CRYPTOGRAPHY

NAME :- Pratham Kumar
ROLL NO :- 24B0985
BRANCH :- CSE
EMAIL-ID :- 24b0985@iitb.ac.in

Mentor :- Param rathour

GitHub Repository

June 14, 2025

Contents

1 Week 1: Classical Cryptography & Number Theory Essentials

In the early days of secure communication, classical ciphers were the foundation of cryptographic practice. Among the most well-known are substitution and transposition ciphers, which form the basis for understanding how plaintext can be transformed into ciphertext.

1.1 Caesar Cipher

The Caesar cipher is a monoalphabetic substitution cipher where each letter in the plaintext is shifted by a fixed number of positions in the alphabet. e.g. with a shift of 3, 'A' becomes 'D', 'B' becomes 'E', and so on. This method is simple and easy to break due to its limited key space (only 25 possible shifts for the English alphabet). Despite its simplicity, it introduces the essential idea of encryption in the earlier days founding the base of modern substitution cryptography techniques.

Caesar Cipher Implementation

The following Python code demonstrates Caesar cipher encryption and decryption.

```python
def caesar_cipher(text, shift, mode='encrypt'):
    cipher_text = ''

# Ensure shift is between 0 to 25
    shift = shift % 26
    if mode == 'decrypt':
        shift = -shift

    for char in text:
        if char.isalpha():
            base = ord('A') if char.isupper() else ord('a')
            cipher = (ord(char) - base + shift) % 26 + base
            cipher_text += chr(cipher)
        else:
            # Leave non-alphabetic characters unchanged
            cipher_text += char

    return cipher_text
```

1.2 Vigenère Cipher

The Vigenère cipher is the advanced substitution cipher that improves on Caesar by using a keyword to apply different shifts to different letters of the plaintext. For example, with the keyword "PRATHAM", each letter of the message is shifted by the value corresponding to a letter in the keyword, cycling through the keyword repeatedly. This polyalphabetic nature makes *frequency analysis* more difficult compared to Caesar, though it's still vulnerable with enough ciphertext because of repetition of patterns.
This cipher was looking like an unbreakable code and was continued to be used for a very long time for encryption.It was broken was by *Charles Babbage*. He broke this using the periodic nature of the cipher and once the key length is found, each part can be solved using frequency analysis.

Vigenère Cipher Implementation

The following Python code demonstrates Vigenère cipher encryption and decryption.

```python
def vigenere_cipher(text, key, mode='encrypt'):
    result = ''
    key_length = len(key)
    key_as_int = [ord(k) - ord('A') for k in key.upper()]
    key_index = 0

    for char in text:
        if char.isalpha():
            is_upper = char.isupper()
            reference = ord('A') if is_upper else ord('a')
            text_int = ord(char) - reference
            key_shift = key_as_int[key_index % key_length]
            if mode == 'decrypt':
                key_shift = -key_shift
            shifted = (text_int + key_shift) % 26
            result += chr(shifted + reference)
            key_index += 1
        else:
            result += char

    return result
```

To understand and analyze such ciphers, *modular arithmetic* is used. Modular arithmetic works with the remainder of division and is denoted by expressions like $a \equiv b \pmod{n}$, meaning a and b leave the same remainder when divided by n. This arithmetic underlies many cryptographic algorithms, particularly in creating mathematical functions that are easy to compute but hard to reverse.(One-way functions also known as trapdoor functions).

1.3 THE ENIGMA CIPHER

The Enigma machine is an electro-mechanical rotor cipher machine invented and used extensively by Nazi Germany during World War II for encrypting military communications. Unlike older substitution ciphers like Caesar or Vigenère, Enigma used a sophisticated arrangement of several rotating cipher wheels (known as rotors), a plugboard (Steckerbrett), and a reflector which vastly increased the potential configurations.

