Application of the Lenstra-Lenstra-Lovász (LLL) Lattice Basis Reduction Algorithm and Minkowski's Theorem to Optimize Small Private Key RSA Decryption

Nayaka Ghana Subrata - 13523090[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523090@mahasiswa.itb.ac.id, nayakaghana39@gmail.com

***Abstract*—With the vast development of computational technology and the increasing prevalence of massive cyber-attacks, cryptosystems have become increasingly essential. One of its examples is Rivest-Shamir-Adleman (RSA) cryptosystems, which has remained a robust solution for decades. However, due to quantum computing developments, RSA becomes unsafe. The discoveries of LLL algorithms to reduce lattice's basis and Minkowski's Theorem for shortest vector problem, making RSA more vulnerable. This paper investigates the theory behind and its implementation, such as encryption and decryption process. Experimental results demonstrate the effectiveness of RSA and Minkowski's Theorem, showcasing the time difference between general number field sieve and decrypt using lattice basis reduction with Minkowski's Theorem.**

***Keywords*—RSA, LLL, Minkowski, Cryptosystem.**

I. Introduction

With the recent development of computational technology, various types of cyber-attacks had been done massively. This condition increases the need for methods that can protect data effectively, one of which is through the cryptosystem concept. Cryptosystems are designed to maintain the confidentiality, integrity and authenticity of information in various forms of modern communication.

One of the important innovations in the field of cryptography is the discovery of the RSA algorithm which was discovered by three people: Rivest, Shamir, and Adleman in 1977. RSA is an asymmetric type of cryptosystem that uses 2 keys, a public key and a private key. These two keys were created using the concept of number theory and modular arithmetic with very large prime numbers to make the decryption process more complex.

However, due to the development of post-quantum computing, RSA then be considered as an unsafe cryptosystem. Quantum computers possess the capability to perform complex calculations at speeds unimaginable by classical computers. This includes solving the integer factorization problem, which is the mathematical foundation of RSA. With its capability, quantum computers could efficiently break RSA encryption, rendering it ineffective for securing data.

Moreover, in the pre-quantum era, certain advancements in mathematical algorithms, like the Lenstra-Lenstra-Lovász (LLL) algorithm, have already demonstrated vulnerabilities within RSA's mathematical structure. While LLL does not directly break RSA, it serves as a precursor to understanding how lattice-based attacks can exploit specific weaknesses in RSA's cryptosystems.

This paper aims to provide a comprehensive exploration of the application of LLL algorithm and Minkowski's theorem to break RSA cryptosystem, especially with RSA that uses small private key in its encryption system. To check the efficacy of the experiment, we calculated the execution time in the decryption process.

The paper has been organized as follows: Section 2 provides the theoretical framework, Section 3 provides the cryptosystem scheme, Section 4 provides the implementation, Section 5 provides the test and the result, and Section 6 provides the conclusion followed by references.

II. Theoretical Framework

A. Cryptosystem

Cryptosystem is an entire set of cryptographic systems needed necessary for the provision of a certain security services, such as data confidentiality and hiding data's crucial information (encryption-decryption process). This can also be defined as converting plaintext to ciphertext to encrypt and decrypt message securely.

In general, cryptosystem consists of three main algorithms: key generation, encryption, and decryption. The basic model of cryptosystem is depicted in the figure below:

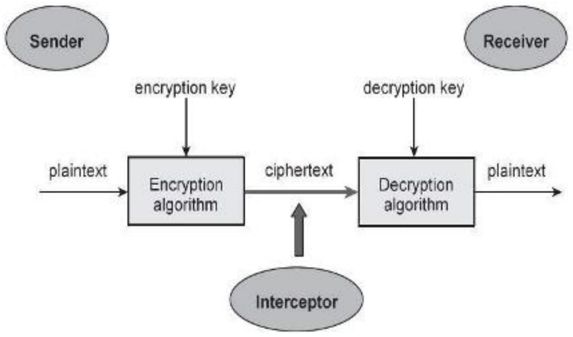


Fig. 2.1 *Basic cryptosystem model*
(Source: Adapted from [4])

Typically, there are two kinds of cryptosystems based on its key-generation process; the first kind of the cryptosystem is symmetric key cryptography, and the second kind of the cryptosystem is asymmetric key cryptography.

Symmetric key cryptography is a cryptography process that uses same keys for encryption and decryption process. A well-known example that uses this cryptosystem are Advanced Encryption Standard (AES), Data Encryption Standard (DES), International Data Encryption Algorithm (IDEA), Blowfish, and Rivest Cipher. Example for this encryption can be seen in Fig 2.2.

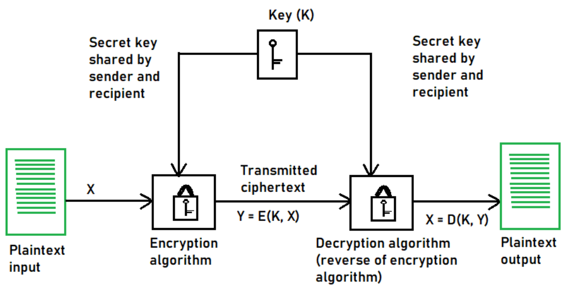

Key (K)
Secret key shared by sender and recipient
Secret key shared by sender and recipient
X
Transmitted ciphertext
Y = E(K, X)
X = D(K, Y)
Plaintext input
Encryption algorithm
Decryption algorithm (reverse of encryption algorithm)
Plaintext output

Fig. 2.2 *Basic symmetric key cryptography model*
(Source: Adapted from [2])

Asymmetric key cryptography is a cryptography process that uses different keys for encryption and decryption process. A well-known example that uses this cryptosystem are Rivest-Shamir-Adleman (RSA), Elliptic Curve Cryptography (ECC), Digital Signature Algorithm (DSA), Diffie-Hellman, and Certificate Authorities (CAs). Example for this encryption can be seen in Fig 2.3.

