Detailed Explanation of RSA Encryption GUI in Python

Abdelrahman wael

Contents

1 Introduction

This report explains in detail the working of a Python program that implements the RSA cryptosystem from scratch and provides a simple Graphical User Interface (GUI) using Tkinter. The program demonstrates how text messages can be encrypted using a public key and decrypted using a private key.

The document covers:

- RSA mathematics (key generation, encryption, decryption).
- Line-by-line explanation of the Python code.
- Flow of data from message input to encrypted output and decrypted text.
- A visual block diagram to represent the flow.
- A solved example using small prime numbers.

—

2 RSA Background

RSA is an **asymmetric encryption algorithm**, meaning it uses two keys:

- **Public key** (n, e): Used to encrypt messages.
- **Private key** (n, d): Used to decrypt messages.

The basic steps of RSA are:

1. Choose two prime numbers p and q.
2. Compute $n = p \times q$.
3. Compute $\varphi(n) = (p-1)(q-1)$.
4. Choose e such that $gcd(e, \varphi(n)) = 1$.
5. Compute d as the modular inverse of $e \mod \varphi(n)$.

Encryption of a message m is done by:

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

Decryption is done by:

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

—

3 Program Code

The following Python code implements RSA from scratch and provides a GUI with Tkinter:

```
import tkinter as tk
from tkinter import scrolledtext, messagebox
import random

# --- RSA Implementation (No external libraries) ---
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

def modinv(a, m):
    # Extended Euclidean Algorithm for modular inverse
    m0, x0, x1 = m, 0, 1
    while a > 1:
        q = a // m
        a, m = m, a % m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1

def is_prime(n, k=20):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False
    # Miller-Rabin test
    r, s = 0, n - 1
    while s % 2 == 0:
        r += 1
        s //= 2
    for _ in range(k):
        a = random.randrange(2, n - 1)
        x = pow(a, s, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True

def generate_prime(bits=512):
    while True:
        num = random.getrandbits(bits)
```

```
        if is_prime(num):
            return num

def generate_keys(bits=512):
    p = generate_prime(bits)
    q = generate_prime(bits)
    n = p * q
    phi = (p - 1) * (q - 1)
    e = 65537
    if gcd(e, phi) != 1:
        return generate_keys(bits)
    d = modinv(e, phi)
    return (n, e), (n, d)

# Encrypt/Decrypt
def rsa_encrypt(msg, pubkey):
    n, e = pubkey
    return [pow(ord(c), e, n) for c in msg]

def rsa_decrypt(cipher, privkey):
    n, d = privkey
    return ''.join([chr(pow(c, d, n)) for c in cipher])

# --- Generate keys ---
public_key, private_key = generate_keys(16)  # small for demo

# --- Functions for GUI ---
def encrypt_message():
    try:
        msg = entry_message.get("1.0", tk.END).strip()
        cipher = rsa_encrypt(msg, public_key)
        entry_encrypted.delete("1.0", tk.END)
        entry_encrypted.insert(tk.END, " ".join(map(str, cipher))
           )
    except Exception as e:
        messagebox.showerror("Error", str(e))

def decrypt_message():
    try:
        cipher_text = entry_encrypted.get("1.0", tk.END).strip()
        cipher = list(map(int, cipher_text.split()))
        plain = rsa_decrypt(cipher, private_key)
        entry_decrypted.delete("1.0", tk.END)
        entry_decrypted.insert(tk.END, plain)
    except Exception as e:
        messagebox.showerror("Error", str(e))

# --- GUI ---
root = tk.Tk()
root.title("RSA Encryption (Pure Python)")
```

```
# Show keys
tk.Label(root, text="Public Key (n, e):").pack()
entry_public = scrolledtext.ScrolledText(root, height=3, width
    =70)
entry_public.pack()
entry_public.insert(tk.END, str(public_key))

tk.Label(root, text="Private Key (n, d):").pack()
entry_private = scrolledtext.ScrolledText(root, height=3, width
    =70)
entry_private.pack()
entry_private.insert(tk.END, str(private_key))

# Message input
tk.Label(root, text="Enter Message:").pack()
entry_message = scrolledtext.ScrolledText(root, height=4, width
    =50)
entry_message.pack()

# Buttons
tk.Button(root, text="Encrypt with Public Key", command=
    encrypt_message, bg="lightblue").pack(pady=5)
tk.Button(root, text="Decrypt with Private Key", command=
    decrypt_message, bg="lightgreen").pack(pady=5)

# Encrypted text
tk.Label(root, text="Encrypted (Numbers):").pack()
entry_encrypted = scrolledtext.ScrolledText(root, height=6, width
    =50)
entry_encrypted.pack()

# Decrypted text
tk.Label(root, text="Decrypted:").pack()
entry_decrypted = scrolledtext.ScrolledText(root, height=4, width
    =50)
entry_decrypted.pack()

root.mainloop()
```

—

4 Solved Example

To better illustrate the RSA encryption and decryption process, let us work through a small example using very small primes. (These values are chosen only for demonstration; real systems use primes with hundreds of digits.)

Step 1: Choose primes

$$p = 11, \quad q = 13$$

$$n = p \times q = 11 \times 13 = 143$$
$$\varphi(n) = (p-1)(q-1) = 10 \times 12 = 120$$

Step 2: Choose public exponent e

Let
$$e = 7$$
Check that $\gcd(e, \varphi(n)) = \gcd(7, 120) = 1$.

Step 3: Compute private exponent d

We need
$$d \equiv e^{-1} \pmod{120}$$
That is, solve $7d \equiv 1 \pmod{120}$. By Extended Euclidean Algorithm:
$$d = 103$$

Step 4: Public and private keys

$$\text{Public key: } (n, e) = (143, 7)$$
$$\text{Private key: } (n, d) = (143, 103)$$

Step 5: Encrypt a message

Suppose the message is:
$$m = 9$$
Ciphertext:
$$c = m^e \bmod n = 9^7 \bmod 143 = 48$$

Step 6: Decrypt the message

$$m = c^d \bmod n = 48^{103} \bmod 143 = 9$$

Summary

$$m = 9 \quad \xrightarrow{\text{Encrypt}} \quad c = 48 \quad \xrightarrow{\text{Decrypt}} \quad m = 9$$

—

5 Conclusion

This program demonstrates the RSA encryption and decryption process through an interactive Tkinter GUI. It allows users to:

- View public and private keys.
- Enter a message and encrypt it.
- Decrypt the ciphertext back to the original message.

Although the implementation uses very small primes (for speed), it effectively shows the concepts of asymmetric encryption, modular arithmetic, and key pairs.