm is secure when implemented correctly, but their use depends on specific requirements of the system. While this paper focuses on analyzing the efficiency of these algorithms, further evaluation of their security is required to determine the best cryptographic algorithm to use in a certain cryptosystem.

VII. Appendix

a. Github repository for this project: https://github.com/grwna/cryptosystem-complexity-analysis

VIII. Acknowledgment

The author would like to thank God for His endless blessings and guidance, as without it, this paper would not have been written succesfully. The deepest thanks also extended to my lecturer for Discrete Mathematics, Dr. Ir. Rinaldi Munir, M.T. for his dedication to guide students with patience and expertise.

The author would also like to thank famuly and friends for their constant and unwavering support and encouragement, which have been very important throughout the writing of this paper, and this academic journey.

References

[1] R. Munir, *Kompleksitas Algoritma Bagian 1*, 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/25-Kompleksitas-Algoritma-Bagian1-2024.pdf. [Accessed: Jan. 1, 2025].

[2] GeeksforGeeks, "RSA Algorithm in Cryptography," [Online]. Available: https://www.geeksforgeeks.org/rsa-algorithm-cryptography/. [Accessed: Jan. 4, 2025].

[3] Cloudflare, "A Relatively Easy to Understand Primer on Elliptic Curve Cryptography," [Online]. Available: https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/. [Accessed: Jan. 4, 2025].

[4] GeeksforGeeks, "Blockchain and Elliptic Curve Cryptography (ECC)," [Online]. Available: https://www.geeksforgeeks.org/blockchain-elliptic-curve-cryptography/. [Accessed: Jan. 4, 2025].

[5] GeeksforGeeks, "Advanced Encryption Standard (AES)," [Online]. Available: https://www.geeksforgeeks.org/advanced-encryption-standard-aes/. [Accessed: Jan. 4, 2025].

[6] Baeldung, "Understanding Cryptographic Algorithm Complexity," [Online]. Available: https://www.baeldung.com/cs/cryptographic-algorithm-complexity. [Accessed: Jan. 4, 2025].

[7] S. I. Serengil, "Elliptic Curve Discrete Logarithm Problem (ECDLP): Hardness of ECC," YouTube. [Online]. Available: https://www.youtube.com/watch?v=iKAotDNz5o8. [Accessed: Jan. 5, 2025].

[8] Substitution-Permutation Networks Article**:**
Naukri.com, "Substitution-Permutation Networks (SPN) in Cryptography," [Online]. Available: https://www.naukri.com/code360/library/substitution-permutation-networksspn-in-cryptography. [Accessed: Jan. 5, 2025].

[9] R. Munir, "Algoritma RSA," Informatika STEI ITB, 2023-2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2023-2024/18-Algoritma-RSA-2024.pdf. [Accessed: Jan. 8, 2025].

[10] R. Munir, "Beberapa Block Cipher Bagian 2," Informatika STEI ITB, 2023-2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2023-2024/15-Beberapa-block-cipher-bagian2-2024.pdf. [Accessed: Jan. 8, 2025].

Statement

I hereby declare that this paper is an original work, written entirely on my own, and does not involve adaptation, translation, or plagiarism of any other individual's work.

Bandung, 8 January 2024

M. Rayhan Farrukh, 13523035