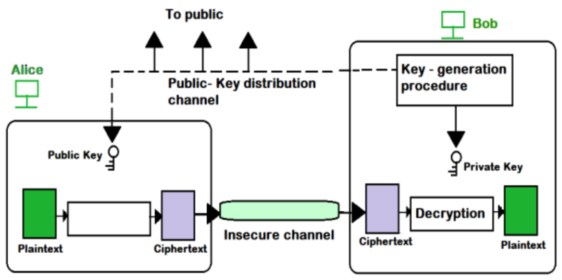


Fig. 2.3 *Basic asymmetric key cryptography model*
(Source: Adapted from [3])

B. Rivest-Shamir-Adleman (RSA)

Rivest-Shamir-Adleman (RSA) algorithm is one of the cryptosystems that uses asymmetric key to encrypt and decrypt the plaintext and the ciphertext. This algorithm is named after its founder: Ron Rivest, Adi Shamir, and Len Adleman in 1977.

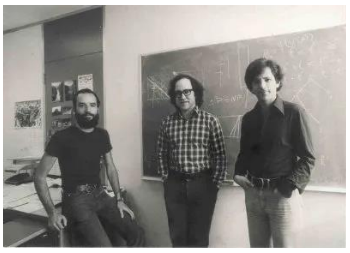


Fig 2.4 *(From left to right) Adi Shamir, Ron Rivest, and Len Adleman*
(Source: Adapted from [5])

a. Encryption

The encryption process of this algorithm is quite simple, first pick two primes, or namely p and q. The size of this primes is freely chosen, but it's recommended to pick big primes to make the decryption process more challenging and difficult.

After picking the two primes number (p and q), we can calculate the modulus for the encryption, or namely N. The N value can be calculated using the equations below:

$$N = pq \ldots (1)$$

With N is the modulus value, and pq is the product of the two primes number. Notice that, if we choose big size of integer for the p and q values, the n size is increased significantly too.

After we calculate N value, the next step is to pick the public exponent or sometimes called the encryption key value (e value). In general, we can pick 65537 (or 0x10001 in hexadecimal representation) to be the public exponent. This value picked because of its common compromise between being high, and its cost of raising to the *e-th* power. But keep in mind that the e value must be coprime with the Euler's totient value that usually represent in phi (φ) symbol (this totient value will be discussed in the decryption part).

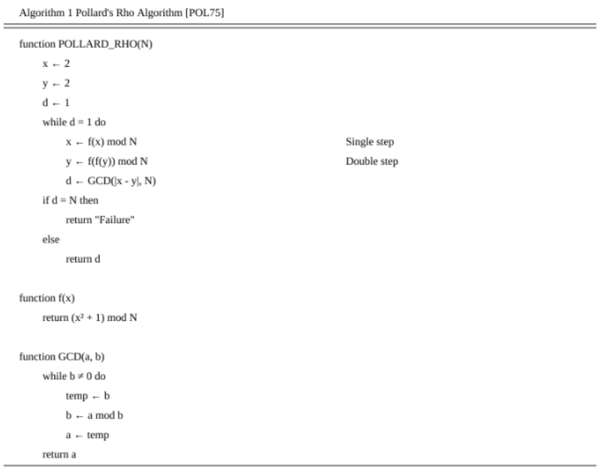
The final step of the RSA encryption process is to convert plaintext to ciphertext, or namely c. To calculate the c value, we must understand what number theory and modular arithmetic is. The c value can be calculated using the equations below:

$$c = m^e \bmod N \ldots (2)$$

With m is the plaintext representation in its integer value. After we calculate the c value, we can share the N, e, and c value to the receiver.

b. Decryption

The decryption process of this algorithm is quite challenging, first, we have to search for prime factors from N value (see eq. (1)), if the encryption process is using conventional RSA, we can use Pollard's Rho algorithm to search for the prime factors from N (or we're searching for p and q values). The algorithm can be seen in Fig 2.5.



```
Algorithm 1 Pollard's Rho Algorithm [POL75]

function POLLARD_RHO(N)
    x ← 2
    y ← 2
    d ← 1
    while d = 1 do
        x ← f(x) mod N                  Single step
        y ← f(f(y)) mod N               Double step
        d ← GCD(|x - y|, N)
    if d = N then
        return "Failure"
    else
        return d

function f(x)
    return (x² + 1) mod N

function GCD(a, b)
    while b ≠ 0 do
        temp ← b
        b ← a mod b
        a ← temp
    return a
```

Figure 2.5 *Pollard's Rho Algorithm*
(Source: writer's archive)

After getting the p and q values, calculate the Euler's totient, Euler's totient is a function to determine how much numbers are coprime relative to the N value (or suppose that the number is k, $1 \le k \le N$, greatest common divisor of k and N must be equal to 1).

Euler's totient is multiplicative function, meaning that if we have two coprime numbers, for example a and b, then:

$$\varphi(ab) = \varphi(a)\varphi(b) \ldots (3)$$

If n-set of numbers $(\{a_1, a_2, \ldots, a_n\})$ are pair-wisely coprime, then:

$$\varphi\left(\prod_{i=1}^{n} a_i\right) = \prod_{i=1}^{n} \varphi(a_i) \ldots (4)$$

From eq. (3), if b is a prime number, then $\varphi(b) = b - 1$. Notice that a and b are different prime numbers because a and b is coprime. From these results, we can get:

$$\varphi(ab) = \varphi(a)\varphi(b)$$
$$\varphi(ab) = (a-1)(b-1) \ldots (5)$$

With $\varphi(ab)$ is the Euler's totient value that we'll use to calculate the private key.