1.3.1 Components

1. Keyboard input
2. Plugboard substitution (swapping letters in pairs)
3. Rotor scrambling (with rotors rotating after each key press, creating a polyalphabetic substitution that changed dynamically).
4. Reflector (ensuring the encryption was reciprocal: if A became G, then G became A when decrypted).
5. Reverse rotor and plugboard path, lighting up the ciphertext letter.

This system meant that the same letter was never encrypted the same way twice in a message—an enormous advance over classical ciphers.

1.3.2 Breaking the ENIGMA - A Historic Cryptanalytic Triumph

The Enigma was first broken in 1932 by *Marian Rejewski*, a Polish mathematician and cryptanalyst. Using group theory and access to some stolen German documents, he reverse-engineered the wiring of the Enigma machine. Rejewski, along with *Jerzy Różycki and Henryk Zygalski*, developed the first techniques and mechanical devices (like the bomba kryptologiczna) to automate parts of the decryption.

In 1939, the Polish shared their findings with British intelligence. Building on this, *Alan Turing* and *Gordon Welchman* at Bletchley Park developed a more advanced electromechanical machine known as the Bombe, which could discover the daily settings of the Enigma machine much faster.

1.3.3 Weakness of ENIGMA

- No letter ever encrypted to itself: because of the reflector component,reflecting the signal to the other pathway.
- Repeating daily keys: Early procedures required sending the same key twice at the beginning of the message (e.g., "ABLABL"), giving cryptanalysts patterns to look for the setup.
- Predictable message content (cribs): Standard phrases like "WETTER" (weather) were used to guess plaintext segments.

2 Week 2: Modular Arithmetic and Symmetric Key Encryption

This Cryptanalysts make deeper understanding of the modular arithmetic, particularly the concept of modular inverses and their computation using the Extended Euclidean Algorithm. While the basic Euclidean algorithm finds the greatest common divisor (GCD) of two numbers, its extended version also finds the coefficients (integers) that express this GCD as a linear combination of the inputs which directly leads to the computation of modular inverses.

Bezout's Identity: For any two non-zero integers a and b, let $d = \gcd(a, b)$ be their greatest common divisor. Then there exist integers x and y such that:

$$\mathbf{ax + by = d}$$

Here, d is also the smallest positive integer that can be expressed in this linear combination.

2.1 Fermat's Little Theorem

Fermat's Little Theorem: If p is a prime number, then for any integer a not divisible by p, we have:

$$a^{p-1} \equiv 1 \pmod{p}$$

Furthermore, for any integer a, we always have:

$$a^p \equiv a \pmod{p}$$

This theorem helps in modular exponentiation and is greatly used in many public key systems later on. It is particularly useful in cryptography for primality testing and as a basis for algorithms like RSA, where operations are performed modulo a large composite number, which itself is a product of large primes. Its simplicity and power make it a one of the most important number theory applications in secure communication.

2.2 Modular Inverse

Now It is the time for one more key concept *modular inverse:* to talk about,for a number a and a modulus m, the modular inverse is a number x such that $a \cdot x \equiv 1 \pmod{m}$. This is central in decryption processes where one must reverse a multiplication or exponentiation operation under a modulus.

Modular Inverse Implementation

The following Python code demonstrates modular inverse method of encryption and decryption.

```python
def modular_inverse(a, m):
    def extended_gcd(a, b):
        if b == 0:
            return a, 1, 0  # gcd, x, y such that ax + by = gcd
        gcd, x1, y1 = extended_gcd(b, a % b)
        x = y1
        y = x1 - (a // b) * y1
        return gcd, x, y

    gcd, x, y = extended_gcd(a, m)

    if gcd != 1:
        print("Modular inverse does not exist because a and m are not coprime.")
        return None
    else:
        return x % m  # Return the positive inverse
```

2.3 DES and AES

Now ,the number theory essentials make an introduction to symmetric key encryption, where the same key is used for both encryption and decryption. A major type of symmetric ciphers is block ciphers, which encrypt fixed-size blocks of data.

