.

RSA ENCRYPTION USING SAGEMATH

Project report submitted
in partial fulfillment of the requirement for the degree of

Bachelor of Engineering in Information Technology

By

Sayan Das (302211001006)
Saugata Ghosh (302211001007)
Suvajit Sadhukhan (302211001005)
Subhankar Das (002111001147)

Under the guidance of

Mr. Utpal Kumar Ray
Assistant Professor (Contractual)

Department of Information Technology,
Faculty of Engineering and Technology,
Jadavpur University, Salt Lake Campus
2024-2025

BONAFIDE CERTIFICATE

This is to certify that this project report entitled **"RSA ENCRYPTION USING SAGEMATH"** submitted to **Department of Information Technology, Jadavpur University, Salt Lake Campus, Kolkata**, is a bonafide record of work done by **Sayan Das (Registration No: 165915 of 2022-2023), Saugata Ghosh (Registration No: 165916 of 2022-2023), Suvajit Sadhukhan (Registration No: 165914 of 2022-2023), Subhankar Das (Registration No: 158864 of 2021-2022)** under my supervision from **09/07/2024** to **15/05/2025.**

Mr. Utpal Kumar Ray
Assistant Professor (Contractual)

Countersigned By:

Prof. Bibhas Chandra Dhara
Head of Department
Department of Information Technonogy

Place : Kolkata
Date : 15.05.2025

DECLARATION

We hereby declare that this report is an original work created entirely by us. It contains no plagiarized content from external sources. Whenever information from external references has been incorporated, it has been duly acknowledged and cited. We accept full responsibility for any instances of plagiarism that may be identified in this report. Moreover, all non-original elements, including information, materials, and methodologies, have been appropriately cited and referenced. Lastly, we confirm that this project has not been submitted previously to fulfill the requirements of any other academic degree.

Sayan Das
Roll no. 302211001006

Saugata Ghosh
Roll no. 302211001007

Suvajit Sadhukhan
Roll no. 302211001005

Subhankar Das
Roll no. 002111001147

Acknowledgments

We wish to express our sincere gratitude to all those whose support and guidance were instrumental in the successful completion of this project. First and foremost, we extend our heartfelt thanks to Mr. Utpal Kumar Ray, our project supervisor, for granting us the opportunity to undertake this endeavour and for his unwavering support throughout.

We are also deeply thankful to Mr. Sujay Kumar Paul for his exceptional mentorship. His patience and insightful guidance helped us overcome challenges and uncertainties, ensuring we stayed focused and motivated. The biweekly sessions under his expert tutelage provided clarity and direction, playing a pivotal role in maintaining our progress.

Furthermore, we are grateful to the University and the Department of Information Technology for their consistent support and for providing us with the necessary resources to carry out this project effectively. We also extend our appreciation to the technical staff of the software labs, whose efforts ensured seamless operations of the systems we utilized.

Finally, we would like to express our profound gratitude to our parents for their steadfast encouragement and emotional support. Their belief in our abilities has been a constant source of strength and inspiration throughout this journey.

Department of Information Technology
Jadavpur University
Salt Lake Campus, Kolkata

Sayan Das.
Roll no. 302211001006

Saugata Ghosh
Roll no. 302211001007

Suvajit Sadhukhan
Roll no. 302211001005

Subhankar Das
Roll no. 002111001147

JADAVPUR UNIVERSITY
Dept. of Information Technology

Vision:

To provide young undergraduate and postgraduate students a responsive research environment and quality education in Information Technology to contribute in education, industry and society at large.

Mission:

M1: To nurture and strengthen professional potential of undergraduate and postgraduate students to the highest level.

M2: To provide international standard infrastructure for quality teaching, research and development in Information Technology.

M3: To undertake research challenges to explore new vistas of Information and Communication Technology for sustainable development in a value-based society.

M4: To encourage teamwork for undertaking real life and global challenges.

Program Educational Objectives (PEOs):

Graduates should be able to:

PEO1: Demonstrate recognizable expertise to solve problems in the analysis, design, implementation and evaluation of smart, distributed, and secured software systems.

PEO2: Engage in the engineering profession globally, by contributing to the ethical, competent, and creative practice of theoretical and practical aspects of intelligent data engineering.

PEO3: Exhibit sustained learning capability and ability to adapt to a constantly changing field of Information Technology through professional development, and self-learning.

PEO4: Show leadership qualities and initiative to ethically advance professional and organizational goals through collaboration with others of diverse interdisciplinary backgrounds.

Mission - PEO matrix:

Ms/ PEOs	M1	M2	M3	M4
PEO1	3	2	2	1
PEO2	2	3	2	1
PEO3	2	2	3	1
PEO4	1	2	2	3

(3 – Strong, 2 – Moderate and 1 – Weak)

Program Specific Outcomes (PSOs):

At the end of the program a student will be able to:

PSO1: Apply the principles of theoretical and practical aspects of ever evolving Programming & Software Technology in solving real life problems efficiently.

PSO2: Develop secured software systems considering constantly changing paradigms of communication and computation of web enabled distributed Systems.

PSO3: Design ethical solutions of global challenges by applying intelligent data science & management techniques on suitable modern computational platforms through interdisciplinary collaboration.

Abstract

This project implements RSA encryption and decryption using SageMath. RSA is a widely used public-key cryptosystem that ensures secure communication by encrypting plaintext into ciphertext and decrypting it back using a pair of keys. The project includes scripts for key generation, encryption, and decryption, along with a benchmarking and testing framework. The implementation demonstrates the practical aspects of cryptography, including secure key management, block-based encryption, and performance evaluation.

Keywords**:** RSA, public-key cryptography, encryption, decryption, SageMath, factoring attacks, timing attacks, chosen ciphertext attacks (CCA), low public exponent attacks, side-channel attacks, poor random number generation (RNG).

TABLE OF CONTENTS

1 Introduction 1
1.1 Motivation 1
1.2 Research Goal and Contribution 1
1.2.1 Research Goal 1
1.2.2 Research Contribution 1
1.3 Organization of the project 1

2 Related Works (Literature Review) 2

3. Basic Concepts and Technology Used 4
3.1 RSA 4
3.1.1 Prime Numbers and Their Role in RSA 4
3.1.2 Pair Public and Private Key 5
3.1.3 Modular Arithmetic in RSA 6
3.2 Technologies Used 6
3.2.1 Basic concepts on SageMath 6

4. Implementation. 8
4.1 Key Generation 8
4.1.1 Generating Large Prime Numbers 9
4.1.2 Computing the Modulus (n) 9
4.1.3 Euler's Totient Function (ϕ) 9
4.1.4 Choosing the Public Exponent (e) 10
4.1.5 Computing the Private Exponent (d) 11
4.2 Encryption and Decryption 13
4.2.1 RSA Encryption Process 13
4.2.2 RSA Decryption Process 16
4.2.3 Block Size and Padding 19
4.3 Relevant Scripts 21
4.3.1 Key Generation Script 21
4.3.2 Encryption Script 22
4.3.3 Decryption Script 24

5. Testcases and Results 26
5.1 Testing RSA Functionality 26
5.2 Benchmarking RSA Performance 27

6. Security Analysis of RSA: Types of Attacks and Defence Mechanisms 29
6.1 Security of RSA 29
6.1.1 Difficulty of Factoring Large Integers 29
6.1.2 Importance of Key Size 29
6.2 Common Attacks on RSA 30
6.2.1 Factoring Attacks 30
6.2.2 Timing Attacks 30
6.2.3 Chosen Ciphertext Attacks 30
6.2.4 Low Public Exponent Attacks 31
6.2.5 Common Modulus Attacks 31
6.2.6 Implementation Error / Side-Channel Attacks 31
6.2.7 Poor Random Number Generation 32

7. Conclusion and Future Work 33

References 35

Appendix I 36

Appendix II 37

LIST of SYMBOLS, ABBREVIATIONS and NOMENCLATURE

Symbol / Term	***Meaning / Description***
RSA	Rivest–Shamir–Adleman public-key cryptosystem
p	First large prime number used for RSA key generation
q	Second large prime number used for RSA key generation
n	Modulus for RSA, $n = p \times q$
e	Public exponent used for encryption
d	Private exponent used for decryption
$\phi(n)$	Euler's Totient function of n, used in key calculation
m	Message represented as an integer
c	Ciphertext after encryption
m'	Decrypted message (recovered original message)
gcd(a, b)	Greatest Common Divisor of two integers a and b
modular inverse	A number d such that $e \times d \equiv 1\ (mod\ \phi(n))$
modular exponentiation	Efficient computation of $b^e mod m$
SageMath	Open-source computational mathematics software system
Python	Programming language used for scripting and implementation
randint(a, b)	SageMath/Python function to generate a random integer between a and b
next_prime(n)	SageMath function returning the next prime greater than n
.nbits()	SageMath method returning number of bits needed to represent an integer
inverse_mod(a, m)	SageMath function computing the modular inverse of $a\ (mod\ m)$
power_mod(b, e, m)	SageMath function for fast modular exponentiation, use to calculate $b^e\ \ (mod\ m)$
PKCS#1, OAEP	Padding standards used to secure RSA encryption
block_size	Maximum data size that can be encrypted in one RSA operation
data_size	Number of bytes available for actual message inside a block (block_size - padding)
padding	Extra bytes added to plaintext before encryption to prevent attacks
L	Message length stored in 1-byte header in each block
ciphertext	Encrypted form of the original message
plaintext_bytes	Encoded bytes of the plaintext string
block_bytes	Bytes structure containing message, header, and padding before encryption
os.urandom()	Python function used to generate random bytes
encoding="utf-8"	Standard encoding format used for text data transformation
base, ext	Variables used to split file names and extensions during file operations

1. Introduction

1.1 Motivation

Cryptography is a cornerstone of digital security, enabling confidential communication and safeguarding sensitive data. As technology evolves and more personal, financial, and governmental information is exchanged digitally, the need for secure encryption systems becomes increasingly critical. RSA (Rivest–Shamir–Adleman), one of the earliest public-key cryptosystems, remains a foundational tool in securing digital communications. It leverages mathematical concepts such as number theory and modular arithmetic to provide secure key exchange and message encryption. This project is motivated by the importance of understanding and implementing robust cryptographic systems, particularly in an era where data breaches and cyberattacks are on the rise.

