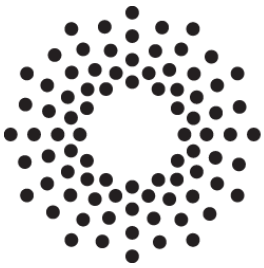

LONDON METROPOLITAN UNIVERSITY

CS7064 Information Security

Spring 2024-2025

KeyCrypt: Practical RSA cryptosystem for secure communication

Tutor

Graham Taylor-Russel

Student Name	Student ID
Jorge Vinuela Parez	23038621
Manahil Khan	24039975
Ayodeji Durojaiye	23000209

Table of Contents

Table of Figures

1. Introduction

Securing information has become a top priority in today's digital world. One key method is cryptography, the practice of protecting data using encryption and decryption techniques to prevent unauthorized access.

It is broadly classified into symmetric and asymmetric techniques. Symmetric encryption uses a single key for both encryption and decryption, making it fast and efficient. In contrast, asymmetric encryption uses a public key to encrypt and a private key to decrypt, eliminating the need to share the private key. The RSA algorithm is a cornerstone of modern cryptography, developed in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman, and is the selected method for this project.

The security of RSA relies on the mathematical difficulty of factoring large prime numbers. It is widely used in securing sensitive data, enabling digital signatures, and ensuring encrypted communication across the internet. It plays a critical role in protocols like SSL/TLS, which protect web traffic, and is fundamental to digital trust in activities such as online banking, email security, and software verification. Despite the development of more efficient algorithms, RSA remains relevant due to its proven reliability and widespread implementation.

1.1 Summary of Encryption Methods

Two options were proposed: El Gamal and RSA. Both are asymmetric encryption schemes, meaning they use a public key for encryption and a private key for decryption.

El Gamal relies on the hardness of the discrete logarithm problem for its security.

1. **Key Generation**: Choose a large prime *p*, a generator *g*, and a private key *x*. The public key is:

$$h = g^x \ mod \ p$$

   - **Public key:** (p, g, h)
   - **Private Key:** x

2. **Encryption**: To encrypt a message *m*, choose a random number *y*, then compute:

$$c_1 = g^y \ mod \ p$$
$$c_2 = m \cdot h^y \ mod \ p$$

   - **Ciphertext** = (c_1, c_2)

3. **Decryption**: Using the private key *x*, calculate $s = c_1^x \ mod$ and find its inverse. Multiply c_2 by the inverse to get the original message.

On the other hand, RSA is based on the mathematical properties of modular arithmetic and prime numbers. Its security relies on the fact that factoring the product of two large prime numbers is computationally difficult.

1. **Key Generation**:
    a. Choose two large prime numbers, p and q.
    b. Compute $n = p \cdot q$
    c. Compute $\Phi(n) = (p-1) \cdot (q-1)$
    d. Choose a public exponent e (commonly 65537).
    e. Calculate the private key d such that $d = e^{-1} \, mod \, \Phi(n)$

- **Public key:** (e, n)
- **Private Key:** (d, n)

2. **Encryption**:
    a. Convert the message into a number m.
    b. Compute the ciphertext $c = m^e \, mod \, n$.
3. **Decryption**:
    a. To decrypt, use $m = c^d \, mod \, n$

1.2 Why did we choose RSA?

RSA was chosen for this project due to its reputation as a trusted public-key encryption system, relying on the difficulty of factoring large primes for security. Unlike symmetric algorithms, RSA uses a public-private key pair, allowing secure communication without pre-shared secrets, making it ideal for a user-friendly GUI application. With a fixed public exponent, RSA ensures robust security and high performance, suitable for real-time communication. Additionally, RSA is well-suited for educational and modular implementations. Widely used in protocols like HTTPS and SSL/TLS, RSA's presence in standard libraries highlights its reliability and makes it ideal for demonstrating both practical and theoretical cryptography.

The software uses RSA encryption and accepts plaintext input of any length via a graphical user interface (GUI). The system encrypts messages using the recipient's public key and outputs the result as an encrypted message, which can then be used for secure communication or storage. Private keys are saved, enabling quick decryption by selecting the associated key address.

2. Advantages and Disadvantage of the Program

2.1 Advantages of the program

1. **User Interface (Usability)**
    A. The program uses a simple GUI, making it accessible and easy to understand even for users unfamiliar with command-line tools.

B. Utility is enhanced through buttons for encryption, decryption, and clearing input/output fields, making the interface more user-friendly and intuitive.

2. **RSA Encryption and Decryption**
   - **A.** It implements the RSA cryptographic algorithm, one of the most widely used and secure public-key encryption methods.
   - **B.** Users encrypt plaintext with the public key and decrypt ciphertext with the private key.

3. **Management of Keys**
   - **A.** It allows for on-demand regeneration of RSA key pair.
   - **B.** A backup of the private keys is stored, which makes it easy for users to switch between them.