Two classic examples of block ciphers are **DES (Data Encryption Standard)** and **AES (Advanced Encryption Standard)**. DES works with 64-bit blocks and uses a 56-bit key. It goes through a series of substitution and permutation operations over 16 rounds. While it was significant in the past, DES is no longer secure because of its short key length.

AES is the modern replacement and offers much stronger security. It uses 128-bit blocks and supports key sizes of 128, 192, or 256 bits. AES operates over several rounds, performing tasks like substitution, shifting rows, mixing columns, and adding round keys.

Feature	**DES**	**AES**
Block Size	64 bits	128 bits
Key Size	56 bits	Varies - 128/192/256 bits
Structure	Feistel Netowrk	Substitution-Permutation
Speed	slower than AES	Faster standard
Data Security	Weaker	Strong

3 Week 3: Public Key Cryptography

Public key cryptography, also known as **asymmetric cryptography**, is a revolutionary approach to secure communication. Unlike symmetric cryptography, where both parties share a single secret key, asymmetric systems use a **key pair** — a **public key** (shared openly) and a **private key** (kept secret). The security relies on mathematical problems that are easy to compute in one direction but computationally hard to do it backwards without specific information. These problems are the foundation of **one-way functions**.

3.1 RSA Cryptosystem

The **RSA** (Rivest–Shamir–Adleman) cryptosystem is one of the most well-known public key algorithms. Its security is based on the difficulty of **factoring large composite numbers**, specifically the product of two large primes.

RSA Key Generation

- Choose two large distinct prime numbers, p and q.
- Compute the modulus $n = p \cdot q$. This value n is used in both encryption and decryption and is part of the public key.
- Calculate Euler's totient function:
$$\phi(n) = (p-1)(q-1)$$
- Choose an encryption exponent e such that:
$$1 < e < \phi(n), \quad \gcd(e, \phi(n)) = 1$$
- Compute the decryption exponent d, which is the modular inverse of $e \mod \phi(n)$:
$$d \equiv e^{-1} \mod \phi(n)$$
Which is :
$$e \cdot d \equiv 1 \mod \phi(n)$$
- The **public key** is the pair (n, e).
- The **private key** is the integer d(kept secret).

3.1.1 RSA Encryption

To encrypt a message M, where $M < n$, compute the ciphertext C:

$$C = M^e \mod n$$

3.1.2 RSA Decryption

To decrypt the ciphertext C, use the private key d:

$$M = C^d \mod n$$

This works due to Euler's theorem and the mathematical properties of modular exponentiation.

3.1.3 Why RSA is Secure

The hardness of deducing the private key d from the public key (n, e) without knowing p and q stems from the fact that:

- Factoring n into its prime components is computationally difficult.
- Without knowing $\phi(n)$, computing d becomes infeasible because modular inverse calculation is computationally difficult.

This creates a one-way function — easy to compute (encryption), but hard to invert (decryption without the private key).

RSA Key Generation Implementation

The following Python code demonstrates RSA key generation process for encryption and decryption .

```python
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

def modinv(a, m):
    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

p = int(input("Enter prime number p: "))
q = int(input("Enter prime number q: "))
m = int(input("Enter the message (an integer less than p*q):
   "))

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

# Step 1: Choose encryption key e
# We'll pick a small e like 3, 5, 7... until gcd(e, phi) == 1
e = 3
while gcd(e, phi) != 1:
    e += 2

# Step 2: Compute private key d
d = modinv(e, phi)

# Step 3: Encrypt and decrypt
c = pow(m, e, n)              # Encrypted message
m_decrypted = pow(c, d, n) # Decrypted message

print(f"\nPublic Key (e, n): ({e}, {n})")
print(f"Private Key (d, n): ({d}, {n})")
print(f"Encrypted Message: {c}")
print(f"Decrypted Message: {m_decrypted}")
```

3.2 Diffie–Hellman Key Exchange

The **Diffie–Hellman key exchange** (proposed in 1976) is primarily designed to securely establish a **shared secret key** over an insecure channel. It was the first practical implementation of public key principles and is the foundation of secure key exchange in many modern protocols.

