Buffalo State University

Computer Information System

**From Chaos to Chiper: The Beauty of RSA**

CIS 494: Undergraduate Research in Computing

Spring 2025

Jade Taylor

Professor Sarbani Banerjee

Faculty Advisor Dr Neal Mazur

May 8th, 2025

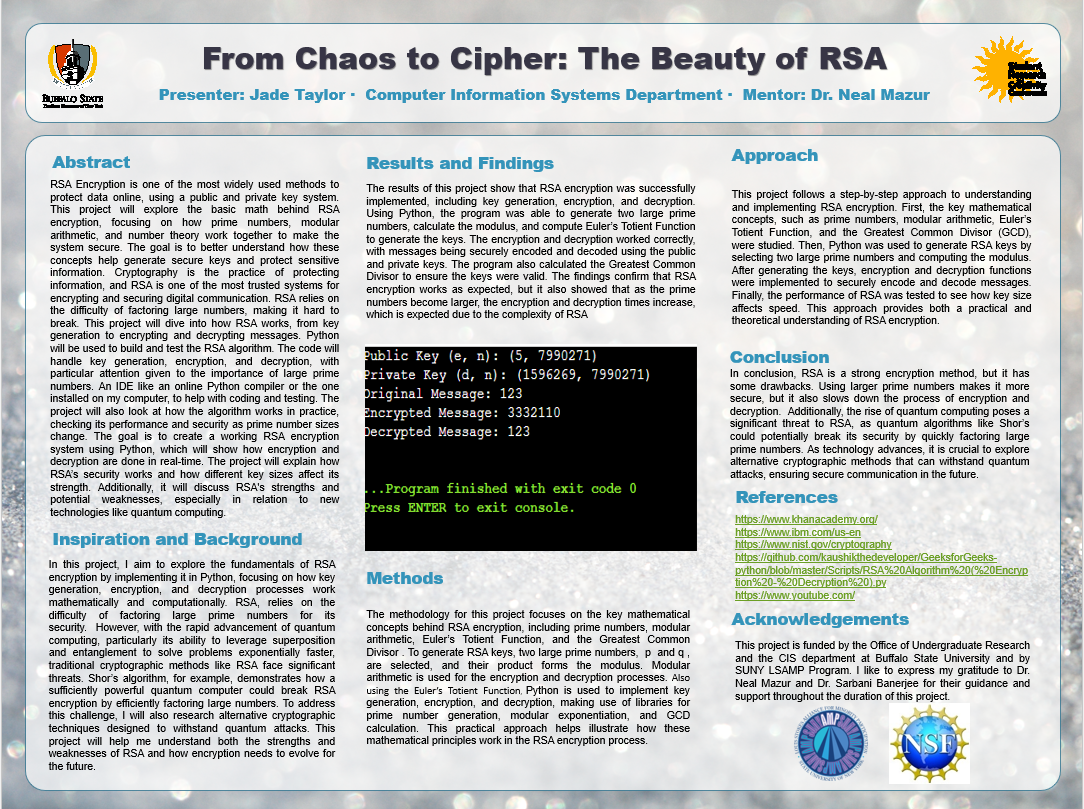
Abstract

RSA Encryption is one of the most widely used methods to protect data online, using a public and private key system. This project will explore the basic math behind RSA encryption, focusing on how prime numbers, modular arithmetic, and number theory work together to make the system secure. The goal is to better understand how these concepts help generate secure keys and protect sensitive information.

Cryptography is the practice of protecting information, and RSA is one of the most trusted systems for encrypting and securing digital communication. RSA relies on the difficulty of factoring large numbers, making it hard to break. This project will dive into how RSA works, from key generation to encrypting and decrypting messages.

Python will be used to build and test the RSA algorithm. The code will handle key generation, encryption, and decryption, with particular attention given to the importance of large prime numbers. An IDE like an online Python compiler or the one installed on my computer, to help with coding and testing. The project will also look at how the algorithm works in practice, checking its performance and security as prime number sizes change.

The goal is to create a working RSA encryption system using Python, which will show how encryption and decryption are done in real-time. The project will explain how RSA’s security works and how different key sizes affect its strength. Additionally, it will discuss RSA's strengths and potential weaknesses, especially in relation to new technologies like quantum computing.