4. **Error Management**
   - **A.** The application also has exception error handling which:
     - I. Alerts the user if they attempt to work with an empty field.
     - II. Alerts when ciphertext does not conform to proper JSON standards.
     - III. This can throw any Number of errors about invalid Private Keys or faulty Decryptions.
   - **B.** This increases reliability and reduces crashes.

5. **Modular Architecture**
   - **A.** The program is designed around creating standard format modules to support RSA functionality (RSA_Functions and RSA_Core files) and as such reinforces separation of tuple concerns, improving maintainability.
   - **B.** Encryption, Decryption, and Regenerate Keys are separate functions, improving clarity and reusability.

6. **Academic Importance**
   - **A.** The software serves as an educational tool, demonstrating how RSA handles encryption, decryption, and key generation through a practical and efficient implementation.
   - **B.** This underscore practical cryptographic procedures with a streamlined implementation.

2.2 Disadvantages of the Program

1. **Security Issues**
   - A. Memory-resident Plaintext and Key.
     - I. The application can be read by malicious programs if plaintext, ciphertext, and keys are kept in memory.
     - II. It lacks secure memory for storing keys, making it vulnerable to memory dumps and related attacks.

B. Insecure key input: entering the private key as plain text in the GUI may expose it to potential exploits.

2. **Limitations of GUI (Graphic User Interface)**
   A. The GUI is basic at best and non-production setting friendly.
   B. The application does not support dynamic resizing nor customisation of the GUI layout.

3. **Static Key Generation**
   A. It dynamically generates keys allowing zero customisable parameters such as key sizes or cryptographic properties.

3. Implementation

This project implements an encryption system based on the RSA algorithm, featuring both key generation and secure message processing through a Graphical User Interface. The system is designed to support message confidentiality and correct decryption, with safeguards against common cryptanalytic attacks and platform limitations.

3.1 Key Generation and Assignment

The system generates the public and private keys following a series of steps. Firstly, two distinct large prime numbers, *p* and *q*, are generated using a random number generator and validated using the "*is_prime()*" function.

```python
def is_prime(x):

    if x < 2:
        return False
    if x in (2, 3):
        return True
    if x % 2 == 0 or x % 3 == 0:
        return False
    i = 5
    while i * i <= x:
        if x % i == 0 or x % (i + 2) == 0:
            return False
        i += 6
    return True

def Generate_Prime(minimum, maximum):
    while True:
        num = random.randint(minimum, maximum)
        if is_prime(num):
            return num
```

Figure 1. is_prime and Generate_Prime functions implementation

The public exponent *e* is fixed at 65537, a common and secure choice, while the private key exponent d is computed using the Extended Euclidean Algorithm to find the modular inverse of e mod φ(n). The modulus *n* is then computed as *p* * *q*, and Euler's totient function:

$$\varphi(n) = (p - 1) * (q - 1).$$

Finally, the decryption key *d* is calculated using the "*mod_inverse()*" function, implementing the Extended Euclidean Algorithm to ensure that *e* and *φ(n)* are coprime and that the inverse exists.

```python
def mod_inverse(e, phi):
    # Returns the modular inverse of e modulo phi using the Extended Euclidean Algorithm.
    def extended_gcd(a, b):
        # Extended Euclidean Algorithm. Returns (gcd, x, y) such that ax + by = gcd(a, b).
        if a == 0:
            return (b, 0, 1)
        else:
            g, x, y = extended_gcd(b % a, a)
            return (g, y - (b // a) * x, x)

    g, x, _ = extended_gcd(e, phi)
    if g != 1:
        raise ValueError(f"No modular inverse exists for e = {e} and phi = {phi}, as they are not coprime.")
    else:
        return x % phi  # Ensure the result is positive
```

Figure 2. mod_inverse function implementation

The implementation supports key sizes with n comfortably exceeding 100; we decided to apply a range between 100 and 100000 for numbers generation as it ensures practical cryptographic strength while remaining computationally feasible. The resulting keys are:

- **Public key:** (e, n)
- **Private key:** (d, n)

3.2 Encryption Procedure

Before encryption, the plaintext is processed by the "*Encryption_Preparation()*" function, which plays a critical role in enhancing both the security and structure of the data. This function converts each character in the input string to its ASCII equivalent, ensuring each code is formatted as a 3-digit string (e.g., 'A' → '065').

These codes are then grouped into blocks of three characters (i.e., nine digits total), forming larger integers that are more secure against frequency analysis attacks. Any remaining characters that do not fill a complete block are padded with leading zeros to maintain a uniform block size.