After calculating the Euler's totient value, we can calculate the private key value, namely d. To calculate d, we will use the equivalencies below:

$$d \equiv e^{-1} \bmod \big(\varphi(N)\big) \ldots (6)$$

From eq. (6), calculate d using modular inverse concept, after we get the d value, we can convert ciphertext to its plaintext using this equation below:

$$m = c^d \bmod N \ldots (7)$$

With c is the ciphertext representation in its integer value. After we calculate the m value, convert it to its string value to get the plaintext.

C. Lenstra-Lenstra-Lovász (LLL) Lattice Basis Reduction

Lattice can be described by a basis B which contains linearly independent basis vectors $(\{b_1, b_2, \ldots, b_r\})$ with $b_i \in \mathbb{R}^n$ and r is the lattice's rank. The lattice can be represented as:

$$L = L(B) = \left\{\sum_{i=1}^{r} a_i b_i \mid a_i \in \mathbb{Z}\right\} \ldots (8)$$

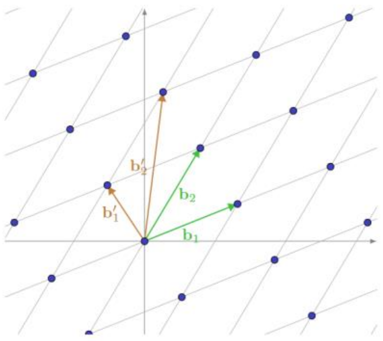


Figure 2.6 *A 2-dimensional lattice*
Source: Adapted from [1]

From that definition, we can reduce the lattice by take an arbitrary lattice's basis and transform it to another basis that has shorter and has more orthogonal vectors.

To begin, suppose that we have two matrixes, $T_{i,j}$ and $L_{i,j}(k)$

$$T_{i,j} = \begin{array}{c} \left[\begin{array}{ccccccc} 1 & & & & & & \\ & \ddots & & & & & \\ & & 0 & & 1 & & \\ & & & \ddots & & & \\ & & 1 & & 0 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{array}\right] \end{array} \begin{array}{c} \\ \\ i \\ \\ j \\ \\ \\ \end{array} \qquad L_{i,j}(k) = \begin{array}{ccccccc} & & j & & & & \\ \end{array} \left[\begin{array}{ccccccc} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & \ddots & & & \\ & & k & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{array}\right] \begin{array}{c} \\ \\ \\ \\ i \\ \\ \\ \end{array}$$

Figure 2.7 $T_{i,j}$ and $L_{i,j}(k)$ Matrix
Source: Adapted from [1]

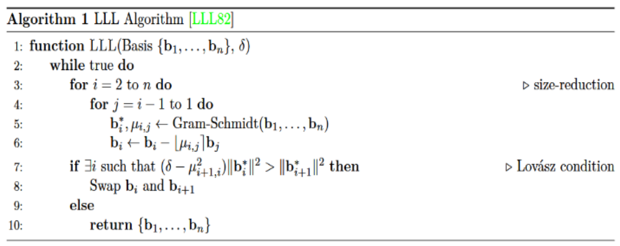
If we left multiplying a basis with $T_{i,j}$, it will yield a new basis with swapped i and j basis vectors. If we left multiplying a basis with $L_{i,j}(k)$, it will yield a new basis with the j_{th} basis added k times to i_{th} basis. These two transformations will be used in the Lenstra-Lenstra-Lovász (LLL) algorithm.

The Lenstra-Lenstra-Lovász (LLL) algorithm starts with taking the lattice basis and computing orthogonal basis with Gram-Schmidt methods.

$$\begin{cases} b_i^* = b_i, & i = 1 \\ b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{i,j} b_j^*, & 1 < i \leq n \end{cases} \quad \mu_{i,j} = \frac{(b_i, b_j^*)}{(b_j^*, b_j^*)}$$

With b_i^* is the orthogonal vector and $\mu_{i,j}$ is the coefficient.

After getting the orthogonal basis vectors, we can do the LLL algorithm. The algorithm can be seen in Fig 2.8.



Algorithm 1 LLL Algorithm [LLL82]

```
function LLL(Basis $\{b_1, \dots, b_n\}$, $\delta$)
    while true do
        for $i = 2$ to $n$ do                                          ▷ size-reduction
            for $j = i - 1$ to 1 do
                $b_i^*, \mu_{i,j} \leftarrow$ Gram-Schmidt($b_1, \dots, b_n$)
                $b_i \leftarrow b_i - \lfloor \mu_{i,j} \rceil b_j$
        if $\exists i$ such that $(\delta - \mu_{i+1,i}^2)\|b_i^*\|^2 > \|b_{i+1}^*\|^2$ then     ▷ Lovász condition
            Swap $b_i$ and $b_{i+1}$
        else
            return $\{b_1, \dots, b_n\}$
```

Figure 2.8 *LLL Algorithm*
(Source: Adapted from [1])

Notice there are two conditions to be fulfilled. The first condition is size reduction. This condition related to the basis vectors length that can be represented as $|\mu_{i,j}| \leq \frac{1}{2}$ for all $i > j$.

The second condition is Lovász condition, Lovász states that lattice is reduced if $(\delta - \mu_{i+1,i}^2)||b_i^*||^2 \leq ||b_{i+1}^*||^2$ for all $1 \leq i \leq n - 1$, with $\delta \in (0.75, 1)$.

D. Minkowski's Theorem

In 1889, Hermann Minkowski, A German Mathematician, states that every convex set in n-dimensional spaces ($\mathbb{R}^n$) that symmetric with the origin coordinate and has volume greater than 2^n, contains a non-zero integer points. This means that a Minkowski's point is in the infinite $\mathbb{Z}^n$ space excluding its origin point.