3.2.1 The Idea

Let p be a large prime and g be a primitive root modulo p (i.e., a generator of the multiplicative group modulo p).

- Alice selects a secret integer a and sends $A = g^a \mod p$ to Bob.
- Bob selects a secret integer b and sends $B = g^b \mod p$ to Alice.
- Both compute the shared key:
  - Alice computes $K = B^a \mod p$
  - Bob computes $K = A^b \mod p$

  By properties of modular exponentiation:

$$K = (g^b)^a \mod p = (g^a)^b \mod p = g^{ab} \mod p$$

Hence, both parties compute the same shared key K without ever transmitting it directly.

3.2.2 Why is it Secure?

An eavesdropper can see p, g, A, and B, but cannot efficiently compute g^{ab} without knowing either a or b. This is known as the **Discrete Logarithm Problem**, which is computationally hard for large primes.

4 Week 4: Primality Testing and Factoring

The foundation of RSA and similar cryptosystems relies on large prime numbers. Thus, primality testing is a crucial part of generating cryptographic keys.

The Fermat Primality Test is one of the simplest methods. It is based on Fermat's Little Theorem and checks whether $a^{n-1} \equiv 1 \pmod{n}$ holds for some base a. However, some composite numbers, known as **Carmichael numbers**, can still pass this test for many bases, which makes it unreliable for strong security.

To address this issue, the **Miller-Rabin Primality Test** provides a probabilistic method that tests whether a number is composite with high confidence. It breaks down $n-1$ into $2^s \cdot d$ and checks certain modular exponentiation conditions across random bases a. If any condition fails, then n is definitely composite. If n passes multiple rounds, it is likely prime with high probability.

For factoring, which is the inverse problem of RSA, several algorithms are available. Trial division is the most basic but not efficient. A more sophisticated option is Pollard's rho algorithm, which is a clever probabilistic technique that uses Floyd's cycle detection to find nontrivial factors of a number. While it may not be efficient enough to break RSA-level numbers, it shows how number-theoretic patterns can be used effectively in the modern cryptography era.

Primality Test Implementation

The following Python code demonstrates Primality Test method used in encryption and decryption techniques.

```python
import random

def is_prime(n, k=5):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False

    r, d = 0, n - 1
    while d % 2 == 0:
        d //= 2
        r += 1

    for _ in range(k):
        a = random.randrange(2, n - 2)
        x = pow(a, d, 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  # composite

    return True  # probably prime
```

— MIDTERM CHECKPOINT —

5 Theory Summary (Weeks 1–4)

Number Theory Fundamentals

- Congruences: $a \equiv b \mod m$, modular arithmetic rules
- Euler's Totient Function: $\phi(n)$ and its use in RSA
- Inverses: If $\gcd(a, m) = 1$ then $\exists\ a^{-1}$ such that $aa^{-1} \equiv 1 \mod m$

Cryptographic Security Concepts

- Confidentiality: Ensuring information is not disclosed to unauthorized parties
- Kerckhoffs's Principle: A system should be secure even if everything but the key is public
- Attacks: Ciphertext-only, known plaintext, chosen plaintext attacks

Symmetric vs. Asymmetric Encryption

- Symmetric: Same key for encryption and decryption (DES, AES)
- Asymmetric: Public and private key pair (RSA)

Algorithms and Implementations

- RSA: $n = pq$, e, $d \equiv e^{-1} \mod \phi(n)$
- Modular inverse via Extended Euclidean Algorithm
- Primality testing: probabilistic (Miller-Rabin)

References

- Hoffstein, Pipher, Silverman: *An Introduction to Mathematical Cryptography*
- Simon Singh: *The Code Book*
- Simplilearn : A full Course on Cryptography