1.2 Research Goal and Contribution

1.2.1 Research Goal

The core objective of this research is to deeply understand the RSA algorithm by implementing it programmatically, evaluating its performance, and analyzing its security against modern threats. By building an RSA encryption system from scratch, the project aims to provide a hands-on understanding of the core cryptographic principles and demonstrate how mathematical theory translates into real-world data protection.

1.2.2 Research Contribution

This thesis contributes to the field of cryptography in the following ways:

- A practical implementation of RSA using Python.
- Detailed walkthrough of key generation, encryption, and decryption processes.
- Performance benchmarking with varying key sizes to understand computational efficiency.
- Exploration of known attack vectors and analysis of RSA's resilience against them.

1.3 Organization of the Project

This project is structured into the following chapters:

- *Chapter 2* reviews related literature and existing work on RSA and public-key cryptography.
- *Chapter 3* introduces the mathematical foundations of RSA and the tools used.
- *Chapter 4* details the implementation process.
- *Chapter 5* presents test cases and performance results.
- *Chapter 6* provides a security analysis.
- *Chapter 7* concludes the research and outlines future directions.

2. Related Works (Literature Review)

Sl No.	Article Title	Author	Journal/ Source Details	Technology/ Algorithms	Result	Issues	Year of Publication
1	Systematic and Critical Review of RSA Based Public Key Cryptographic Schemes: Past and Present Status	S. A. A. Shah, M. A. Gondal, M. Hussain	ResearchGate (Link)	Review of RSA enhancements	Highlighted advancements and persistent vulnerabilities in RSA	Emphasized need for continuous improvement to counteract emerging threats	2021
2	Methods toward Enhancing RSA Algorithm: A Survey	A. A. A. Yousif, M. A. Maarof	SSRN Electronic Journal (SSRN)	RSA Enhancement Techniques	Surveyed various methods proposed to enhance RSA security	Not all methods are practical for real-world applications	2019
3	Post-Quantum RSA	Daniel J. Bernstein, Nadia Heninger, Paul Lou, Luke Valenta	IACR Cryptology ePrint Archive(Link)	Analysis of RSA in the context of quantum computing	Explored feasibility of RSA parameters resistant to quantum attacks	Highlighted impracticality of large key sizes required for post-quantum security	2017
4	An Enhanced Version of RSA to Increase the Security	Ritu Patidar, Rupali Bhartiya	Journal of Network Communications and Emerging Technologies (Link)	Modified RSA Algorithm with three prime numbers	Improved security by introducing a third prime in key generation	Increased computational overhead due to additional prime	2017
5	A Literature Review of Some Modern RSA Variants	Akansha Tuteja, Amit Shrivastava	International Journal for Scientific Research & Development (IJSRD)	Variants of RSA Algorithm	Reviewed modern adaptations of RSA to enhance security and performance	Some variants may introduce implementation complexity	2014
6	A Study and Performance Analysis of RSA Algorithm	S. Sowjanya, K. Srinivasa Rao	International Journal of Computer Science and Mobile Computing (IJCSMC)	RSA Algorithm	Analysed execution time of RSA with varying key sizes	Performance impact with increased key sizes; lacks discussion on modern optimizations	2013
7	Analysis and Research of the RSA Algorithm	Zhang, Cao	Information Technology Journal (Science Alert)	RSA Algorithm	Discussed RSA's mathematical foundation and security aspects	Lacked empirical performance analysis	2013

Sl No.	Article Title	Author	Journal/ Source Details	Technology/ Algorithms	Result	Issues	Year of Publication
8	The RSA Cryptosystem	Paar, Pelzl	Understanding Cryptography (Link)	RSA Algorithm	Detailed explanation of RSA operations and security considerations	Focused on theoretical aspects; lacks implementation details	2010
9	Twenty Years of Attacks on the RSA Cryptosystem	Dan Boneh	Notices of the AMS (Link)	Review of attack models on RSA	Comprehensive catalog of attacks; emphasized importance of padding and key length	Highlighted vulnerabilities in implementations lacking padding and proper key management	1999
10	Handbook of Applied Cryptography	Menezes, Van Oorscho, Vanstone	CRC Press (Link)	Cryptographic algorithms including RSA, DSA, and ECC	Presented mathematical foundation and implementation advice	Theoretical focus, lacks performance benchmarking for modern RSA implementations	1996
11	A Method for Obtaining Digital Signatures and Public-Key Cryptosystems	Rivest, Shamir, Adleman	Communications of the ACM (Link)	RSA Algorithm	Proposed RSA, the first practical public-key cryptosystem	Lacked resistance to side-channel attacks due to lack of implementation details	1978

3. Basic Concepts and Technology Used

This chapter elaborates on the mathematical foundations and the technological tools used in developing our RSA-based cryptographic project. RSA (Rivest–Shamir–Adleman) is grounded in principles of number theory, particularly those concerning prime numbers and modular arithmetic. Our implementation focuses on key generation, encryption, and decryption using Python and SageMath.

3.1 RSA

RSA is a public-key cryptosystem that facilitates secure communication by using a pair of mathematically linked keys: two keys: a **public key** for encryption and a **private key** for decryption. Its security is based on the computational difficulty of factoring the product of two large prime numbers. It is widely used in digital signatures, secure email, and web encryption protocols like HTTPS. RSA's security is grounded in the mathematical difficulty of factoring large integers.

3.1.1 Prime Numbers and Their Role in RSA

What Are Prime Numbers?
Prime numbers are integers greater than 1 that have no divisors other than 1 and themselves. For example, 2, 3, 5, 7, and 11 are prime numbers. Their distribution becomes less frequent as numbers grow larger, but finding large prime numbers is crucial for cryptography.

Why Prime Numbers?
Prime numbers have properties that contribute to RSA's security:
1. Unique Factorization : The Fundamental Theorem of Arithmetic ensures that every integer has a unique factorization into primes, which is pivotal for RSA's design.
2. Difficulty of Factoring : Factoring large numbers into their prime components is computationally expensive, especially when n is hundreds or thousands of bits long.

Role of Prime Numbers in RSA
- Prime numbers are crucial to RSA. We start by generating two large random prime numbers p and q, and their product (n) becomes the modulus for both the public and private keys.
  $$n = p \times q$$
- The security of RSA depends on the difficulty of factoring n back into p and q. This is known as the factoring problem, and for sufficiently large n, it becomes computationally infeasible.

Euler's Totient Function :
Euler's Totient Function, denoted as $\phi(n)$, is a fundamental concept in number theory. It plays a crucial role in RSA encryption by calculating the number of integers less than n that are co-prime (relatively prime) to n, i.e.,
$\phi(n) =$ Count of integers k such that $1 \leq k < n$ and $\gcd(k, n) = 1$

Calculation of $\phi(n)$:

1. If n is prime, all integers 1,2,…, $n-1$ are co-prime to n. Hence:

$$\phi(n) = n - 1$$

2. If n is Composite (product of primes):

$$\phi(n) = n \cdot \left(1 - \frac{1}{p_1}\right) \cdot \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_k}\right),$$

where, $p_1, p_2, \ldots, p_k$ are the distinct prime factors of n.
Alternatively, when n is expressed as $p \cdot q$ (for RSA):

$$\phi(n) = (p-1)(q-1) = \phi(p) \cdot \phi(q)$$

Role in RSA :
In RSA, $\phi(n)$ is used to compute the private key. It ensures the encryption and decryption keys are mathematically linked:

As a simple example,
let $p = 61$ and $q = 53$
Compute the modulus:

$$n = p \times q = 61 \times 53 = 3233$$

Compute Euler's Totient Function for $n = 3233$,

$$\phi(n) = (p-1)(q-1)$$
$$\Rightarrow \phi(3233) = (61-1)(53-1) = 60 \times 52 = 3120.$$

3.1.2 Public and Private Key Pair

The public key consists of an exponent (e) and modulus (n) and is used for encryption. The private key (d, n) is used for decryption. These values are linked through Euler's Totient function and modular inverse calculations, making it practically impossible to derive one from the other without factoring n.

Once p and q are known:
1. Compute : $n = p \times q$
2. Compute Euler's Totient : $\phi(n) = (p-1)(q-1)$
3. Choose e such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$, typically
$e = 3, 17, ..., 65537 \ (2^{16}+1), \ldots 4.$

Compute, $d = e^{-1} \mod \phi(n)$
If $e \cdot d \equiv 1 \mod \phi(n)$
Then d is the modular inverse of e modulo $\phi(n)$.

Continuing with our example :
- $p = 61$ *and* $q = 53$
- $\phi(n) = (p-1)(q-1) = (61-1)(53-1) = 60 \times 52 = 3120$
- Choose public exponent e such that $1 < e < \phi(n)$, and $gcd(e, \phi(n)) = 1$. let $e = 17$.
- Compute the private exponent d, such that:
  $d \equiv e^{-1} \mod \phi(n)$
  $d \equiv 17^{-1} \mod 3120 = 2753$

Now (e, n) is the public key and (d, n) is the private key.