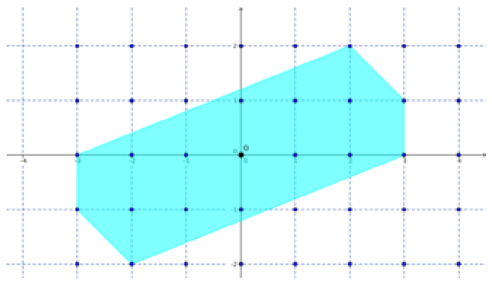


Figure 2.9 *A Minkowski's set in 2-dimensional space*
Source: (https://www.anyrgb.com/en-clipart-2haql#google_vignette)

Later then, he introduces the extended version of the theorem, Minkowski's First Theorem. This theorem gives an upper bound for the length of the shortest non-zero vector.

First, we have to understand successive minima of a lattice. Let n be the lattice's rank, and L be a full-rank n-*dimensional* lattice, for $i \in \{1,2, \dots, n\}$, the i_{th} successive minima of L (λ) is the smallest r so that L has i linearly independent vectors of biggest length r. So, Minkowski, in his first theorem, propose that:

$$\lambda(L) \leq \sqrt{n} |\det(L)|^{\frac{1}{n}} \dots (9)$$

The example of this theorem can be seen in Fig 2.9.

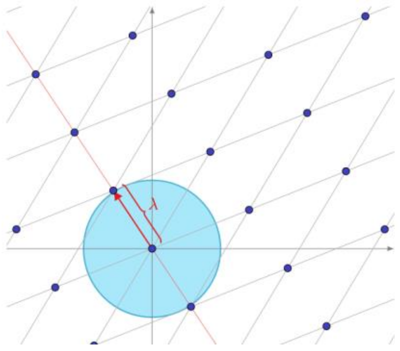


Figure 2.10 *Successive minima and shortest non-zero vector of a lattice*
(Source: Adapted from [1])

III. CRYPTOSYSTEM SCHEME

A. Encryption

The proposed encryption method begins with picking some random *p* and *q* numbers (or two random prime numbers) with 512 bits of size. Then, calculate *N* (modulus) with eq. (1).

Notice that the size of *N* is very big (approximately 1024 bits of size), making it harder and challenging to compute on the decryption process later.

After choosing *p* and *q* and calculating *N* value, the next step is to calculate the Euler's totient value with eq. (5). To make it harder to decrypt, we will separate the plaintext into two ciphertext and two public exponents (notice that we will have two private keys and two ciphertext from this scheme). Suppose we want to search for *d1* (first private key), *e1* (first public exponent), *d2* (second private key), and *e2* (second public exponent) values. First, take random *d1* value, but make it 0.16 times smaller than the *N* size (so we have first private key with approximately 164 bits of size), then pick random *e1* values from 1 to Euler's totient value. After we get the *d1* and *e1* values, calculate *ed1* with these equations below:

$$ed1 = (e1d1) \bmod \varphi \dots (10)$$

After that, take random *d2* value, but make it 0.16 times smaller than the *N* size (so we have first private key with approximately 164 bits of size), then calculate *e2* with calculating factor from *d2* and Euler's totient value with extended Euclidean algorithm (egcd). After getting the *e2*, multiply it by $(\varphi + 1 - ed1) \bmod \varphi$. After we get the *d2* and *e2* values, calculate *ed2* with these equations below:

$$ed2 = (e2d2) \bmod \varphi \dots (11)$$

The next step, is to assert if the conditions shown below is met:

$$e1d1 + e2d2 \equiv 1 \; mod \; \varphi \ldots (12)$$

If the conditions are met, the next step is to convert plaintext to its ciphertext, we have two ciphertext here, *c1* and *c2*. Calculate each ciphertext with eq. (2), after that, we can share the *N*, *e1*, *c1*, *e2*, *c2* to the receiver. More detailed description provided in the implementation section.

B. Decryption

From the sender, we only get *N*, *e1*, *c1*, *e2*, *c2* values, and from the encryption process, we know that $d1, d2 < N^{0.16}$, $e1d1 + e2d2 \equiv 1 \; mod \; \varphi$, and plaintext ($m$) with $m \equiv (c1^{d1} c2^{d2}) \; mod \; N$.

Because of its private key are small (*d1* and *d2* is small), we can first assume that the vector produced is small and we can use smallest vector problem. First, extend eq. (12) first:

$$\begin{gathered} e1d1 + e2d2 \equiv 1 \; mod \; \varphi \\ e1d1 + e2d2 = 1 + k\varphi \\ e1d1 + e2d2 = 1 + k((p-1)(q-1)) \\ e1d1 + e2d2 = 1 + k(pq - p - q + 1) \\ e1d1 + e2d2 = 1 + k(pq - (p+q) + 1) \\ e1d1 + e2d2 = 1 + k(N - (p+q) + 1) \\ e1d1 + e2d2 - kN = 1 + k(-(p+q) + 1) \ldots (13) \end{gathered}$$

From eq. (13), we can approximate each size of the components. p, q = 512 bits; *N* = 1024 bits; *d* = 164 bits; *e* = 1024 bits; and *k* = 164 bits (from min(*e*,*d*)).

Because of smallest vector problem, we can use LLL algorithm, first take 3 vector basis, the first basis is $b_1 = (1, 0, e_1)$, the second basis is $b_2 = (0, 1, e_2)$, and the third basis is $b_3 = (0, 0, -N)$. This basis is adapted from the eq. (13) (or from the $e1d1 + e2d2 - kN$ parts) and to keep maintaining the bits size too.

