# 

# 

# 

# 

# **Alternative Public-Key Encryption System**

Student’s Name

Institution

Course

Date

# **Alternative Public-Key Encryption System Implementation and Analysis**

## **1. Introduction**

This report describes an improved system of public key encrypting, describing procedures for generating keys using specific mathematical properties. The system is in fact a version of asymmetric cryptography and was aimed at fast communication over secured channels with help of the public and private keys. This implementation also describes how to implement key generation, encryption and decryption using modular arithmetic and integer sequence. This report contains the theoretical overview of the key generation process with the concrete encryption and decryption algorithms and the step-by-step description of the system’s implementation with the pseudocode for this implementation and the executable python code to check the output.

## **2. Theoretical Framework**

### **2.1 Key Generation Process**

The generation process of keys in this system is systematic and thus embraces the issue of security. It starts with a set of interger numbers and each number must be the sum of consecutive preceding numbers. A prime number *q* is then chosen which is greater than twice the largest element in the sequence such that, *q* > 2 \* max A;The public key *h* is computed by the formula hi=(w⋅ei) mod q where *w* is a randomly generated number that is coprime to *q*. This process guarantees the creation of a unique public key, while the private key consists of the sequence eee, the prime *q*, and the multiplier *w* (Umapathy & Kalpana, 2023).

### **2.2 Encryption and Decryption Mechanisms**

The encryption changes the plain message by utilizing the mathematical product of the message with elements of the public key where those elements are summed to give the cipher message. The decryption process uses especially the inverse of *w*, which is the private key, to produce the message. It means that the modular arithmetic allows performing the work in bits, and with capturing each integer within the sequence, the pile builds a message like a puzzle. This method makes it possible to decode the message by only a person holding the private key, and is therefore hard to decode without the key (Raj & Sridhar, 2021).

## **3. Implementation**

### **3.1 Pseudo-code**

The key generation function first defines an increasing sequence of integers and then choose a prime number greater than (2 times the maximum of the integer sequence). A random multiplier, that is relatively prime with the prime number is then used to come up with the public key. The encryption function sums the products of the message bits and the public key arriving at the ciphertext. The decryption function performing operations inversely to the random multiplier within the properties of the sequence to generate an extract of the original message.

**Function** GenerateKeyPair(n):

1. Initialize an empty list e and a variable sum = 0.
2. **For** i from 1 to n:
   * Generate a random integer next\_value such that next\_value > sum.
   * Append next\_value to e.
   * Update sum = sum + next\_value.
3. Select a prime number q such that q > 2 \* e[n].
4. Generate a random integer w such that gcd(w, q) = 1.
5. Initialize an empty list h.
6. **For** i from 1 to n:
   * Compute h[i] = (w \* e[i]) mod q.
7. Return the public key h and the private key (e, q, w).

**Function** Encrypt(message, public\_key):

1. Initialize ciphertext = 0.
2. **For** i from 1 to n:
   * Update ciphertext = ciphertext + message[i] \* public\_key[i].
3. Return ciphertext.

**Function** Decrypt(ciphertext, private\_key):

1. Extract e, q, and w from the private key.
2. Compute w\_inverse = modular\_inverse(w, q).
3. Compute c\_prime = (ciphertext \* w\_inverse) mod q.
4. Initialize an empty list message.
5. **For** i from n down to 1:
   * **If** c\_prime >= e[i]:
     + Set message[i] = 1.
     + Update c\_prime = c\_prime - e[i].
   * **Else**:
     + Set message[i] = 0.
6. Return message.

### **3.2 Python Code Implementation**

The implemented Python code corresponds to the structure presented in the pseudo-code presented above. It starts with the CryptoSystem class containing methods to generate a strictly increasing sequence of integers, to look whether the number is prime or not, to find modular inverse and finally to generate both public and private keys. The encrypt method calculates the ciphertext by using sum of the product of the message and some parameters of the public key. The decrypt method reverses the process, it uses the inverse of the multiplier and loops through private key integer representation. Some of the codes mentioned above can be run to generate the key pairs, encrypt a message and then decrypt the ciphertext thus proving that the system can actually work.