3.1.3 Modular Arithmetic in RSA

Modular arithmetic allows for calculations where numbers wrap around after reaching a certain value (modulus). RSA uses modular exponentiation for encryption ($m^e \mod n$) and decryption ($c^d \mod n$), which are computationally efficient yet secure against reverse computation.

RSA encryption and decryption use modular exponentiation:

1. **Public Key Encryption:**
   - A public key is generated, consisting of e (the public exponent) and n (the modulus).
   - To encrypt a message m (where $m < n$), modular exponentiation is performed:

$$c = m^e \mod n$$

Here, c is the cipher text, which is sent securely to the receiver. Modular arithmetic ensures the result c stays within the range of n.

2. **Private Key Decryption:**
   - The receiver uses their private key d to decrypt the ciphertext c:

$$m = c^d \mod n$$

This restores the original message m by applying the modular inverse relationship. RSA ensures that $e \cdot d \equiv 1 \mod \phi(n)$, where $\phi(n)$ is Euler's Totient Function of n.

Why Modular Arithmetic Works

1. **Efficiency:** Modular exponentiation is computationally efficient for large numbers, which is vital for RSA as n, e, and d are often hundreds or thousands of bits long.
2. **Security:** Modular arithmetic combined with the properties of large prime numbers makes reversing the encryption process (without the private key) computationally infeasible.
3. **Uniqueness:** Thanks to modular arithmetic, every message produces a unique cipher-text under the public key, reducing the risk of ambiguity.

3.2 Technologies Used

To implement the RSA algorithm effectively and securely, our project combines the power of **Python** for practical scripting and **SageMath** for computational mathematics. This section outlines the tools used, how SageMath integrates into our workflow, examples of its usage, and how to set it up for development.

3.2.1 Basic Concepts on SageMath

SageMath (or Sage) is an open-source mathematical software system that integrates several powerful libraries including GMP, PARI/GP, NTL, and SymPy. It provides a Python-based interface and supports advanced computations in number theory, cryptography, algebra, and symbolic mathematics. It simplifies the generation of large primes, execution of modular arithmetic, and algorithm validation. For instance, it enables fast implementation of the Extended Euclidean Algorithm and Fermat's primality test, which are key components of RSA key generation.

SageMath is a robust mathematics engine ideal for number theory and cryptographic applications like RSA due to its:

3. Built-in support for prime verification and generation
4. High-performance modular arithmetic
5. Easy-to-use interface for symbolic algebra
6. Seamless integration with Python scripts

Relevance in our Project:
Our project involves operations like:
- Large prime generation: p and q of a fixed bit size
- Modular exponentiation
- Modular inverse calculation
- GCD computations for co-primality checks

SageMath provides the exact tools we need to ensure these operations are fast and accurate, especially when dealing with large integers.

Setting Up SageMath

You can run SageMath in three ways:
1. Online : https://sagecell.sagemath.org
2. Local Installation :
   - Download from: https://www.sagemath.org/download.html
   - Installation for Linux/macOS : (bash cmd)

```
sudo apt install sagemath  # On Ubuntu/Debian
brew install sagemath      # On macOS (if supported)
```

   - Windows users can install using WSL (Windows Subsystem for Linux).
3. Jupyter Notebook with Sage Kernel: (bash cmd)

```
sage -n jupyter
```

SageMath Operations Used in Our Project

- *randint(a, b)* : generates a random integer between a and b (inclusive)..
- *next_prime(n)* : is a SageMath function that returns the next prime number greater than input n.
- *.nbits()* : a method that returns the number of bits required to represent the integer.
- *gcd(a, b)* : computes the greatest common divisor (GCD) of two numbers a and b.
- *inverse_mod(a, m)* : computes the modular inverse of a modulo m (ie. $a^{-1} \mod m$).
- *power_mod(b, e, m)* : performs modular exponentiation efficiently (ie. $b^e \mod m$).

Using SageMath in Python Scripts :

```
from sage.all import *
"""
    write code in python and sage

"""
```

Save as example.py

Ran in terminal (bash cmd) :

```
sage example.py <command line argument(s)>
```

4. Implementation

This chapter explains how the RSA algorithm was implemented in our project using SageMath and Python. The focus is on the complete **Key Generation Process**, **Encryption and Decryption** and **Relevant Scripts** as implemented in the *rsa_keygenerator.py*, *rsa_encrypt.py*, *rsa_decrypt.py* script.

4.1 Key Generation

RSA key generation is the foundational part of public-key cryptography. It involves generating two large prime numbers and using them to compute the public and private keys as discussed previous chapter.

Pseudocode:

```
INPUT: bits (e.g., 512, 1024).
OUTPUT: public key (e, n), private key (d, n)

1. Generate large prime p:
   a. p = random number in [2^(bits−1), 2^bits − 1]
   b. If p is not prime, p = next_prime(p)
   c. Repeat until p is a valid prime

2. Generate large prime q (distinct from p):
   a. Repeat steps 1a-1c to generate q
   b. Ensure q != p

3. Compute modulus:
   n = p * q

4. Compute Euler's totient:
   phi_n = (p - 1) * (q - 1)

5. Choose public exponent:
   e = 65537
   While gcd(e, phi_n) != 1:
      e = next_prime(e)

6. Compute private exponent:
   d = modular_inverse(e, phi_n)

7. Store keys:
   public_key = (e, n)
   private_key = (d, n)
```

4.1.1 Generating Large Prime Numbers (p and q)

Objective: Generate two large prime numbers p and q, each of a given bit size.

Method Used:
1. We implemented a custom function ***generate_prime(bits)***.
2. It uses ***randint()*** to generate a random number in the correct bit range.
3. It uses ***next_prime()*** to find the next available prime.
4. It checks that the result has exactly the desired bit size using ***.nbits()***.

Algorithm: generate_prime(bits):

1. Generate a random number *candidate* in the range $[2^{bits-1}, 2^{bits} - 1]$.
2. Find the next prime number $p \geqslant candidate$
   - If *p.nbits()* == bits: Return p
   - Otherwise, repeat this process 1 and 2.

Purpose: Ensure primes are exactly of size bits (e.g., 512, 1024).

Code:

```
def generate_prime(bits):
    while True:
        candidate = randint(2**(bits-1), 2**bits - 1)
        p = next_prime(candidate)
        if p.nbits() == bits:
            return p
```

4.1.2 Computing the Modulus (n)

Once the primes p and q are generated, we compute the modulus n.

$$n = p \times q$$

Purpose: n is part of both the public and private keys.

Note: if p and q is of bit size of 512 bis then n is of 1024 bits. Similarly, if p and q is of 1024 bits then n be of 2048 bits integer.

Significance: It is used in the encryption and decryption operations $m^e \mod n$ and $c^d \mod n$.

4.1.3 Euler's Totient Function $\phi(n)$

Euler's Totient is the count of integers less than n that are co-prime to n.
For RSA, we compute:
$\phi(n) = (p - 1)(q - 1)$

Purpose: $\phi(n)$ is needed to compute the private key and to ensure the public exponent is co-prime to it.

Code:

```
phi = (p - 1) * (q - 1)
```

4.1.4 Choosing the Public Exponent (e)

The public exponent e must be:
- A prime number
- $1 < e < \phi(n)$
- $\gcd(e, \phi(n)) = 1$

Common Choice:
We chose $e = 65537$, a standard Fermat prime.

Verification and Adjustment:
If $\gcd(e, \phi(n)) \neq 1$, the code adjusts e using *next_prime()* until it satisfies the condition.

Algorithm :
- Assign $e = 65537$ (a commonly used value for RSA due to its efficiency and security properties).
- Check the greatest common divisor (gcd) of e and $\phi(n)$.
  If $\gcd(e, \phi(n)) = 1$: e is valid. Stop and return e
  Otherwise: Proceed to the next step.
- While $\gcd(e, \phi(n)) \neq 1$:
  Replace e with the next prime greater than the current value of e (using a function like *next_prime(e)*)
- Repeat until $\gcd(e, \phi(n)) = 1$.

Code:

```
e = 65537
if gcd(e, phi) != 1:
    while gcd(e, phi) != 1:
        e = next_prime(e)
```

Why 65537 ?

The number 65537 (which is $2^{16} + 1$) is one of the most commonly used values for the RSA public exponent e, and here's why it strikes a perfect balance between security and performance.

1. Mathematical Properties of 65537
- It is a prime number.
- It is a Fermat number (specifically $F_4 = 2^{2^4} + 1$).
- It has only two 1's in its binary representation:
  $65537_{10} = 10000000000000001_2$

This makes modular exponentiation fast, since exponentiation time depends on the number of 1's in the binary form.

2. Security Considerations
- 65537 is large enough to avoid low-exponent attacks (like when $e = 3$ or $e = 17$, which can make RSA vulnerable without padding).
- It's small enough that encryption operations (modular exponentiations with e) are very fast, especially in devices with limited power (e.g., smart cards, mobile phones).

3. Why Not Choose a Very Large e ?

• RSA's security doesn't depend on e being secret or large; it depends on the difficulty of factoring n.

• A very large e would make encryption slower and increase computational overhead, especially during multiple encryptions or validations like in HTTPS or digital signatures.

• Additionally, if e is too close to $\phi(n)$, the modular inverse d may become small and introduce vulnerabilities.

4.1.5 Computing the Private Exponent (d)

The private key exponent d is calculated as the modular inverse of e modulo $\phi(n)$:

$$d \equiv e^{-1} \mod \phi(n)$$

Purpose:

- d is used to decrypt messages encrypted with the public key.
- This ensures $(m^e)^d \equiv m \mod n$.

