Lab 3: Implement One-Time Pad

Information Security and Cryptography Lab

Learning Goals

- Understand the concept of **perfect secrecy**.
- Implement the **One-Time Pad** encryption and decryption in C++.
- Appreciate the importance of key randomness and why key reuse breaks security.
- Gain practical skills in handling binary/ASCII/hex data for cryptographic operations.

Background

The One-Time Pad (OTP) is a symmetric cipher where:

1. A key K is chosen uniformly at random with the same length as the plaintext M.
2. Encryption: $C = M \oplus K$
3. Decryption: $M = C \oplus K$

Claude Shannon proved that OTP has **perfect secrecy** if:

- Keys are chosen uniformly at random.
- Each key is used *only once.*

Lab Tasks

Task 1: Implement OTP in C++

1. Write a program that:
   (a) Reads a plaintext message from a file `plaintext.txt`.
   (b) Generates a **truly random key** of the same length as the plaintext.
   (c) Saves the key to `key.bin`.
   (d) Encrypts the plaintext using XOR to produce `ciphertext.bin`.
   (e) Decrypts the ciphertext using the same key to produce `decrypted.txt`.
2. Ensure that the program works for both ASCII text and arbitrary binary data.

Task 2: Demonstrate Key Reuse Attack

1. Encrypt two different messages M_1 and M_2 with the *same key*.
2. Show that $C_1 \oplus C_2 = M_1 \oplus M_2$ can leak information.

Input/Output Specifications

- `plaintext.txt` — UTF-8 or ASCII text to encrypt.
- `key.bin` — Binary file containing the random key.
- `ciphertext.bin` — Encrypted data.
- `decrypted.txt` — Decrypted plaintext (must match the original).

Concept vs Implementation Note

Conceptually: For a message such as `"I want to meet with you at 6 pm today."`, the OTP process is:

1. Convert each character to its ASCII value.
2. Represent each ASCII value as an 8-bit binary string.
3. XOR each bit with a random bit from the key.

In practice (C++ implementation): You can skip explicit ASCII-to-binary conversion. Each character in a file is already stored as a byte (8 bits). You simply XOR each plaintext byte with the corresponding key byte. This is mathematically equivalent to the conceptual method but more efficient in code.

Starter C++ Code (Skeleton)

```cpp
#include <bits/stdc++.h>
using namespace std;

vector<unsigned char> readFile(const string &filename) {
    ifstream file(filename, ios::binary);
    return vector<unsigned char>((istreambuf_iterator<char>(file)
       ), {});
}

void writeFile(const string &filename, const vector<unsigned char
   > &data) {
    ofstream file(filename, ios::binary);
    file.write((char*)data.data(), data.size());
}

// TODO: Implement this function
```

```
vector<unsigned char> generateKey(size_t length) {
    vector<unsigned char> key(length);
    // Your code here: fill 'key' with random bytes
    return key;
}

// TODO: Implement this function
vector<unsigned char> xorData(const vector<unsigned char> &a,
                              const vector<unsigned char> &b) {
    vector<unsigned char> result(a.size());
    // Your code here: XOR each byte of 'a' with corresponding
       byte of 'b'
    return result;
}

int main() {
    // Encryption
    auto plaintext = readFile("plaintext.txt");
    auto key = generateKey(plaintext.size());
    writeFile("key.bin", key);
    auto ciphertext = xorData(plaintext, key);
    writeFile("ciphertext.bin", ciphertext);

    // Decryption
    auto readKey = readFile("key.bin");
    auto decrypted = xorData(ciphertext, readKey);
    writeFile("decrypted.txt", decrypted);
    return 0;
}
```

Testing

1. Encrypt and decrypt a sample text file. Verify that the decrypted output matches exactly.
2. Try encrypting a binary file (e.g., an image) and confirm correct round-trip decryption.

Key Reuse Attack Example

Let M_1 = "HELLO", M_2 = "WORLD", key K reused.

$$
\begin{aligned}
C_1 &= M_1 \oplus K \\
C_2 &= M_2 \oplus K \\
C_1 \oplus C_2 &= (M_1 \oplus K) \oplus (M_2 \oplus K) \\
&= M_1 \oplus M_2
\end{aligned}
$$

An attacker knowing part of M_1 can recover part of M_2.