After we take 3 basis, make the matrix representation of it:

$$\begin{bmatrix} 1 & 0 & e_1 \\ 0 & 1 & e_2 \\ 0 & 0 & -N \end{bmatrix}$$

Check the matrix with Minkowski's first theorem (eq. 9), we have 3-dimensional lattice, with $|\det(L)| = |(1)(1)(-N)| = N$, so we can approximate $\sqrt{n} |\det(L)|^{\frac{1}{n}}$ bits size, that is $\frac{|\det(L)| \; size}{n \; size} \approx \frac{1024}{3} bits \approx 341 \; bits$

So, the inequality becomes:

$$\lambda(L) \leq 341 \; bits \ldots (14)$$

The $\lambda(L)$ value is the $1 + k(-(p+q) + 1)$ value, we know that $k \approx 164$ bits and $p + q \approx 512$ bits, so we can approximate the $\lambda(L)$, that is 676 bits. The inequality becomes:

$$676 \; bits \leq 341 \; bits \ldots (15)$$

Which is false, so we have to scale the basis to make the Minkowski's first theorem conditions are met. Suppose that we scale the basis with M, so the pre-LLL matrix will be:

$$\begin{bmatrix} M & 0 & e_1 \\ 0 & M & e_2 \\ 0 & 0 & -N \end{bmatrix}$$

Check the matrix with Minkowski's first theorem (eq. 9), we have 3-dimensional lattice, with $|\det(L)| = |(M)(M)(-N)| = M^2 N$. Keep in mind that we have to make the $\sqrt{n} |\det(L)|^{\frac{1}{n}}$ bits size is greater or equal than 676 bits. From this conditions, we can take $M = 2^{512}$ so the M size is 512 bits, calculate the size for the $\sqrt{n} |\det(L)|^{\frac{1}{n}}$, that is $\frac{|\det(L)| \; size}{n \; size} \approx \frac{512+512+1024}{3} bits \approx \frac{2048}{3} bits \approx 682 \; bits$, which finally satisfies the Minkowski's conditions.

After getting the pre-LLL matrix, reduce it by the LLL algorithm shown in Fig 2.8. After we get the reduced lattice, calculate the *d1* and *d2* values from the first row, the first value from the row is the *d1* value and the second value from the row is the *d2* value, but don't forget to divide each *d* by M because we did scale the value by M before.

Finally, we have all the value needed to decrypt the ciphertext. Calculate the plaintext value using these equivalencies below:

$$m \equiv (c1^{d1} c2^{d2}) \; mod \; N \ldots (16)$$

Then, convert it to its string value to get the plaintext.

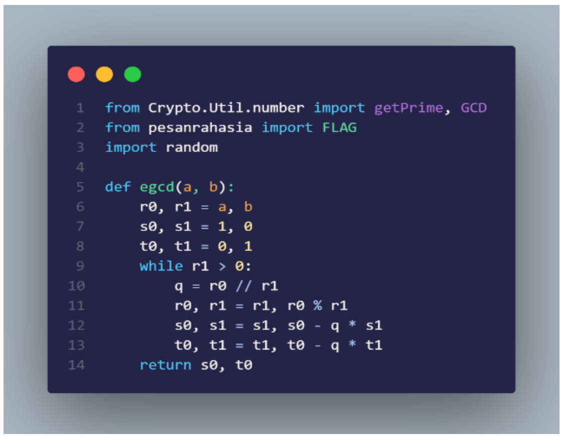
IV. Implementation

This program is developed using Python as its primary programming language due to its simplicity and versatility in mathematical processing. The libraries included are *Crypto.Util* to help with RSA process, *random* to get the random primes, and *time* to calculate the time taken to encrypt or decrypt the message.

This program also developed using Sage as its framework due to its efficiency and capability of handling big numbers and its calculations. Several limitations have been incorporated into the implementation to ensure its feasibility. The limitations are it can't be done if the plaintext is too long due to the python constraints.

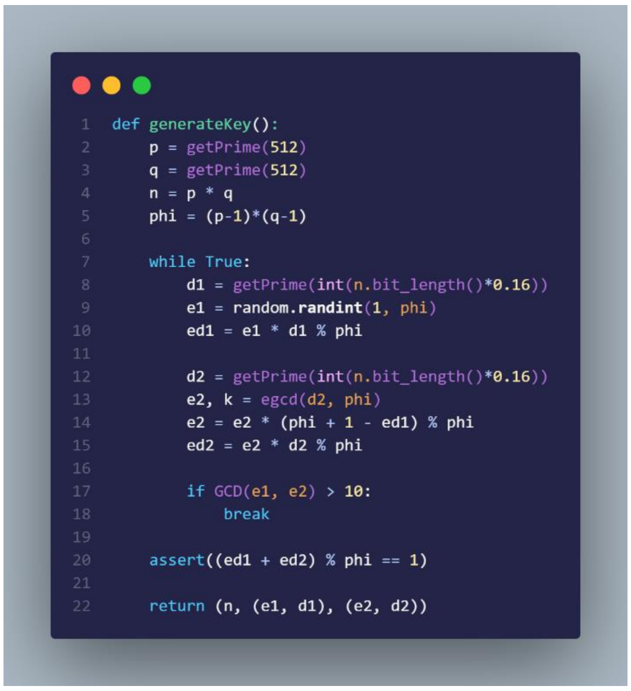
The source code of this program can be accessed in appendix section.

