## Name: **Ketan Kunkalikar**

## Roll: **21CSE1016**

## Lab Subject: **Security Lab**

## Topic: **Knapsack and Elgamal Algorithm.**

## Instructor: **Dr. Meenakshi Panda**

## Date: **30th October 2024**

## **Q1. What is the Knapsack Algorithm?**

## **Ans**: The Knapsack algorithm (specifically Merkle-Hellman) is a public-key cryptosystem based on the knapsack problem, which involves determining if a subset of numbers can sum to a given target value.

## **Main points:**

## 1. Key Type: Asymmetric encryption

## 2. Security Basis: Based on the subset sum problem

## 3. Parameters: Uses a superincreasing sequence and modular multiplier

## 4. Status: Considered broken for practical use

## **Key Components:**

## 1. System Parameters:

## - Superincreasing sequence (private)

## - Modulus m (private)

## - Multiplier w (private)

## - Public sequence derived from private parameters

## 2. Key Generation:

## - Create a superincreasing sequence

## - Choose modulus m larger than sum of sequence

## - Choose w coprime to m

## - Generate public key by multiplying each element by w mod m

## 3. Encryption/Decryption:

## - Encryption: Multiply message bits with public sequence

## - Decryption: Use modular inverse of w and solve easy knapsack

#include <iostream>

#include <vector>

using namespace std;

class KnapsackCrypto {

private:

    vector<int> privateKey;  // Super-increasing sequence

    int modulus;

    int multiplier;

    vector<int> publicKey;

    // Helper function to calculate modular multiplicative inverse

    int modInverse(int a, int m) {

        for (int x = 1; x < m; x++) {

            if (((a % m) \* (x % m)) % m == 1) {

                return x;

            }

        }

        return 1;

    }

public:

    KnapsackCrypto() {

        // Initialize with a simple super-increasing sequence

        privateKey = {2, 3, 7, 14, 30, 57, 120, 251};

        modulus = 491;  // Must be greater than sum of private key

        multiplier = 41; // Coprime with modulus

        cout << "Initializing Knapsack Cryptosystem\n";

        cout << "Private Key (super-increasing sequence): ";

        for (int x : privateKey) cout << x << " ";

        cout << "\nModulus: " << modulus;

        cout << "\nMultiplier: " << multiplier << "\n\n";

        // Generate public key

        publicKey.resize(privateKey.size());

        for (size\_t i = 0; i < privateKey.size(); i++) {

            publicKey[i] = (privateKey[i] \* multiplier) % modulus;

        }

        cout << "Generated Public Key: ";

        for (int x : publicKey) cout << x << " ";

        cout << "\n\n";

    }

    vector<int> encrypt(string message) {

        vector<int> encrypted;

        cout << "Encrypting message: " << message << "\n";

        for (char c : message) {

            int sum = 0;

            vector<int> binary = toBinary(c);

            cout << "Character '" << c << "' in binary: ";

            for (int bit : binary) cout << bit;

            cout << "\n";

            for (size\_t i = 0; i < binary.size(); i++) {

                if (binary[i] == 1) {

                    sum += publicKey[i];

                }

            }

            encrypted.push\_back(sum);

            cout << "Encrypted value: " << sum << "\n";

        }

        return encrypted;

    }

    string decrypt(vector<int> encrypted) {

        string decrypted;

        int inverse = modInverse(multiplier, modulus);

        cout << "\nDecrypting message using multiplicative inverse: " << inverse << "\n";

        for (int cipher : encrypted) {

            int sum = (cipher \* inverse) % modulus;

            vector<int> binary = solveSubsetSum(sum);

            char c = fromBinary(binary);

            decrypted += c;

            cout << "Cipher: " << cipher << ", Binary: ";

            for (int bit : binary) cout << bit;

            cout << " -> '" << c << "'\n";

        }

        return decrypted;

    }

private:

    vector<int> toBinary(char c) {

        vector<int> binary(8, 0);

        for (int i = 7; i >= 0; i--) {

            binary[7-i] = (c >> i) & 1;

        }

        return binary;

    }

    char fromBinary(vector<int> binary) {

        char c = 0;

        for (size\_t i = 0; i < binary.size(); i++) {

            if (binary[i]) {

                c |= (1 << (7-i));

            }

        }

        return c;

