RSA-OAEP Algorithm and Steganography Report

Raihan Bin Mofidul

Contents

1 RSA-OAEP Algorithm

1.1 Algorithm Overview

RSA (Rivest-Shamir-Adleman) is a widely used public-key cryptographic system. RSA-OAEP (Optimal Asymmetric Encryption Padding) enhances RSA by adding padding to prevent certain types of attacks, making the encryption more secure.

1.2 Mathematical Formulas and Variables

1.2.1 Key Generation

- **Variables:**
  - p: A large prime number
  - q: Another large prime number
  - n: The modulus, product of p and q
  - $\phi(n)$: Euler's totient function of n
  - e: Public exponent, an integer
  - d: Private exponent, computed as the modular multiplicative inverse of e modulo $\phi(n)$
- **Steps:**
  1. Generate two large prime numbers p and q.
  2. Compute $n = p \times q$.
  3. Compute $\phi(n) = (p - 1) \times (q - 1)$.
  4. Choose an integer e such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$.
  5. Compute d as the modular multiplicative inverse of e modulo $\phi(n)$.
- **Formulas:**

$$n = p \times q$$
$$\phi(n) = (p - 1) \times (q - 1)$$
$$e \times d \equiv 1 \pmod{\phi(n)}$$

1.2.2 Encryption

- **Variables:**
  - m: Plaintext message converted to an integer
  - c: Ciphertext
  - (e, n): Public key
- **Steps:**
  1. Convert the plaintext message into an integer m such that $0 \leq m < n$.
  2. Compute the ciphertext c using the public key (e, n).
- **Formula:**

$$c = m^e \pmod{n}$$

1.2.3 Decryption

- **Variables:**
  - c: Ciphertext
  - m: Plaintext message
  - (d, n): Private key
- **Steps:**
  1. Compute the plaintext message m using the private key (d, n).
- **Formula:**

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

1.3 Flow Chart

- **Key Generation:**
  1. Start
  2. Generate p and q
  3. Compute n and $\phi(n)$
  4. Choose e
  5. Compute d
  6. Output (e, n) and (d, n)
  7. End
- **Encryption:**
  1. Start
  2. Input plaintext message
  3. Convert message to integer m
  4. Compute ciphertext c
  5. Output ciphertext
  6. End
- **Decryption:**
  1. Start
  2. Input ciphertext
  3. Compute plaintext message m
  4. Convert integer m to message
  5. Output plaintext message
  6. End

1.4 Detailed Explanations

- **Key Generation:** The process involves generating two large prime numbers and calculating the modulus and totient. The public and private keys are derived to allow secure encryption and decryption.
- **Encryption:** The plaintext message is converted to an integer and encrypted using the public key. Padding (OAEP) is applied to the message to ensure security against attacks.
- **Decryption:** The ciphertext is decrypted using the private key, reversing the encryption process to retrieve the original message.

2 4096-bit RSA Encryption

2.1 Importance of 4096-bit Key Size

- **Enhanced Security:** Larger key sizes offer stronger encryption, making it significantly harder for attackers to break the encryption through brute force or other methods.
- **Future-proofing:** As computational power increases, larger key sizes ensure that encryption remains secure for a longer period.
- **Regulatory Compliance:** Certain industries and government regulations require the use of larger key sizes for sensitive data.

2.2 Specific Use in RSA

- **Public Key (e):** Often a small value like 65537 to optimize encryption speed.
- **Private Key (d):** A large value computed as the modular inverse of e modulo $\phi(n)$.

3 Least Significant Bit (LSB) Steganography

3.1 Overview

LSB steganography involves hiding data within the least significant bits of image pixel values. It is a simple yet effective method for concealing information within an image without significantly altering its appearance.

3.2 Process

- **Hiding Data:**
  1. Convert the data to be hidden into a binary format.

2. Modify the least significant bit of each pixel's color value to match the binary data.
3. Save the modified image.

- **Revealing Data:**
  1. Extract the least significant bits from each pixel's color value.
  2. Combine these bits to reconstruct the hidden binary data.
  3. Convert the binary data back to its original format.