Code:

```
d = inverse_mod(e, phi)
```

Key Output and Storage

After key computation:

- Public key (e, n) is stored in *public_key.csv*
- Private key (d, n) is stored in *private_key.csv*

Code to write keys:

```
with open("public_key.csv", "w") as pub_file:
    writer = csv.writer(pub_file)
    writer.writerow(["e", "n"])
    writer.writerow([int(e), int(n)])

with open("private_key.csv", "w") as priv_file:
    writer = csv.writer(priv_file)
    writer.writerow(["d", "n"])
    writer.writerow([int(d), int(n)])
```

Efficiency Note:

We also recorded the total time taken to generate the keys using:

```
start_time = time.time()
    """
      key generation code
    """
end_time = time.time()
print("Time taken to generate keys:", end_time - start_time)
```

Command Line Invocation: `sage rsa_keygenerator.py 1024`

```
(base) suvajitsadhukhan@Suvajits-MacBook-Air rsa_final-year-project % sage rsa_keygenerator.py 1024
Keys generated successfully.
Public key stored in public_key.csv
Private key stored in private_key.csv
Time taken to generate keys: 1.444803 seconds
```

public_key.csv

```
e,n
65537,
1925889598309329047412475826671401084146869486503345712099676294878013964863640555499383301943919245913546688479850504453953562240606569917508604556558415247194932862307045399491419894411081112097728599616725110006337099175126436827997152262481399569292301299479371221405016138796242350929889152107083563135733602267352811826394509278999409229391295016429670728244117251929786923554841111221092720120686497738177716300430300116332032036830422687093976196677608360460022266556642569440121081980232560113095508074782442733435639107422191958140689873554488723644312841537027148535176217274944616414707641886725841158910
```

private_key.csv

```
d,n
1518566166932158747792653017670496397826779187093502800226785971498790754051810265132461533351449882537016321679124077515945256606841334066339891433945271627193868121202647173450293138934627809556070957037694564573733121651851372124942522300862091344038861958820049723419328950041098340977849080403872087841373155559410736547119836267084915534678336003944113555184612070505806955836661478147283767070413094015193665224744097947689935262676388058917312231595805054389438444196043354751932731699893134157560131227569404120291678866747762703334276572610484489699022729661567185160928406418777143048214450201818766432049611925889598309329047412475826671401084146869486503345712099676294878013964863640555499383301943919245913546688479850504453953562240606569917508604556558415247194932862307045399491419894411081112097728599616725110006337099175126436827997152262481399569292301299479371221405016138796242350929889152107083563135733602267352811826394509278999409229391295016429670728244117251929786923554841111221092720120686497738177716300430300116332032036830422687093976196677608360460022266556642569440121081980232560113095508074782442733435639107422191958140689873554488723644312841537027148535176217274944616414707641886725841158910
```

Another Instance of Command Line Invocation: `sage rsa_keygenerator.py 512`

```
(base) suvajitsadhukhan@Suvajits-MacBook-Air rsa_final-year-project % sage rsa_keygenerator.py 512
Keys generated successfully.
Public key stored in public_key.csv
Private key stored in private_key.csv
Time taken to generate keys: 0.428171 seconds
```

public_key.csv

```
e,n
65537,
9080193244561187795216045065444379024049609371269582998762931740043898290728707992718692243755957562123569628319222649623354649330741981490539007632947430430624909201375971664846522548084905851918737305002104730649444461031289748950889756406960906096794359135412806760389551362367259843225584312705101962013
```

private_key.csv

```
d,n
2819089856830286526225809374042094999806787034457210511250581688430553710445351809326750249228717648298339430053423611881630179134421885307353055347476712818964647159267067498378289655870472687427066519159279228661360284035174366968298156607047335575473809981817784656452270896400278426258655289871880190656 1,
9080193244561187795216045065444379024049609371269582998762931740043898290728707992718692243755957562123569628319222649623354649330741981490539007632947430430624909201375971664846522548084905851918737305002104730649444461031289748950889756406960906096794359135412806760389551362367259843225584312705101962013
```

4.2 Encryption and Decryption

Once the RSA key pair (public key (e, n) and private key (d, n)) has been generated, the core cryptographic operations of encryption and decryption can be performed. This section details these processes step-by-step, highlighting the algorithms used, the specific SageMath and Python functions involved, and relevant code snippets from the *rsa_encrypt.py* and *rsa_decrypt.py* scripts. It also covers the crucial aspects of handling message data, specifically block sizing and padding.

4.2.1 RSA Encryption Process

RSA encryption transforms a plaintext message (M) into an unreadable cipher-text (C) using the recipient's public key (e, n). Only someone possessing the corresponding private key (d, n) can decrypt the cipher-text.

Algorithm and Implementation:

1. **Read Public Key (e, n) :**

   Algorithm: Open the specified public key CSV file, skip the header, read the row containing e and n, and convert them into SageMath Integer objects for large number arithmetic.

   Functions Used: *open(), csv.reader(), next(), Integer()* (SageMath).

   Code:

```
# Read public key (e, n) from CSV.
try:
    with open(public_key_csv, "r", encoding="utf-8") as f:
        reader = csv.reader(f)
        next(reader)  # skip header
        row = next(reader)
        if len(row) < 2:
            print("Error: Invalid public key file format.")
            sys.exit(1)
        e = Integer(row[0]) # Convert e to SageMath Integer
        n = Integer(row[1]) # Convert n to SageMath Integer
except Exception as ex:
    print("Error reading public key CSV file:", ex)
    sys.exit(1)
```

2. **Read and Encode Plaintext:**

   Algorithm: Open the specified plaintext file, read its entire content, and encode the resulting string into a sequence of bytes using UTF-8 encoding.

   Functions Used: *os.path.exists(), open(), .read(), .encode()*.

   Code:

```
# Read plaintext.
if not os.path.exists(plaintext_filename):
    print("Error: File", plaintext_filename, "does not exist.")
    sys.exit(1)
with open(plaintext_filename, "r", encoding="utf-8") as f:
    plaintext = f.read()
plaintext_bytes = plaintext.encode("utf-8") # Encode to bytes
```

3. **Determine Block and Data Sizes:**

Algorithm: Calculate the maximum number of bytes (*block_size*) that can be safely encrypted based on the bit length of the modulus n. This ensures the integer representation of the block is less than n. Then, calculate the available space for actual message data (*data_size*) within each block, accounting for padding overhead (15 bytes in this implementation).

Functions Used: *.nbits()* (SageMath method for Integer objects), // (integer division).

Code:

```
# Determine block size: ensure m < n.
block_size = (n.nbits() - 1) // 8 # Calculate max bytes per block
# print("Block size:", block_size)
if block_size < 15: # Check if block size is sufficient for padding
    print("Error: Block size too small for padding requirements.")
    sys.exit(1)
data_size = block_size - 15  # Calculate available bytes for
message data
```

4. **Split Plaintext into Chunks:**

Algorithm: Divide the *plaintext_bytes* sequence into a list of smaller byte sequences (*chunks*), where each chunk's length is no more than *data_size*.

Functions Used: Python list comprehension and slicing [:].

Code:

```
# Split plaintext into chunks of size <= data_size.
chunks = [plaintext_bytes[i:i+data_size] for i in range(0,
len(plaintext_bytes), data_size)]
```

5. **Pad Each Chunk:**

Algorithm: For each *chunk*, construct a full *block_bytes* sequence of exactly *block_size* bytes using the custom padding scheme: 1 byte header (length *L*), the chunk itself, intermediate random padding (if *L* < *data_size*), and 14 bytes of random tail padding (see detailed in Section 4.2.3).

Functions Used: *len(), bytes(), os.urandom()*.

Code:

```
# Inside the loop iterating through 'chunks':
L = len(chunk)  # actual message length for this block
# Header: 1 byte indicating L.
header = bytes([L])
# Pad the message chunk (if necessary) to exactly data_size bytes.
if L < data_size:
    pad_len = data_size - L
    pad_bytes = os.urandom(pad_len) # Generate random intermediate
padding
else:
    pad_bytes = b""
# Tail: 14 random bytes.
tail = os.urandom(14) # Generate random tail padding
# Construct the full block.
block_bytes = header + chunk + pad_bytes + tail
```

6. **Convert Padded Block to Integer:**

Algorithm: Convert the *block_bytes* sequence (representing the padded block) into a standard Python integer, interpreting the bytes in big-endian order. Then, convert this Python integer into a SageMath *Integer* object.

Functions Used: *int.from_bytes()* — is used to convert the plaintext message (bytes format) into an integer, *Integer()* (SageMath).

Code:

```
# Convert block bytes to SageMath integer.
m_int = Integer(int.from_bytes(block_bytes, byteorder="big"))
```

7. **Encrypt using Modular Exponentiation:**

Algorithm: Compute the cipher-text integer $c = m^e \ (mod\ n)$ using the SageMath function optimized for modular exponentiation with large numbers. Append the result to a list of encrypted blocks.

Functions Used: *power_mod()* (SageMath).

Code:

```
c_int = power_mod(m_int, e, n) # Perform RSA encryption
encrypted_blocks.append(c_int)
```

8. **Store Ciphertext:**

Algorithm: Open the designated output file and write each computed cipher-text integer (*c_int*) from the *encrypted_blocks* list to the file, one integer per line.

Functions Used: *os.path.splitext(), open(),* f-string formatting*, .write(), int()*.