A. Encryption



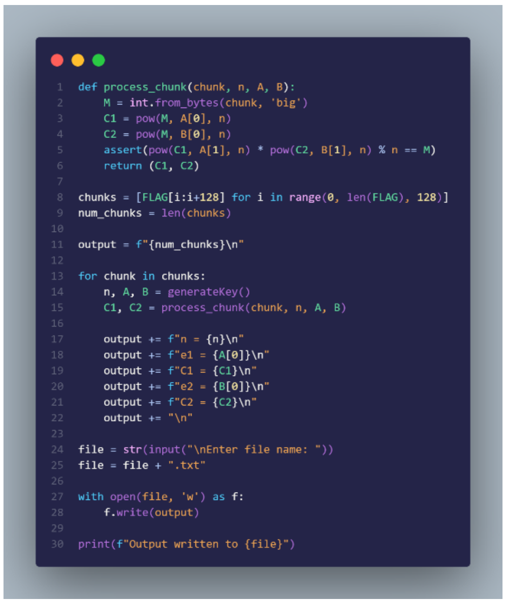
```python
from Crypto.Util.number import getPrime, GCD
from pesanrahasia import FLAG
import random

def egcd(a, b):
    r0, r1 = a, b
    s0, s1 = 1, 0
    t0, t1 = 0, 1
    while r1 > 0:
        q = r0 // r1
        r0, r1 = r1, r0 % r1
        s0, s1 = s1, s0 - q * s1
        t0, t1 = t1, t0 - q * t1
    return s0, t0
```

Figure 4.1 *Source code of egcd: encryption process*
(Source: writer's archive)



```python
def generateKey():
    p = getPrime(512)
    q = getPrime(512)
    n = p * q
    phi = (p-1)*(q-1)

    while True:
        d1 = getPrime(int(n.bit_length()*0.16))
        e1 = random.randint(1, phi)
        ed1 = e1 * d1 % phi

        d2 = getPrime(int(n.bit_length()*0.16))
        e2, k = egcd(d2, phi)
        e2 = e2 * (phi + 1 - ed1) % phi
        ed2 = e2 * d2 % phi

        if GCD(e1, e2) > 10:
            break

    assert((ed1 + ed2) % phi == 1)

    return (n, (e1, d1), (e2, d2))
```

Figure 4.2 *Source code of key generator: encryption process*
(Source: writer's archive)

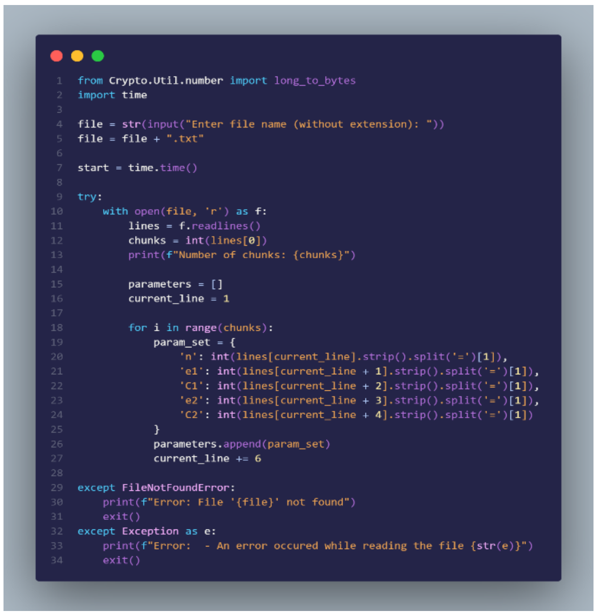


```python
def process_chunk(chunk, n, A, B):
    M = int.from_bytes(chunk, 'big')
    C1 = pow(M, A[0], n)
    C2 = pow(M, B[0], n)
    assert(pow(C1, A[1], n) * pow(C2, B[1], n) % n == M)
    return (C1, C2)

chunks = [FLAG[i:i+128] for i in range(0, len(FLAG), 128)]
num_chunks = len(chunks)

output = f"{num_chunks}\n"

for chunk in chunks:
    n, A, B = generateKey()
    C1, C2 = process_chunk(chunk, n, A, B)

    output += f"n = {n}\n"
    output += f"e1 = {A[0]}\n"
    output += f"C1 = {C1}\n"
    output += f"e2 = {B[0]}\n"
    output += f"C2 = {C2}\n"
    output += "\n"

file = str(input("\nEnter file name: "))
file = file + ".txt"

with open(file, 'w') as f:
    f.write(output)

print(f"Output written to {file}")
```

Figure 4.3 *Source code of chunk processor: encryption process*
(Source: writer's archive)

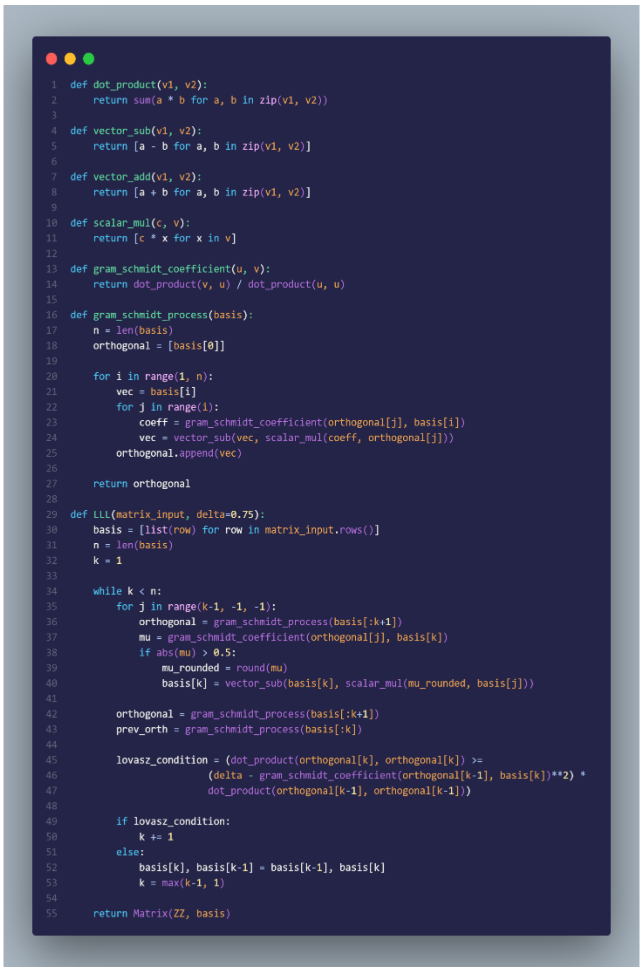
The encryption process is inspired by the SECCON 2020 CTF "sharsable problem". This source code implements the encryption process that had been mentioned in the cryptosystem scheme. Additionally, it incorporates an additional feature, chunk. Chunk used in this process to make the decryption process more efficient and avoid overflow when encrypting the plain text and saved it to txt file. The idea is, when the plain text input is more than 127 bytes, the program will separate it to different chunks, so it will handle the overflow in the process and make the decryption process more efficient.