My research project focused on learning and understanding the mathematics behind RSA encryption. I also explored the potential threats to this widely used public-key encryption system. Since I began with little to no prior knowledge of the topic, I taught myself the fundamentals and studied a sample RSA code I found online. This hands-on experience helped me connect theoretical mathematical concepts to their real-world implementation in code.

RSA ensures data security with two keys: a public key and a private key. It is grounded in number theory, especially the use of prime numbers and modular arithmetic. RSA is commonly used to secure sensitive information such as emails, digital signatures, and online banking transactions.

The main objective of my research was to gain insight into how secure keys are generated and how messages are encrypted and decrypted. RSA's security is based on the difficulty of factoring large numbers. Prime numbers are defined to be a number greater than 1 that can only be divided by 1 and themselves, which is crucial to this process. While multiplying two large primes is computationally easy, factoring the resulting product back into its original primes is extremely difficult. Modular arithmetic, often described as clock math, is essential for securely scrambling and unscrambling messages. Together, prime numbers, modular arithmetic, and number theory form the foundational building blocks of modern encryption.

One of RSA’s primary strengths is that it is widely trusted and used. However, it is not without weaknesses. A significant threat to RSA is Shor’s Algorithm, a quantum computing algorithm developed by Peter Shor. This algorithm can factor large integers exponentially faster than classical algorithms, posing a major risk to RSA's long-term security. As of now, RSA remains secure because conventional computers cannot factor large numbers within a reasonable time frame.

The Python program included in my research presents a simplified version of the RSA encryption algorithm. It demonstrates how number theory and modular arithmetic are applied to digital security. Key generation begins by selecting two prime numbers and computing them using a specific formula, followed by Euler’s totient function. Encryption and decryption follow the mathematical formulas C = M^e and M = C^d, where M is the plaintext message, C is the ciphertext, e is the public exponent, and d is the private exponent.

The code provides a hands-on example that illustrates how RSA uses mathematical principles to generate secure public and private key pairs. The comments in the code enhance understanding by clarifying how each step works, especially the concepts of modular arithmetic and key generation. Although this version of RSA is simplified and not secure for real-world use, it served its educational purpose well.

The implementation of the RSA algorithm in Python reinforces how core mathematical concepts can be used to secure digital communication. By applying number theory, prime factorization, modular arithmetic, and key generation, the program successfully demonstrates the encryption and decryption process. This research highlights the importance of understanding the mathematical foundation behind encryption, which is essential for developing and analyzing secure systems in today's evolving technological world.

Overall, I am grateful to have chosen this research topic. Not only did this project deepen my knowledge of RSA encryption, but it also allowed me to share what I learned with others while continuously expanding my understanding throughout the entire process.

# Jade Taylor

# Python Program for implementation of RSA Algorithm

def power(base, expo, m):

res = 1

base = base % m

while expo > 0:

if expo & 1:

res = (res \* base) % m

base = (base \* base) % m

expo = expo // 2

return res

# Function to find modular inverse of e modulo phi(n)

# Here we are calculating phi(n) using Hit and Trial Method

# but we can optimize it using Extended Euclidean Algorithm

def modInverse(e, phi):

for d in range(2, phi):

if (e \* d) % phi == 1:

return d

return -1

# RSA Key Generation

def generateKeys():

p = 7919

q = 1009

n = p \* q

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

# Choose e, where 1 < e < phi(n) and gcd(e, phi(n)) == 1

e = 0

for e in range(2, phi):

if gcd(e, phi) == 1:

break

# Compute d such that e \* d ≡ 1 (mod phi(n))

d = modInverse(e, phi)

return e, d, n

# Function to calculate gcd

def gcd(a, b):

while b != 0:

a, b = b, a % b

return a

# Encrypt message using public key (e, n)

def encrypt(m, e, n):

return power(m, e, n)

# Decrypt message using private key (d, n)

def decrypt(c, d, n):

return power(c, d, n)

# Main execution

if \_\_name\_\_ == "\_\_main\_\_":

# Key Generation

e, d, n = generateKeys()

print(f"Public Key (e, n): ({e}, {n})")

print(f"Private Key (d, n): ({d}, {n})")

# Message

M = 123

print(f"Original Message: {M}")

# Encrypt the message

C = encrypt(M, e, n)

print(f"Encrypted Message: {C}")

# Decrypt the message

decrypted = decrypt(C, d, n)

print(f"Decrypted Message: {decrypted}")