Code:

```
# Prepare output filename.
base, ext = os.path.splitext(plaintext_filename)
output_filename = f"{base}_cipher{ext}" if ext else
f"{plaintext_filename}_cipher"
with open(output_filename, "w", encoding="utf-8") as f:
    for c in encrypted_blocks:
        # Write each ciphertext (as an integer) on its own line.
        f.write(f"{int(c)}\n") # Write integer to file
```

Command Line Invocation : `sage rsa_encrypt.py message.txt public_key.csv`

```
(base) suvajitsadhukhan@Suvajits-MacBook-Air rsa_final-year-project % sage rsa_encrypt.py message.txt public_key.csv

Encryption complete.
Encrypted file: message_cipher.txt
Time taken to encrypt: 0.000636 seconds
```

4.2.2 RSA Decryption Process

Decryption reverses the encryption process, transforming the cipher-text C back into the original plaintext message M using the recipient's private key (d, n).

Algorithm and Implementation:

1. **Read Private Key (d, n):**

   Algorithm: Open the specified private key CSV file, skip the header, read the row containing d and n, and convert them into SageMath Integer objects.

   Functions Used: *open(), csv.reader(), next(), Integer()* (SageMath).

   Code:

```
# Read private key (d, n) from CSV.
try:
    with open(private_key_csv, "r", encoding="utf-8") as f:
        reader = csv.reader(f)
        next(reader)  # skip header
        row = next(reader)
        if len(row) < 2:
            print("Error: Invalid private key file format.")
            sys.exit(1)
        d = Integer(row[0]) # Convert d to SageMath Integer
        n = Integer(row[1]) # Convert n to SageMath Integer
except Exception as ex:
    print("Error reading private key CSV file:", ex)
    sys.exit(1)
```

2. **Determine Block Size:**

   Algorithm: Calculate the *block_size* based on the modulus n's bit length, ensuring consistency with the encryption process.

   Functions Used: .nbits() (SageMath) .

   Code:

```
# Determine block size.
block_size = (n.nbits() - 1) // 8 # Calculate block size (must
match encryption)
if block_size < 15:
    print("Error: Block size too small.")
    sys.exit(1)
# data_size = block_size - 15 # Not strictly needed for decryption
logic itself
```

3. **Read Cipher-text Integers:**

   Algorithm: Open the specified ciphertext file. Read the file line by line, stripping whitespace, skipping empty lines, and converting each valid line (representing a ciphertext block) into a SageMath *Integer*. Store these integers in a list.

   Functions Used: *os.path.exists(), open(), .strip(), Integer()* (SageMath).

   Code:

```
if not os.path.exists(ciphertext_filename):
    print("Error: File", ciphertext_filename, "does not exist.")
    sys.exit(1)

encrypted_blocks = []
with open(ciphertext_filename, "r", encoding="utf-8") as f:
    for line in f:
        line = line.strip()
        if not line:
            continue
        try:
            c_int = Integer(line) # Convert line to SageMath Integer
            encrypted_blocks.append(c_int)
        except Exception as ex:
            print("Error parsing line:", line)
            sys.exit(1)
```

4. **Decrypt using Modular Exponentiation:**

   Algorithm: For each cipher-text integer *c_int* in the list, compute the corresponding padded message block integer $m = c^d \ (mod\ n)$ using SageMath's modular exponentiation function.

   Functions Used: *power_mod()* (SageMath).

   Code:

```
# Inside the loop iterating through 'encrypted_blocks':
m_int = power_mod(c_int, d, n) # Perform RSA decryption
```

5. **Convert Integer back to Padded Block:**

   Algorithm: Convert the decrypted SageMath Integer *m_int* back into a standard Python integer, and then convert that integer into a sequence of bytes (*block_bytes*) of length *block_size*, using big-endian byte order. Padding with leading null bytes occurs automatically if needed to reach *block_size*.

   Functions Used: *int(), .to_bytes()*.

   Code:

```
# Convert decrypted integer back to a full block (block_size
bytes).block_bytes = int(m_int).to_bytes(block_size, byteorder="big")
# Convert Integer to bytes
```

6. **De-pad Block (Extract Message Part):**

   Algorithm: Read the first byte of *block_bytes* to get the actual message length *L*. Extract the subsequent *L* bytes, which constitute the original message chunk for this block (see detailed in Section 4.2.3).

   Functions Used: *Indexing []*, *Slicing [:]*.

   Code:

```
# Extract header (first byte): actual message length L.
L = block_bytes[0] # Get message length from header byte
# Extract the actual message bytes: next L bytes.
message_part = block_bytes[1:1+L] # Extract message using slicing
```

7. **Concatenate Message Parts:**

   Algorithm: Append the extracted *message_part* bytes to a running sequence *decrypted_bytes* that accumulates the full original message.

   Code:

```
decrypted_bytes += message_part
```

8. **Decode Plaintext:**

   Algorithm: After processing all blocks, attempt to decode the complete *decrypted_bytes* sequence back into a string using UTF-8 encoding.

   Functions Used: *.decode()*.

   Code:

```
try:
    plaintext = decrypted_bytes.decode("utf-8") # Decode bytes to string
except UnicodeDecodeError:
    print("Error: Decrypted bytes do not form valid UTF-8.")
    sys.exit(1)
```

9. **Store Plaintext:**

   Algorithm: Open the designated output file and write the decoded plaintext string to it.

   Functions Used: *os.path.splitext()*, *open()*, *.write()*.

   Code:

```
# Prepare output filename.
base, ext = os.path.splitext(ciphertext_filename)
output_filename = f"{base}_decrypted{ext}" if ext else
f"{ciphertext_filename}_decrypted"
with open(output_filename, "w", encoding="utf-8") as f:
    f.write(plaintext) # Write decrypted string to file
```

Command Line Invocation :

```
sage rsa_decrypt.py message_cipher.txt private_key.csv
```

```
(base) suvajitsadhukhan@Suvajits-MacBook-Air rsa_final-year-project % sage rsa_decrypt.py message_cipher.txt private_key.csv
Decryption complete.
Decrypted file: message_cipher_decrypted.txt
Time taken to decrypt: 0.093950 seconds
```

4.2.3 Block Size and Padding

RSA operates on integers m such that $0 \leq m < n$. Since practical messages (files, text, etc.) are sequences of bytes and often much larger than what can be represented by an integer smaller than n, they must be processed in blocks. The size of these blocks and the method used to pad them are crucial for both functionality and security.

Determining Block Size: The maximum value for an integer m is $n - 1$. To ensure $m < n$, the message block, when converted to an integer, must be smaller than n. A common practice is to determine the maximum number of *bytes* that can fit into a block. Since n is approximately 2^{bits} (where '*bits*' is the bit length of n), the maximum number of bytes is typically $\lfloor (bits - 1)/8 \rfloor$. Our implementation (*rsa_encrypt.py*) calculates the block size in bytes as:

Code :

```
# Determine block size: ensure m < n.
block_size = (n.nbits() - 1) // 8
```

This ensures that the integer representation of the byte block will always be less than n.

Padding Scheme: Simply breaking the message into blocks and encrypting them (known as textbook RSA) is insecure. It's vulnerable to various attacks. Padding schemes add structure and randomness to each block before encryption. Our implementation (*rsa_encrypt.py*) employs a custom padding scheme for each block:

- The block size is fixed at block_size bytes, calculated as described above.
- The actual message data to be placed in a block (*chunk*) can be at most *data_size* = *block_size* - *15* bytes. This leaves *15* bytes for overhead (header and tail padding).
- Each *block_bytes* structure before integer conversion is constructed as follows:

Code:

```
L = len(chunk)  # actual message length for this block
# Header: 1 byte indicating L.
header = bytes([L])
# Pad the message chunk (if necessary) to exactly data_size bytes.
if L < data_size:
    pad_len = data_size - L
    pad_bytes = os.urandom(pad_len)
else:
    pad_bytes = b""
# Tail: 14 random bytes.
tail = os.urandom(14)
# Construct the full block.
block_bytes = header + chunk + pad_bytes + tail
```

This ensures every block sent to the *power_mod* function for encryption has the same fixed length (*block_size* bytes) and incorporates randomness.

De-padding: During decryption (*rsa_decrypt.py*), after computing $m = c^d mod n$ and converting m back to *block_bytes* of length *block_size*, the padding is removed using the header byte :

- After converting the decrypted integer back to block_bytes (length *block_size*), read the first byte (*block_bytes[0]*) to get the original length *L*.

- Extract the next *L* bytes (*block_bytes[1:1+L]*) as the original message part. The remaining padding bytes are implicitly discarded.

Code:

```
# Convert decrypted integer back to a full block (block_size bytes)
block_bytes = int(m_int).to_bytes(block_size, byteorder="big")
# Extract header (first byte): actual message length L.
L = block_bytes[0]
# Extract the actual message bytes: next L bytes.
message_part = block_bytes[1:1+L]
decrypted_bytes += message_part
```

The remaining bytes (intermediate padding and tail padding) are implicitly discarded as only *message_part* is appended to the final result.

This padding scheme, while custom, aims to provide basic security enhancements over textbook RSA by adding randomness and obscuring the original message length within the fixed block size. However, it's important to note that standardized padding schemes like OAEP (Optimal Asymmetric Encryption Padding) are generally recommended for robust security in real-world applications, as they provide proven protection against a wider range of attacks.

4.3 Relevant Scripts

4.3.1 Key Generation Script

`rsa_keygenerator.py` generates primes, calculates n, φ(n), e, and d, then saves the public and private keys.