The resulting list of 9-digit integers forms the input to the RSA encryption algorithm. This preprocessing step ensures consistency in block size and strengthens the encryption process.

```
83  def Encryption_Preparation(Plaintext):
84      PlaintextAscii = []
85      for i in Plaintext:
86          PlaintextCh = str(ord(i))               # Converts to Ascii code each character
87
88          while(len(PlaintextCh) < 3):            # Adds zeros in the left to ascii codes which have less than 3 digits
89              PlaintextCh = '0' + PlaintextCh
90
91          PlaintextAscii.append(PlaintextCh)
92
93      PlaintextAscii3Ch = ''
94      PlaintextAscii3ChList = []
95      concat = 0
96
97      for i in PlaintextAscii:
98
99          if(concat < 3):
100             PlaintextAscii3Ch += i          # Joins the characters in groups of 3
101             concat += 1
102
103             if(len(PlaintextAscii3Ch) == 9):
104                 PlaintextAscii3ChList.append(PlaintextAscii3Ch)     # If a group is formed, it is added to the list to be encrypted
105                 PlaintextAscii3Ch = ''
106                 concat = 0
107
108
109     if PlaintextAscii3Ch:
110         PlaintextAscii3ChList.append(PlaintextAscii3Ch)             # Any remaining elements are added to the list to be encrypted
111
112     for i in range(len(PlaintextAscii3ChList)):
113         PlaintextAscii3ChList[i] = PlaintextAscii3ChList[i].rjust(9, '0')   # Adds zeros at the left so that every element has 9 digits
114
115     return [int(i) for i in PlaintextAscii3ChList]      #Converts each block of nunbers in integers
```

Figure 3. Encryption_Preparation function implementation

Once prepared, each numerical block is encrypted using the "*RSA_Encrypt()*" function, which follows the RSA encryption method:

$$Ciphertext = block^e \bmod n$$

This transformation is applied iteratively to all blocks, producing a list of ciphertext integers suitable for secure transmission or storage in the form of *[X, X, ….]* where X represents the encryption of each group of 3 characters.

3.3 Decryption Procedure

The "*RSA_Decrypt()*" function is responsible for reversing the encryption process using the private key. It decrypts each integer in the ciphertext list by computing $Block = Ciphertext^d \ mod \ n$, recovering the original 9-digit numerical representations of the plaintext. Each decrypted block is formatted as a string and split into 3-digit ASCII codes, where any leading '000' padding, added during encryption to ensure block consistency, is removed. Finally, these codes are then converted back into characters using their ASCII values.

```
53  def RSA_Decrypt(Ciphertext, PrivateKey):
54      d, n = PrivateKey
55
56      Splitted = []
57
58      try:
59          for i in Ciphertext:
60              # Decrypt and convert to a 9-character string, padding with leading zeros if necessary
61              Concat_plaintext = str(pow(i, d, n)).rjust(9, '0')
62              blocks = [Concat_plaintext[i:i+3] for i in range(0, len(Concat_plaintext), 3)]
63
64              # Delete '000' blocks from the beginning
65              while blocks and blocks[0] == '000':
66                  blocks.pop(0)
67
68              Splitted.extend(blocks)
69
70          Plaintext = ''
71          for code in Splitted:
72              char = chr(int(code))
73              if not (32 <= ord(char) <= 126):
74                  return "Error"
75              Plaintext += char
76
77      except Exception as e:
78          return "Incorrect Key, Impossible to decrypt."
79
80      return Plaintext
```

Figure 4. RSA_Decrypt function implementation

To ensure integrity, the function verifies that each recovered character falls within the printable ASCII range. If invalid characters are detected, or if an error occurs (e.g., due to a wrong key), the function gracefully returns an error message. Otherwise, it reconstructs and returns the original plaintext message, confirming the correctness and reversibility of the encryption scheme.

3.4 Graphical User Interface (GUI)

The GUI of the application provides an interactive environment for encrypting and decrypting text using the RSA algorithm. It features two main text areas: an input box on the left where the user can enter plaintext (for encryption) or ciphertext (for decryption), and an

output box on the right to display the resulting message. Between these two fields are buttons for the main operations: Encrypt, Decrypt, and Clean.

The Encrypt button converts plaintext into ciphertext using the current public key, while the Decrypt button restores the original message using the provided or selected private key. The Clean button clears both input and output fields to prepare for a new operation.