B. Decryption



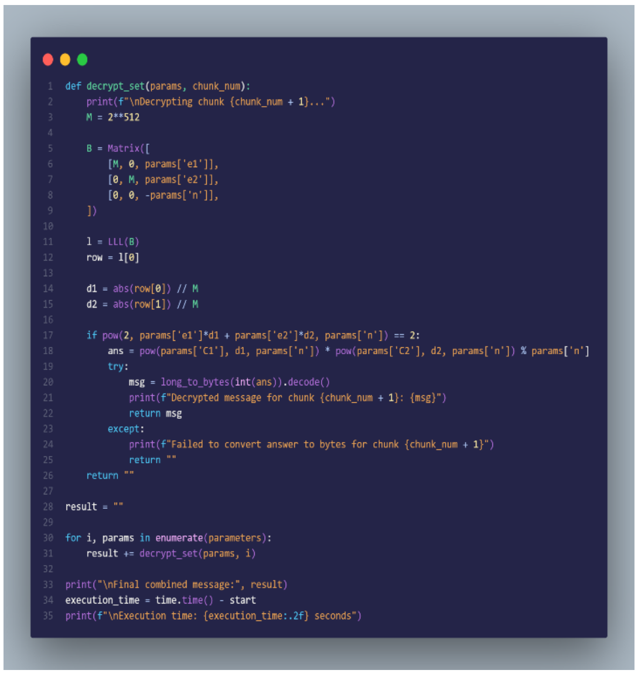
```python
from Crypto.Util.number import long_to_bytes
import time

file = str(input("Enter file name (without extension): "))
file = file + ".txt"

start = time.time()

try:
    with open(file, 'r') as f:
        lines = f.readlines()
        chunks = int(lines[0])
        print(f"Number of chunks: {chunks}")

        parameters = []
        current_line = 1

        for i in range(chunks):
            param_set = {
                'n': int(lines[current_line].strip().split('=')[1]),
                'e1': int(lines[current_line + 1].strip().split('=')[1]),
                'C1': int(lines[current_line + 2].strip().split('=')[1]),
                'e2': int(lines[current_line + 3].strip().split('=')[1]),
                'C2': int(lines[current_line + 4].strip().split('=')[1])
            }
            parameters.append(param_set)
            current_line += 6

except FileNotFoundError:
    print(f"Error: File '{file}' not found")
    exit()
except Exception as e:
    print(f"Error:  - An error occured while reading the file {str(e)}")
    exit()
```

Figure 4.4 *Source code of read file: decryption process*
(Source: writer's archive)



```python
def dot_product(v1, v2):
    return sum(a * b for a, b in zip(v1, v2))

def vector_sub(v1, v2):
    return [a - b for a, b in zip(v1, v2)]

def vector_add(v1, v2):
    return [a + b for a, b in zip(v1, v2)]

def scalar_mul(c, v):
    return [c * x for x in v]

def gram_schmidt_coefficient(u, v):
    return dot_product(v, u) / dot_product(u, u)

def gram_schmidt_process(basis):
    n = len(basis)
    orthogonal = [basis[0]]

    for i in range(1, n):
        vec = basis[i]
        for j in range(i):
            coeff = gram_schmidt_coefficient(orthogonal[j], basis[i])
            vec = vector_sub(vec, scalar_mul(coeff, orthogonal[j]))
        orthogonal.append(vec)

    return orthogonal

def LLL(matrix_input, delta=0.75):
    basis = [list(row) for row in matrix_input.rows()]
    n = len(basis)
    k = 1

    while k < n:
        for j in range(k-1, -1, -1):
            orthogonal = gram_schmidt_process(basis[:k+1])
            mu = gram_schmidt_coefficient(orthogonal[j], basis[k])
            if abs(mu) > 0.5:
                mu_rounded = round(mu)
                basis[k] = vector_sub(basis[k], scalar_mul(mu_rounded, basis[j]))

        orthogonal = gram_schmidt_process(basis[:k+1])
        prev_orth = gram_schmidt_process(basis[:k])

        lovasz_condition = (dot_product(orthogonal[k], orthogonal[k]) >=
                            (delta - gram_schmidt_coefficient(orthogonal[k-1], basis[k])**2) *
                            dot_product(orthogonal[k-1], orthogonal[k-1]))

        if lovasz_condition:
            k += 1
        else:
            basis[k], basis[k-1] = basis[k-1], basis[k]
            k = max(k-1, 1)

    return Matrix(ZZ, basis)
```