    }

    vector<int> solveSubsetSum(int sum) {

        vector<int> result(8, 0);

        for (int i = privateKey.size() - 1; i >= 0; i--) {

            if (sum >= privateKey[i]) {

                result[i] = 1;

                sum -= privateKey[i];

            }

        }

        return result;

    }

};

int main() {

    KnapsackCrypto crypto;

    string message = "helloworld";

    cout << "Original message: " << message << "\n\n";

    vector<int> encrypted = crypto.encrypt(message);

    cout << "\nFinal encrypted values: ";

    for (int x : encrypted) cout << x << " ";

    cout << "\n";

    string decrypted = crypto.decrypt(encrypted);

    cout << "\nDecrypted message: " << decrypted << "\n";

    return 0;

}

## **Output:**

## 

## **Q2. What is the Elgamal Algorithm?**

## **Ans:** ElGamal is a public-key cryptosystem based on the Diffie-Hellman key exchange, providing both encryption and digital signatures.

## **Main points:**

## 1. Key Type: Asymmetric encryption

## 2. Security Basis: Based on the discrete logarithm problem

## 3. Parameters: Uses prime modulus (p) and generator (g)

## 4. Features: Probabilistic encryption (same plaintext encrypts to different ciphertexts)

## **Key Components:**

## 1. System Parameters:

## - Prime modulus (p)

## - Generator (g)

## - Private key (x)

## - Public key (y = g^x mod p)

## 2. Key Generation:

## - Select large prime p and generator g

## - Choose random private key x

## - Compute public key y = g^x mod p

## 3. Encryption Process:

## - Choose random k

## - Compute c1 = g^k mod p

## - Compute c2 = m \* (y^k mod p) mod p

## - Ciphertext is (c1, c2)

## 4. Decryption Process:

## - Compute s = c1^x mod p

## - Compute m = c2 \* s^(p-2) mod p

## The security of ElGamal, like Diffie-Hellman, relies on the computational difficulty of the discrete logarithm problem.

#include <iostream>

#include <cmath>

#include <random>

using namespace std;

class ElGamal {

private:

    // System parameters

    long long p;  // Prime number

    long long g;  // Generator

    // Private key

    long long x;  // Random number < p

    // Public key

    long long y;  // y = g^x mod p

    long long power\_mod(long long base, long long exp, long long mod) {

        long long result = 1;

        base = base % mod;

        while (exp > 0) {

            if (exp & 1)

                result = (result \* base) % mod;

            base = (base \* base) % mod;

            exp >>= 1;

        }

        return result;

    }

public:

    ElGamal() {

        // Using small numbers for demonstration

        p = 23;  // Prime number

        g = 5;   // Generator

        x = 6;   // Private key

        y = power\_mod(g, x, p);  // Public key = g^x mod p

        cout << "ElGamal Parameters:\n";

        cout << "Prime (p): " << p << "\n";

        cout << "Generator (g): " << g << "\n";

        cout << "Public key (y): " << y << "\n\n";

    }

    pair<long long, long long> encrypt(long long message) {

        // Generate random k

        long long k = 4;  // Usually random, fixed for demonstration

        // Calculate c1 = g^k mod p

        long long c1 = power\_mod(g, k, p);

        // Calculate c2 = m \* y^k mod p

        long long c2 = (message \* power\_mod(y, k, p)) % p;

        cout << "Encrypting message: " << message << "\n";

        cout << "c1: " << c1 << ", c2: " << c2 << "\n";

        return make\_pair(c1, c2);

    }

    long long decrypt(pair<long long, long long> cipher) {

        long long c1 = cipher.first;

        long long c2 = cipher.second;

        // Calculate s = c1^x mod p

        long long s = power\_mod(c1, x, p);

        // Calculate s^(p-2) mod p (modular multiplicative inverse)

        long long s\_inv = power\_mod(s, p-2, p);

        // m = c2 \* s^(p-2) mod p

        long long m = (c2 \* s\_inv) % p;

        cout << "Decrypting (c1=" << c1 << ", c2=" << c2 << "): " << m << "\n";

        return m;

    }

};

int main() {

    ElGamal crypto;

    // Example message (must be less than p)

    long long message = 12;

    cout << "Original message: " << message << "\n\n";

    // Encrypt

    auto encrypted = crypto.encrypt(message);

    // Decrypt

    long long decrypted = crypto.decrypt(encrypted);

    cout << "\nDecrypted message: " << decrypted << "\n";

    return 0;

}

## **Output:**

## 