Code:

```
#!/usr/bin/env sage -python
"""
rsa_keygenerator.py
-------------------
Generates RSA keys using primes of a specified bit length and stores keys in CSV
files.

Usage:
    sage rsa_keygenerator.py <bit_length>
Example:
    sage rsa_keygenerator.py 1024

This script:
    - Generates two prime numbers (p and q) of the given bit length.
    - Computes the RSA modulus n and Euler's totient phi.
    - Sets the public exponent e as a prime with bit length equal to (bits*2)-1
of the provided bit length.
    - Ensures gcd(e, phi) = 1 (if not, e is adjusted to the next prime).
    - Computes the private exponent d as the modular inverse of e modulo phi.
    - Stores the public key (e, n) in public_key.csv and the private key (d, n)
in private_key.csv.
"""

from sage.all import *
import sys, random, csv, time

def generate_prime(bits):
    """
    Generate a prime number with exactly 'bits' bits.
    It randomly selects a candidate in the range [2^(bits-1), 2^bits-1] and
returns the next prime.
    """
    while True:
        candidate = randint(2**(bits-1), 2**bits - 1)
        p = next_prime(candidate)
        if p.nbits() == bits:
            return p

def main():
    if len(sys.argv) != 2:
        print("Usage: sage rsa_keygenerator.py <bit_length>")
        sys.exit(1)
    try:
        bits = int(sys.argv[1])
    except ValueError:
        print("Error: bit_length must be an integer (e.g., 512, 1024).")
        sys.exit(1)

    start_time = time.time()  # Start timer for key generation

    # Generate two primes of the given bit length.
    p = generate_prime(bits)
    q = generate_prime(bits)
    # print(f"p = {p} ({len(str(p))} digits)")
    # print(f"q = {q} ({len(str(q))} digits)")
    n = p * q
    phi = (p - 1) * (q - 1)
```

```
    # Choose the public exponent e (2^16+1 = 65537 is common) and ensure it's
coprime with phi.
    e = 65537
    if gcd(e, phi) != 1:
        while gcd(e, phi) != 1:
            e = next_prime(e)

    # Compute private exponent d as the modular inverse of e modulo phi.
    d = inverse_mod(e, phi)

    # Write the public key (e, n) to public_key.csv.
    with open("public_key.csv", "w", newline="") as pub_file:
        writer = csv.writer(pub_file)
        writer.writerow(["e", "n"])
        writer.writerow([int(e), int(n)])

    # Write the private key (d, n) to private_key.csv.
    with open("private_key.csv", "w", newline="") as priv_file:
        writer = csv.writer(priv_file)
        writer.writerow(["d", "n"])
        writer.writerow([int(d), int(n)])

    end_time = time.time()  # End timer after keys are generated and saved
    elapsed = end_time - start_time

    print("Keys generated successfully.")
    print("Public key stored in public_key.csv")
    print("Private key stored in private_key.csv")
    print("Time taken to generate keys: {:.6f} seconds".format(elapsed))

if __name__ == "__main__":
    main()
```

4.3.2 Encryption Script

`rsa_encrypt.py` uses the public key to encrypt text messages, converting them to numeric form and storing cipher-text.

Code:

```
#!/usr/bin/env sage -python
"""
rsa_encrypt.py
--------------
Usage:
    sage rsa_encrypt.py <plaintext_filename> <public_key_csv>
Example:
    sage rsa_encrypt.py message.txt public_key.csv

Encrypts a plaintext file using RSA encryption and a public key stored in a CSV
file.
Now, each plaintext block is constructed as follows:
  - 1 byte: the actual length (L) of the message chunk.
  - (block_size - 15) bytes: message data (if shorter than this, padded with
random bytes).
  - 14 bytes: random tail padding.
Thus, each block has a fixed size = block_size, where:
    block_size = (n.nbits() - 1) // 8
Each block is then converted to an integer and encrypted.
The ciphertext file contains one ciphertext integer per line.
Intermediate steps are printed for demonstration.
"""

from sage.all import *
import sys, os, time, csv
```

```
def usage():
    print("Usage: sage rsa_encrypt.py <plaintext_filename> <public_key_csv>")
    sys.exit(1)

def main():
    if len(sys.argv) != 3:
        usage()

    plaintext_filename = sys.argv[1]
    public_key_csv = sys.argv[2]

    # Read public key (e, n) from CSV.
    try:
        with open(public_key_csv, "r", encoding="utf-8") as f:
            reader = csv.reader(f)
            next(reader)  # skip header
            row = next(reader)
            if len(row) < 2:
                print("Error: Invalid public key file format.")
                sys.exit(1)
            e = Integer(row[0])
            n = Integer(row[1])
    except Exception as ex:
        print("Error reading public key CSV file:", ex)
        sys.exit(1)

    # Read plaintext.
    if not os.path.exists(plaintext_filename):
        print("Error: File", plaintext_filename, "does not exist.")
        sys.exit(1)
    with open(plaintext_filename, "r", encoding="utf-8") as f:
        plaintext = f.read()
    plaintext_bytes = plaintext.encode("utf-8")

    # Determine block size: ensure m < n.
    block_size = (n.nbits() - 1) // 8
    # print("Block size:", block_size)
    if block_size < 15:
        print("Error: Block size too small for padding requirements.")
        sys.exit(1)
    data_size = block_size - 15  # available bytes for actual message in each
block

    # Split plaintext into chunks of size <= data_size.
    chunks = [plaintext_bytes[i:i+data_size] for i in range(0,
len(plaintext_bytes), data_size)]

    encrypted_blocks = []
    enc_start_time = time.time()

    # print("=== Encrypting Blocks ===")
    for i, chunk in enumerate(chunks):
        L = len(chunk)  # actual message length for this block
        # Header: 1 byte indicating L.
        header = bytes([L])
        # Pad the message chunk (if necessary) to exactly data_size bytes.
        if L < data_size:
            pad_len = data_size - L
            pad_bytes = os.urandom(pad_len)
        else:
            pad_bytes = b""
        # Tail: 14 random bytes.
        tail = os.urandom(14)
        # Construct the full block.
        block_bytes = header + chunk + pad_bytes + tail
        if len(block_bytes) != block_size:
            print("Error: Block length mismatch. Expected:", block_size, "Got:",
len(block_bytes))
            sys.exit(1)
```

```
        # Convert block bytes to integer.
        m_int = Integer(int.from_bytes(block_bytes, byteorder="big"))
        c_int = power_mod(m_int, e, n)
        encrypted_blocks.append(c_int)
    enc_end_time = time.time()

    # Prepare output filename.
    base, ext = os.path.splitext(plaintext_filename)
    output_filename = f"{base}_cipher{ext}" if ext else
f"{plaintext_filename}_cipher"
    with open(output_filename, "w", encoding="utf-8") as f:
        for c in encrypted_blocks:
            # Write each ciphertext (as an integer) on its own line.
            f.write(f"{int(c)}\n")

    print("\nEncryption complete.")
    print("Encrypted file:", output_filename)
    print("Time taken to encrypt: {:.6f} seconds".format(enc_end_time -
enc_start_time))

if __name__ == "__main__":
    main()
```

4.3.3 Decryption Script

`rsa_decrypt.py` performs decryption by using the private key to revert ciphertext back to the original plaintext.

Code:

```
#!/usr/bin/env sage -python
"""
rsa_decrypt.py
--------------
Usage:
    sage rsa_decrypt.py <ciphertext_filename> <private_key_csv>
Example:
    sage rsa_decrypt.py message_cipher.txt private_key.csv

Decrypts an RSA-encrypted ciphertext file using a private key stored in a CSV
file.
Each ciphertext corresponds to a fixed-size block of length block_size, where:
    block_size = (n.nbits() - 1) // 8.
After decryption, the block is interpreted as follows:
  - The first byte (header) gives L, the actual number of message bytes in this
block.
  - The next L bytes are the true message.
  - The remaining bytes are random padding and are discarded.
The recovered message blocks are concatenated and then written to an output file.
"""

from sage.all import *
import sys, os, time, csv

def usage():
    print("Usage: sage rsa_decrypt.py <ciphertext_filename> <private_key_csv>")
    sys.exit(1)

def main():
    if len(sys.argv) != 3:
        usage()

    ciphertext_filename = sys.argv[1]
    private_key_csv = sys.argv[2]
```

```
    # Read private key (d, n) from CSV.
    try:
        with open(private_key_csv, "r", encoding="utf-8") as f:
            reader = csv.reader(f)
            next(reader)  # skip header
            row = next(reader)
            if len(row) < 2:
                print("Error: Invalid private key file format.")
                sys.exit(1)
            d = Integer(row[0])
            n = Integer(row[1])
    except Exception as ex:
        print("Error reading private key CSV file:", ex)
        sys.exit(1)

    if not os.path.exists(ciphertext_filename):
        print("Error: File", ciphertext_filename, "does not exist.")
        sys.exit(1)

    # Determine block size.
    block_size = (n.nbits() - 1) // 8
    if block_size < 15:
        print("Error: Block size too small.")
        sys.exit(1)
    data_size = block_size - 15  # number of bytes allocated for actual message

    # Read ciphertext file.
    encrypted_blocks = []
    with open(ciphertext_filename, "r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if not line:
                continue
            try:
                c_int = Integer(line)
                encrypted_blocks.append(c_int)
            except Exception as ex:
                print("Error parsing line:", line)
                sys.exit(1)

    decrypted_bytes = b""
    dec_start_time = time.time()
    for c_int in encrypted_blocks:
        m_int = power_mod(c_int, d, n)
        # Convert decrypted integer back to a full block (block_size bytes).
        block_bytes = int(m_int).to_bytes(block_size, byteorder="big")
        # Extract header (first byte): actual message length L.
        L = block_bytes[0]
        # Extract the actual message bytes: next L bytes.
        message_part = block_bytes[1:1+L]
        decrypted_bytes += message_part
    dec_end_time = time.time()

    try:
        plaintext = decrypted_bytes.decode("utf-8")
    except UnicodeDecodeError:
        print("Error: Decrypted bytes do not form valid UTF-8.")
        sys.exit(1)

    # Prepare output filename.
    base, ext = os.path.splitext(ciphertext_filename)
    output_filename = f"{base}_decrypted{ext}" if ext else
f"{ciphertext_filename}_decrypted"
    with open(output_filename, "w", encoding="utf-8") as f:
        f.write(plaintext)

    print("Decryption complete.")
    print("Decrypted file:", output_filename)
    print("Time taken to decrypt: {:.6f} seconds".format(dec_end_time -
dec_start_time))

if __name__ == "__main__":
    main()
```

5. Testcases and Results

This chapter details the testing procedures used to verify the functional correctness of the implemented RSA system and presents the results of performance benchmarks conducted on the key generation, encryption, and decryption processes using SageMath.