Figure 4.5 *Source code of lattice basis reduction: decryption process*
(Source: writer's archive)



```python
def decrypt_set(params, chunk_num):
    print(f"\nDecrypting chunk {chunk_num + 1}...")
    M = 2**512

    B = Matrix([
        [M, 0, params['e1']],
        [0, M, params['e2']],
        [0, 0, -params['n']],
    ])

    l = LLL(B)
    row = l[0]

    d1 = abs(row[0]) // M
    d2 = abs(row[1]) // M

    if pow(2, params['e1']*d1 + params['e2']*d2, params['n']) == 2:
        ans = pow(params['C1'], d1, params['n']) * pow(params['C2'], d2, params['n']) % params['n']
        try:
            msg = long_to_bytes(int(ans)).decode()
            print(f"Decrypted message for chunk {chunk_num + 1}: {msg}")
            return msg
        except:
            print(f"Failed to convert answer to bytes for chunk {chunk_num + 1}")
            return ""
    return ""

result = ""

for i, params in enumerate(parameters):
    result += decrypt_set(params, i)

print("\nFinal combined message:", result)
execution_time = time.time() - start
print(f"\nExecution time: {execution_time:.2f} seconds")
```

Figure 4.6 *Source code chunk decryption process*
(Source: writer's archive)

This source code implements the decryption process that had been mentioned in the cryptosystem scheme. Additionally, it incorporates an additional feature, execution time. Execution time used in this process to observe time changes corresponding to each test case variation.

V. Test and Result

To test the implementation of the program, there would be 3 test cases used: short text, medium text, and long text.

Table 5.1 Test Cases for Implementation

Parameters	Filename		
	Test.txt	Winter.txt	Danger.txt
types	short	medium	Long
chunks	1	2	4
Size of number searched	1024 bits	2048 bits	4096 bits

For the test cases and the result, it also attached in the appendix section, the input file will be at the input folder.

To compare the time needed to decode from lattice method, this paper used general number field sieve to compare the time needed to decrypt the ciphertext, the code used to calculate the sieve time is attached in the appendix section.

This comparison is assumed we have computer that can do 3×10^9 operations per core, with 1000 cores inside the computer and with the program's time complexity of $L(n) = exp((c + o(1))(ln\ n)\text{^}(1/3)(ln\ ln\ n)\text{^}(2/3))$. Furthermore, writer also calculate the improvements from sieve to lattice methods. The formula to calculate the improvements are $\frac{sieve}{lattice}(seconds)$.

Table 5.2 Results for Implementation

Filename	Time-Lattice (seconds)	Time-Sieve (seconds)	Improvement (times)
Test.txt	0.10	4.39×10^{13}	4.39×10^{14}
Winter.txt	0.19	5.11×10^{22}	5.79×10^{23}
Danger.txt	0.35	4.30×10^{34}	8.57×10^{34}

Notice that if the plaintext gets longer, the time required will also increase, the time required with sieve methods will increase greatly, but the time required with lattice methods is not. So, with lattice methods, we can efficiently attack small private key RSA encryption in no time.

VI. Conclusion

With Lenstra-Lenstra-Lovász algorithm and Minkowski's theorem, we can simply attack RSA with small private key, searching for its prime factors using the reduced lattice form. LLL calculated the smallest possible vectors to search for the possible private key value, and

Minkowski's first theorem make the bound for the LLL computation, so we can efficiently find the right vectors. With these methods, we can greatly reduce the time required to decrypt the encrypted plaintext from years to less than a seconds.

This study lays the groundwork for further development. The current program developed is exclusively encrypt and decrypt small size of text. Furthermore, the program's limitation to certain file types, and plaintext size could be broadened, allowing for more extensive file and text size compatibility.

VII. Appendix

The program that used in this paper can be seen in this link:

https://github.com/Nayekah/Lattice

VIII. Acknowledgment

All praise and gratitude belong to the Almighty God, Allah Subhanahu wa Ta'ala, for his blessings and grace, enable the writer to complete this paper. The writer also giving sincere thanks to Dr. Ir. Rinaldi Munir, M.T., the lecturer for the IF2123 - linear and geometrical algebra for his guidance and kindness to the writer. And the writer also appreciates for author's families and friends for their motivational support throughout the process of finishing this paper.

References

[1] Surin Joseph, C. Shaanan "A Gentle Tutorial for Lattice-Based Cryptanalisis", https://eprint.iacr.org/2023/032.pdf, 2023, accessed 31st January 2024, 18.33 UTC+7.

[2] Geeksforgeeks, "Symmetric Key Cryptography", https://www.geeksforgeeks.org/symmetric-key-cryptography/, 2024, accessed 31st January 2024, 14.24 UTC+7.

[3] Geeksforgeeks, "Asymmetric Key Cryptography", https://www.geeksforgeeks.org/asymmetric-key-cryptography/, 2024, accessed 31st January 2024, 14.24 UTC+7.

[4] JaneW, "Cryptosystem Model", https://uwillnvrknow.github.io/deCryptMe/pages/cryptosystem.html, 2024, accessed 31st January 2024, 13.33 UTC+7.

[5] M. Rinaldi, "Algoritma RSA", https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2020-2021/Algoritma-RSA-2020.pdf, 2020, accessed 31st January 2024, 16.07 UTC+7.

[6] D. Boneh, G. Durfee, "Cryptanalysis of RSA with Private Key d Less than $N^{0.292}$," in Advances in Cryptology — EUROCRYPT '99, Ed. Berlin: Springer, 1999, pp. 1-11.

[7] L. Babai, "On Lovász' lattice reduction and the nearest lattice point problem," in Combinatorica, 6.1, Ed. New York: Springer, 1986, pp. 1-13.

[8] R. Kannan, "Minkowski's Convex Body Theorem and Integer Programming," in Mathematics of Operations Research, 12.3, Ed. Online: Informs, 1987, pp. 415-440.

Statement of Originality

I hereby declare that this paper is my own writing, not an adaptation, or translation of someone else's paper, and not plagiarized.

Bandung, 02 January 2025

[illegible]

Nayaka Ghana Subrata
13523090