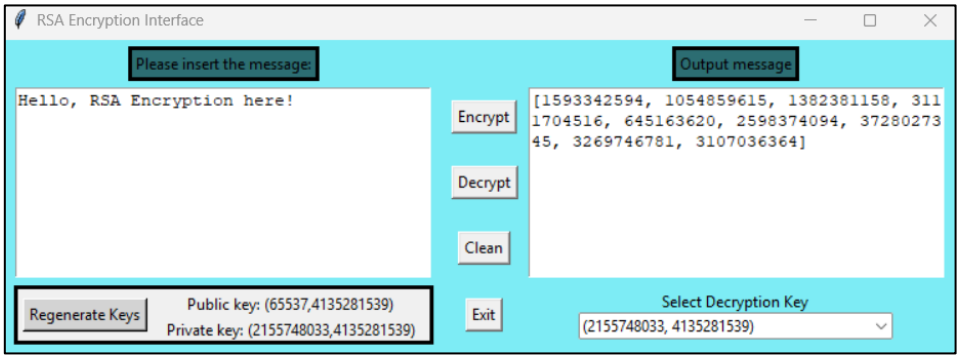

Figure 5. RSA encryption GUI

The GUI features dynamic key management, saving each generated private key in a history log that's accessible via a dropdown menu for easy reuse during decryption. Error handling is integrated throughout the interface; if any issue arises, such as invalid input, formatting errors, or incorrect keys, a pop-up window immediately alerts the user with a descriptive error message and avoiding crashes. This makes the application not only functional and secure but also user-friendly and resilient against common mistakes.

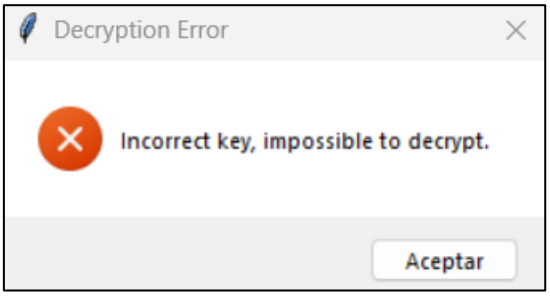

Figure 6. Error message box example

4. Conclusion

In this study, we explored the implementation of the RSA-based encryption and decryption program with the help of a graphical user interface. The program effectively demonstrates RSA's suitability for secure communication and key management through a clear, modular design (Stallings, 2017).

The software showcases dynamic RSA by encrypting with a public key and decrypting with a private key. Key features include an intuitive GUI, error handling, and modular RSA key generation. Its modular structure allows for well-defined, reusable, and maintainable code, making it a good example of both a working application and a learning tool for RSA encryption (Menezes et al., 1996; Paar and Pelzl, 2010). This creates several restrictions for the application; in terms of security threats such as storing plaintexts and keys in volatile memory, the lack of advanced features such as scalability or customization etc., and assumption of static keys without adjustable parameters/policies which limits the cryptographic flexibility of the application (Ferguson and Schneier, 2003).

To address these issues, the project implements RSA encryption and decryption with clear explanations of key processes such as prime number generation, modular arithmetic, and key pair creation (Boneh & Shoup, 2020). The system follows best practices, such as using a fixed public exponent (65537), a widely accepted and secure choice (NIST, 2016). Enhancing secure key input, memory protection, and GUI design would elevate the software to a production-ready level.

Overall, the app strikes a solid balance between usability and security, making it valuable for both practical use and cryptographic learning (Kaufman et al., 2011).

References

Binance Academy (2025) *Symmetric vs Asymmetric Encryption*. Available at: https://academy.binance.com/en/articles/symmetric-vs-asymmetric-encryption (Accessed: 24 April 2025).

Boneh, D. and Shoup, V. (2020) *A Graduate Course in Applied Cryptography*. Available at: https://crypto.stanford.edu/~dabo/cryptobook/ (Accessed: 10 April 2025).

Buchmann, J. (2004) *Introduction to Cryptography*. 2nd edn. New York: Springer.

Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. (2009) *Introduction to Algorithms*. 3rd edn. Cambridge, MA: MIT Press.

Ferguson, N. and Schneier, B. (2003) *Practical Cryptography*. Indianapolis: Wiley Publishing.

Kaufman, C., Perlman, R. and Speciner, M. (2016) *Network Security: Private Communication in a Public World*. 2nd edn. Upper Saddle River, NJ: Prentice Hall.

Menezes, A.J., van Oorschot, P.C. and Vanstone, S.A. (1996) *Handbook of Applied Cryptography*. Boca Raton, FL: CRC Press.

National Institute of Standards and Technology (NIST) (2016) *Recommendation for Key Management – Part 1: General (Special Publication 800-57 Rev. 4)*. Available at: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf (Accessed: 9 April 2025).

Paar, C. and Pelzl, J. (2010) *Understanding Cryptography: A Textbook for Students and Practitioners*. Berlin: Springer.

RSA Cryptosystem (2025) *Wikipedia*. Available at: https://en.wikipedia.org/wiki/RSA_cryptosystem (Accessed: 24 April 2025).

RSA Laboratories (2007) *PKCS #1 v2.1: RSA Cryptography Standard*. Available at: https://www.rsa.com/rsalabs/node.asp?id=2125 (Accessed: 10 March 2025).

Stallings, W. (2017) *Cryptography and Network Security: Principles and Practice*. 7th edn. Harlow: Pearson Education.