#### **Code Snippet** class CryptoSystem:

def \_\_init\_\_(self, n):

self.n = n

self.public\_key = None

self.private\_key = None

def generate\_increasing\_sequence(self):

"""Generates sequence where each element is greater than sum of previous elements."""

sequence = []

current\_sum = 0

for \_ in range(self.n):

next\_val = random.randint(current\_sum + 1, 2 \* current\_sum + 10)

sequence.append(next\_val)

current\_sum += next\_val

return sequence

def is\_prime(self, n):

"""Checks if number is prime."""

if n < 2:

return False

for i in range(2, int(n \*\* 0.5) + 1):

if n % i == 0:

return False

return True

def generate\_prime(self, min\_value):

"""Generates prime number greater than min\_value."""

candidate = min\_value + 1

while not self.is\_prime(candidate):

candidate += 1

return candidate

def mod\_inverse(self, a, m):

"""Calculates modular multiplicative inverse."""

def extended\_gcd(a, b):

if a == 0:

return b, 0, 1

gcd, x1, y1 = extended\_gcd(b % a, a)

x = y1 - (b // a) \* x1

y = x1

return gcd, x, y

\_, x, \_ = extended\_gcd(a, m)

return (x % m + m) % m

def generate\_keys(self):

"""Generates public and private key pair."""

e = self.generate\_increasing\_sequence()

q = self.generate\_prime(2 \* e[-1])

while True:

w = random.randint(2, q-1)

if gcd(w, q) == 1:

break

h = [(w \* e\_i) % q for e\_i in e]

self.public\_key = h

self.private\_key = (e, q, w)

return h, (e, q, w)

def encrypt(self, message):

"""Encrypts a binary message."""

if len(message) != self.n:

raise ValueError(f"Message must be {self.n} bits long")

if not self.public\_key:

raise ValueError("Generate keys first")

c = sum(h\_i \* m\_i for h\_i, m\_i in zip(self.public\_key, message))

return c

def decrypt(self, ciphertext):

"""Decrypts a ciphertext back to a binary message."""

if not self.private\_key:

raise ValueError("Generate keys first")

e, q, w = self.private\_key

w\_inv = self.mod\_inverse(w, q)

c\_prime = (ciphertext \* w\_inv) % q

message = []

for e\_i in reversed(e):

if c\_prime >= e\_i:

message.insert(0, 1)

c\_prime -= e\_i

else:

message.insert(0, 0)

return message

def get\_binary\_message():

"""Prompts the user for a message and converts it to a binary list."""

message = input("Enter a message (up to 8 characters): ").strip()

if len(message) > 8:

raise ValueError("Message length must not exceed 8 characters.")

binary\_message = []

for char in message:

binary\_rep = format(ord(char), '08b') # Convert each character to binary (8 bits)

binary\_message.extend([int(bit) for bit in binary\_rep])

return binary\_message[:8] # Limit to 8 bits

def binary\_to\_string(binary\_message):

"""Converts a binary message back to a string."""

chars = []

for i in range(0, len(binary\_message), 8):

byte = binary\_message[i:i+8]

chars.append(chr(int(''.join(map(str, byte)), 2)))

return ''.join(chars)

## **4. Testing and Results**

To check how it works, the key pair consisting of two sequences having the length of eight integers were generated for practice. Sample plaintext of binary numbers of simple message with a combination of 1’s and 0’s was encrypted and a ciphertext was obtained. Next the ciphertext was decrypted then the result was the msg confirming the correctness of the implemented system. Finally, the generation of the rank, encryption and decryption were accomplished and this showed that the system has the capability of operating as required.

## **5. Security Analysis**

Security of this encryption system lies in the ability to factor the public key components from which the private key could be derived from. Also, the employment of a prime number and the modular arithmetic makes the attack all the more difficult. Nevertheless, the identified features do not make the system immune to some computational attacks and, thus, are insufficient for achieving high practical security levels (Ivanov & Stoianov, 2023). In light of this, the possibilities of distinct mathematical relations in the elements of the public key allow them to be unsuitable for secure communication relating to real world applications.

## **6. Conclusion**

In this project, we implement an alternative public key encryption system, which is based on a novel approach to cryptography, where the method for key generation and the method for message encryption is different. The system works fairly well functionally, but its security shortcomings makes it a bad choice for production use. However, it is an interesting educational tool to learn what key generation, encryption and decryption do in public key cryptography.

References

Ivanov, A., & Stoianov, N. (2023). Implications of the Arithmetic Ratio of Prime Numbers for RSA Security. *International Journal of Applied Mathematics and Computer Science*, 33, 57 - 70. <https://doi.org/10.34768/amcs-2023-0005>.

Raj, B., & Sridhar, V. (2021). Identity Based Cryptography Using Matrices. *Wireless Personal Communications*, 120, 1637 - 1657. <https://doi.org/10.1007/s11277-021-08526-9>.

Umapathy, B., & Kalpana, G. (2023). A Key Generation Algorithm for Cryptographic Algorithms to Improve Key Complexity and Efficiency. 2023 5th International Conference on Smart Systems and Inventive Technology (ICSSIT), 647-652. <https://doi.org/10.1109/ICSSIT55814.2023.10060906>.

#### 