3.3 Detailed Explanation

- **Data Conversion:** The data is first converted to a binary format. Each byte of data is represented by 8 bits.
- **Pixel Modification:** The LSB of each color channel (Red, Green, Blue) in the image pixels is modified to store the binary data. This results in minimal visual changes to the image.
- **Data Extraction:** The hidden data is retrieved by extracting the LSB from each pixel and reconstructing the binary data.

4 Code Explanation

4.1 RSA Key Generation

```
def generate_rsa_keys():
    new_key = RSA.generate(4096)
    public_key = new_key.publickey().exportKey("PEM")
    private_key = new_key.exportKey("PEM")
    return public_key, private_key
```

- **Function:** Generates RSA keys with a size of 4096 bits.
- **Output:** Returns the public and private keys in PEM format.

4.2 RSA Encryption

```
def rsa_encrypt(public_key, message):
    rsa_public_key = RSA.importKey(public_key)
    rsa_public_key = PKCS1_OAEP.new(rsa_public_key)
    encrypted_message = rsa_public_key.encrypt(message)
    return binascii.hexlify(encrypted_message)
```

- **Function:** Encrypts a message using the RSA public key and OAEP padding.
- **Output:** Returns the encrypted message in hexadecimal format.

4.3 RSA Decryption

```
def rsa_decrypt(private_key, encrypted_message):
    rsa_private_key = RSA.importKey(private_key)
    rsa_private_key = PKCS1_OAEP.new(rsa_private_key)
    decrypted_message = rsa_private_key.decrypt(binascii.unhexlify(encrypted_mes
    return decrypted_message
```

- **Function:** Decrypts an encrypted message using the RSA private key.
- **Output:** Returns the decrypted message.

4.4 Random Image Generation

```
def generate_random_image(width, height):
    random_image = Image.new('RGB', (width, height))
    for x in range(width):
        for y in range(height):
            random_color = (random.randint(0, 255), random.randint(0, 255), rando
            random_image.putpixel((x, y), random_color)
    return random_image
```

- **Function:** Generates a random image with specified width and height.
- **Output:** Returns the generated image object.

4.5 Steganography - Hiding Data

```
def hide_data_in_image(image, data, output_path):
    encoded_image = image.copy()
    width, height = image.size
    index = 0

    binary_data = ''.join(format(byte, '08b') for byte in data)
    data_len = len(binary_data)

    for x in range(width):
        for y in range(height):
            pixel = list(image.getpixel((x, y)))
            for n in range(0, 3):
                if index < data_len:
                    pixel[n] = pixel[n] & ~1 | int(binary_data[index])
                    index += 1
                else:
                    break
```

```
            encoded_image.putpixel((x, y), tuple(pixel))
            if index >= data_len:
                break
        if index >= data_len:
            break

    encoded_image.save(output_path)
    return output_path
```

- **Function:** Hides binary data within the least significant bits of the image pixels.
- **Output:** Saves and returns the path of the image with hidden data.

4.6 Steganography - Revealing Data

```
def reveal_data_in_image(image_path, expected_data_length):
    image = Image.open(image_path)
    binary_data = ""
    for x in range(image.size[0]):
        for y in range(image.size[1]):
            pixel = list(image.getpixel((x, y)))
            for n in range(0, 3):
                if len(binary_data) < expected_data_length * 8:
                    binary_data += str(pixel[n] & 1)
                else:
                    break
            if len(binary_data) >= expected_data_length * 8:
                break
        if len(binary_data) >= expected_data_length * 8:
            break

    all_bytes = [binary_data[i: i+8] for i in range(0, len(binary_data), 8)]
    decoded_data = bytes([int(byte, 2) for byte in all_bytes])
    return decoded_data
```

- **Function:** Extracts hidden binary data from the least significant bits of the image pixels.
- **Output:** Returns the extracted binary data.