5.1 Testing RSA Functionality

To ensure the reliability and correctness of the RSA implementation (*rsa_keygenerator.py, rsa_encrypt.py, rsa_decrypt.py*), a dedicated test script (*test_rsa.py*) was developed and executed. This script automates the process of key generation, encrypting various messages, decrypting the resulting cipher-texts, and comparing the final decrypted message with the original plaintext.

The testing procedure was divided into two main groups:

Group 1: Standard Key Size (1024 bits)

This group focused on testing the core functionality under typical conditions.

1. **Key Generation:** A 1024-bit RSA key pair was generated using *rsa_keygenerator.py* 1024.

2. **Block Size Calculation:** The corresponding *block_size* and *data_area* (maximum message bytes per block before padding) were calculated based on the generated key's modulus size ($n \approx 2 \times 1024$ bits).

3. **Encryption/Decryption Tests:** Several test cases with varying message lengths were executed using the generated 1024-bit keys:
   - **Exact Size:** A message with length exactly equal to the calculated *data_area*.
   - **One Byte Smaller:** A message one byte shorter than *data_area*.
   - **One Byte Larger:** A message one byte longer than data_area.
   - **Very Small Message:** A short message (e.g., "HelloRSA!X").
   - **Very Large Message:** A 1 MB message composed of random characters.

4. **Verification:** For each test case, the *test_rsa.py* script automatically compared the original plaintext file with the final decrypted file.

Results: All tests in Group 1 completed successfully, with the decrypted message perfectly matching the original plaintext in every case. This confirms the functional correctness of the key generation, encryption (including padding and blocking), and decryption processes for a standard key size.

Group 2: Small Key Size (64 bits)

This group tested the system's behaviour under edge-case conditions where the key size is too small for the implemented padding scheme.

1. **Key Generation:** A 64-bit RSA key pair was generated using rsa_keygenerator.py 64.

2. **Block Size Calculation:** The block_size and data_area were calculated. For a 64-bit key (n≈128 bits), the block_size is 15 bytes, resulting in a data_area of 0.

3. **Encryption Attempt:** The rsa_encrypt.py script was invoked with a dummy message.

4. **Verification:** The test script expected the encryption process to fail due to insufficient block_size for the padding overhead.

Results: The test in Group 2 behaved as expected. The rsa_encrypt.py script exited with an error, confirming that the implementation correctly handles situations where the key size is incompatible with the padding requirements.

Overall, the testing phase demonstrated that the implemented RSA scripts function correctly for valid inputs and handle predictable error conditions appropriately.

5.2 Benchmarking RSA Performance

To evaluate the computational cost of the implemented RSA operations using SageMath, a benchmark script (*benchmark_rsa.py*) was executed. This script measured the time taken for key generation, encryption, and decryption under varying parameters. The results were logged and are summarized below. All encryption and decryption benchmarks were performed using a fixed 1024-bit RSA key pair generated specifically for these tasks.

Task 1: Key Generation Benchmarking

This task measured the time required to generate RSA key pairs of different sizes.

Key Size (bits)	Time Taken (seconds)
512	0.412522
768	2.154601
1024	1.609887
1280	3.306087
1536	7.288917
1792	10.806548
2048	19.840382

(A separate 1024-bit key pair generated for Tasks 2 & 3 took 2.065729 *seconds).*

Analysis: As expected, the time required for key generation generally increases with the key size. This is primarily due to the increased computational effort needed by SageMath's *next_prime()* function (likely relying on probabilistic primality tests followed by deterministic checks) to find large prime numbers (p and q) of the specified bit length. The slight dip at 1024 bits compared to 768 bits could be due to random variations in the prime search or system load during the benchmark. However, the overall trend clearly shows a positive correlation between key size and generation time.

Task 2: Encryption Benchmarking (1024-bit Key)

This task measured the time required to encrypt messages of different sizes using the generated 1024-bit public key ($e = 65537$).

Message Size	Encryption Time (seconds)
1KB	0.000206
10KB	0.001330
100KB	0.013017
1MB	0.124622
10MB	1.240533

Analysis: Encryption time shows a relatively small increase as the message size grows from 1KB up to 1MB. This is because RSA encryption, using the public exponent $e = 65537$, involves modular exponentiation (*power_mod(m, e, n)*) that is relatively fast due to the small number of set bits in e. The time is dominated by the number of blocks processed rather than the complexity of the operation per block. The jump in time for the 10MB message suggests that overhead associated with handling a very large number of blocks (reading the file, splitting into chunks, Python loop overhead) becomes more significant relative to the cryptographic computation itself.

Task 3: Decryption Benchmarking (1024-bit Key)

This task measured the time required to decrypt cipher-texts corresponding to different original message sizes, using the generated 1024-bit private key.

Message Size	Decryption Time (seconds)
1KB	0.023835
10KB	0.203589
100KB	2.018619
1MB	20.618169
10MB	229.986114

Analysis: Decryption time increases significantly and non-linearly with the message size. This is a characteristic feature of RSA. Decryption uses the private exponent d, which is generally a large number (comparable in size to n). Modular exponentiation with a large exponent ($m = c^d \ (mod\ n)$, performed by *power_mod(c, d, n)*, is computationally much more intensive than encryption with the small public exponent e. The results clearly demonstrate that decryption is the performance bottleneck in RSA, especially for large amounts of data. The time taken grows substantially as the number of blocks to decrypt increases.

6. Security Analysis of RSA: Types of Attacks and Defence Mechanisms

While RSA is a cornerstone of modern cryptography, its security is not absolute and depends critically on proper implementation, key management, and the underlying mathematical problem's difficulty. This chapter discusses the foundation of RSA's security and common attack vectors, evaluating the defences present in this project's implementation and suggesting improvements.

6.1 Security of RSA

The security of the RSA algorithm hinges primarily on two factors: the difficulty of factoring large integers and the use of sufficiently large keys.

6.1.1 Difficulty of Factoring Large Integers

The fundamental security assumption of RSA is that factoring the public modulus n into its two large prime factors, p and q, is computationally infeasible for sufficiently large values of n. If an attacker could efficiently factor n, they could then compute $\phi(n) = (p-1)(q-1)$ and subsequently derive the private key d from the public key (e, n) by calculating the modular inverse $d \equiv e^{-1} \ (mod\ \phi(n))$.

Integer factorization is a well-studied problem in number theory. The best-known classical algorithm for factoring large integers suitable for RSA is the General Number Field Sieve (GNFS). The computational effort required by GNFS grows sub-exponentially but very rapidly with the size of the number n. Factoring numbers of the size currently recommended for RSA (e.g., 2048 bits or more) is considered far beyond the reach of current classical computing technology.

6.1.2 Importance of Key Size

The difficulty of factoring n is directly related to its size (bit length). Therefore, the choice of key size is paramount to RSA security. A larger key size results in a larger modulus n, exponentially increasing the computational resources needed to factor it using algorithms like GNFS.

- ***Recommendations:*** Cryptographic standards bodies like NIST provide recommendations for minimum key sizes. For example, 2048-bit RSA keys are generally considered the minimum for security through the 2030s, with 3072-bit or 4096-bit keys recommended for longer-term security.

- ***Trade-offs:*** As demonstrated by the benchmarking results in Section 5.2, increasing the key size significantly increases the time required for key generation and decryption. This represents a trade-off between security and performance.

- ***Quantum Computing Threat:*** The advent of large-scale, fault-tolerant quantum computers poses a significant threat to RSA, as Shor's algorithm can factor large integers efficiently. This motivates research into post-quantum cryptography (PQC).

6.2 Common Attacks on RSA

Beyond direct factorization, various attacks target specific weaknesses in RSA implementations or usage protocols. Proper implementation practices, including padding and careful parameter selection, are crucial defences.

6.2.1 *Factoring Attacks*

- ***Description:*** Directly attempting to factor the modulus n using algorithms like GNFS.
- ***Defence in Project:*** The defence relies entirely on the user choosing a sufficiently large *bit_length* when running rsa_keygenerator.py. The script itself does not enforce a minimum size beyond what's implicitly required for the padding scheme to function (tested in Section 5.1).

6.2.2 Timing Attacks:

- ***Description:*** A type of side-channel attack measuring the precise time taken for decryption operations. Variations in time can leak information about the private exponent d.
- ***Defence in Project:*** This project relies on SageMath's *power_mod* function. It's likely that the underlying libraries used by SageMath (like GMP) implement countermeasures such as exponent blinding or constant-time modular exponentiation techniques. However, this is an implicit defence based on the library's implementation, not an explicit measure in the project's Python scripts.

6.2.3 Chosen Ciphertext Attacks (CCA):

- **Description:** The attacker obtains the decryption of chosen ciphertexts to deduce information about the private key or decrypt a target ciphertext. Textbook RSA (without padding) is completely vulnerable.
- **Defence in Project:** The custom padding scheme implemented in *rsa_encrypt.py* (1-byte length header + message + random intermediate padding + 14-byte random tail) provides *some* defence compared to textbook RSA by adding randomness. However, this scheme is non-standard and **does not provide proven security against CCA**. An attacker could potentially craft ciphertexts that, upon decryption, reveal information based on whether the padding structure is valid or how error handling is performed.
- **Future Improvements: This is a critical area for improvement.** Replace the custom padding with the **OAEP (Optimal Asymmetric Encryption Padding)** standard. This is the most important "Future Work" item (item 1) for enhancing security. Libraries like Python's cryptography provide OAEP implementations.

6.2.4 Low Public Exponent Attacks:

- ***Description:*** If a small public exponent e (like $e = 3$) is used *without* proper padding, attacks can recover the plaintext m if $m^e < n$.

- ***Defence in Project:*** The rsa_keygenerator.py script defaults to $e = 65537$, which is not considered cryptographically "small" in this context. Furthermore, the custom padding scheme ensures that the integer m fed into the exponentiation is always large (close to n), effectively mitigating this specific attack even if e were small. The primary defence against broader issues related to small e is the use of secure padding.

- ***Future Improvements:*** Continue using $e = 65537$. Implementing OAEP (Future Work item 1) provides robust protection regardless of the standard e value chosen.

6.2.5 Common Modulus Attack:

- ***Description:*** Occurs if multiple users share the same n but have different e, d pairs.

- ***Defence in Project:*** This attack is related to key management, not the core algorithm implementation. The scripts generate a unique key pair (including n) each time *rsa_keygenerator.py* is run. The defence relies on users generating unique keys for each distinct entity.

6.2.6 Implementation Errors / Side-Channel Attacks:

- ***Description:*** Flaws in software/hardware leaking information (power analysis, fault injection, cache timing).

- ***Defence in Project:*** Relies heavily on the security of the underlying Python interpreter, SageMath libraries, and the operating system. No specific defences are implemented at the script level beyond using standard library functions.

- ***Future Improvements:*** For high-security needs, consider code hardening techniques, formal verification (difficult), or running critical operations within HSMs. This is generally beyond the scope of a typical project unless specifically focused on low-level implementation security.

6.2.7 Poor Random Number Generation:

- ***Description:*** Predictable random numbers compromise prime generation (p,q) and padding randomness.

- ***Defence in Project:*** The custom padding in rsa_encrypt.py uses *os.urandom()*. This function is generally considered a cryptographically secure pseudo-random number generator (CSPRNG) as it draws entropy from the operating system. SageMath's *randint* used for prime candidate generation likely also relies on a secure RNG.

- **Future Improvements:** Ensure the underlying OS provides a well-seeded and robust source of entropy for *os.urandom()*. For extremely critical systems, dedicated hardware RNGs might be considered.

In summary, while the mathematical foundation of RSA is strong given large enough keys, its practical security relies heavily on using standardized, secure padding schemes (like OAEP), robust random number generation, appropriate key sizes, and careful implementation to avoid side-channel vulnerabilities. The most significant security enhancement for this project would be implementing OAEP padding.

7. Conclusion and Future Work

Conclusion

This project successfully implemented the RSA public-key cryptosystem, encompassing key generation, encryption, and decryption functionalities. Leveraging the capabilities of the Python programming language and the SageMath computer algebra system, particularly its support for arbitrary-precision integers and efficient modular arithmetic functions (*power_mod, inverse_mod, next_prime, gcd*), a working RSA model was developed.

The functional correctness of the implementation was verified through a series of test cases (*test_rsa.py*) covering various message sizes and edge conditions, confirming that messages could be successfully encrypted and subsequently decrypted back to their original form. Performance benchmarks (*benchmark_rsa.py*) provided insights into the computational costs associated with RSA operations. Key generation time was shown to increase with key size, reflecting the difficulty of finding large primes. Encryption was observed to be relatively fast due to the use of the standard small public exponent $e = 65537$, while decryption proved to be significantly more computationally intensive, especially for large messages, highlighting the asymmetry inherent in RSA operations.

The security analysis reiterated that RSA's strength is founded on the computational difficulty of factoring the large modulus n. The critical importance of using sufficiently large key sizes (e.g., 2048 bits or more) was emphasized as the primary defence against factorization attacks. Furthermore, the analysis explored common attack vectors beyond factorization. It highlighted that while the current implementation includes defences like using a standard public exponent and a secure RNG (*os.urandom*), its reliance on a custom padding scheme is a significant limitation compared to standardized approaches like OAEP, particularly concerning chosen ciphertext attacks.

In conclusion, this project provides a practical demonstration of RSA implementation using SageMath, illustrating its core principles, performance characteristics, and the fundamental security considerations required for its effective use, while also identifying key areas for security enhancement.

Future Work

Several avenues exist for extending and improving upon this project:

1. ***Implement Standardized Padding (OAEP): (High Priority for Security)*** Replace the custom padding scheme with the industry-standard OAEP (Optimal Asymmetric Encryption Padding). This would significantly enhance the security of the implementation against chosen ciphertext attacks and align it with best practices.

2. ***Optimize Decryption (CRT):*** Explore and implement decryption optimization using the Chinese Remainder Theorem (CRT). This requires modifying the key generation to store p and q (securely) alongside d, and changing the decryption logic. This would address the performance bottleneck identified in the benchmarks.

3. ***Develop a User Interface/Application:*** Integrate the RSA scripts into a simple graphical user interface (GUI) or command-line application for easier file encryption/decryption or secure messaging simulation.

4. ***Enhance Error Handling and Input Validation:*** Improve the robustness of the scripts by adding more comprehensive error handling for file I/O, key format validation, input argument validation (e.g., minimum key size), and potential cryptographic edge cases.

5. ***Comparative Benchmarking:*** Benchmark the SageMath implementation against other cryptographic libraries (e.g., Python's cryptography library) performing RSA operations (especially with OAEP and potentially CRT) to compare performance.

6. ***Security Auditing and Attack Simulation:*** Conduct a more formal security review of the code or attempt to simulate specific attacks (e.g., a simplified timing attack if feasible in the environment) to better understand vulnerabilities.

7. ***Explore Post-Quantum Cryptography:*** Investigate lattice-based or other post-quantum cryptographic algorithms being standardized by NIST as potential long-term replacements for RSA in the face of the quantum computing threat.

References

1. **Rivest, R. L., Shamir, A., & Adleman, L.** (1978). *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. Communications of the ACM. https://people.csail.mit.edu/rivest/Rsapaper.pdf

2. **Paar, C., & Pelzl, J.** (2010). *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer. https://uim.fei.stuba.sk/wp-content/uploads/2018/02/Understanding_Cryptography_Chptr_7-The_RSA_Cryptosystem.pdf

3. **Boneh, D.** (1999). *Twenty Years of Attacks on the RSA Cryptosystem*. Notices of the AMS. https://www.ams.org/notices/199902/boneh.pdf

4. **Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A.** (1996). *Handbook of Applied Cryptography*. CRC Press. https://dl.icdst.org/pdfs/files3/f7ba35bf7149b541644785c9270cc6b8.pdf

5. **Shah, S. A. A., Gondal, M. A., & Hussain, M.** (2021). *Systematic and Critical Review of RSA Based Public Key Cryptographic Schemes: Past and Present Status*. ResearchGate. https://www.researchgate.net/publication/356372929

6. **Yousif, A. A. A., & Maarof, M. A.** (2019). *Methods toward Enhancing RSA Algorithm: A Survey*. SSRN Electronic Journal. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3412776

7. **Bernstein, D. J., Heninger, N., Lou, P., & Valenta, L.** (2017). *Post-Quantum RSA*. IACR Cryptology ePrint Archive. https://eprint.iacr.org/2017/351.pdf

8. **Tuteja, A., & Shrivastava, A.** (2014). *A Literature Review of Some Modern RSA Variants*. International Journal for Scientific Research & Development (IJSRD). https://ijsrd.com/articles/IJSRDV2I8134.pdf

9. **Sowjanya, S., & Rao, K. S.** (2013). *A Study and Performance Analysis of RSA Algorithm*. International Journal of Computer Science and Mobile Computing (IJCSMC). https://ijcsmc.com/docs/papers/June2013/V2I6201330.pdf

10. **SageMath**. (n.d.). *Open-source mathematics software system for algebra and cryptography*. https://www.sagemath.org

11. **Python Software Foundation**. (n.d.). *Python Language Reference, Version 3.x*. https://www.python.org/

Appendix I – File Descriptions

File Name	Description
rsa_keygenerator.py	Script to generate RSA public and private key pairs using SageMath.
rsa_encrypt.py	Script to encrypt a plaintext message using the RSA public key.
rsa_decrypt.py	Script to decrypt the ciphertext using the RSA private key.
public_key.csv	Stores the public key pair (e, n) in CSV format.
private_key.csv	Stores the private key pair (d, n) in CSV format.
message.txt	Input plaintext file for encryption.
message _cipher.txt	Output ciphertext file generated after encryption.
message _cipher _decrypted.txt	Output file containing the final decrypted plaintext.
benchmark_rsa.py	Script used to test key generation speed and encryption/decryption times.
benchmark_output.txt	Text file containing test results and benchmarking data.

Appendix II – Environment and Tools

SageMath Version: 9.3
Python Version: 3.9+
Editor: Visual Studio Code / Jupyter Notebook
Platform: macOS 15.4 / Ubuntu 22.04
Libraries Used:
- sage.all for cryptographic number theory functions
- csv, os, sys, time for file and process handling