Київський національний університет імені Тараса Шевченка

Факультет комп’ютерних наук і кібернетики

Лабораторна робота №2

з курсу «Основи криптології»

на тему

**«Шифрування повідомлень»**

Виконали:

Студенти групи ІПС-31

***Андрєєв Ілля Євгенович***

***Безкровна Дар'я Олегівна***

***Вербицький Артем Віталійович***

***Дубина Андрій Володимирович***

***Павлюченко Василь Іванович***

Київ

2025

**Зміст**

Блочне шифрування з перестановкою …………………………………………. 3

Криптосистема Гольдвассера-Мікалі ………………………………………….. 5

**Блочне шифрування з перестановкою**

У цьому методі ми розбиваємо повідомлення на блоки фіксованого розміру і застосовуємо перестановку символів всередині кожного блоку. Давайте розглянемо приклад шифрування блочним методом з перестановкою.

**Хід шифрування**

**Повідомлення:** hello world i love cryptology

1. **Розбиття тексту на блоки:** Для простоти будемо використовувати блоки по 3 символи: Hel loW orl dIL ove Cry pTo log y
2. **Задання ключа перестановки:** Вибираємо ключ перестановки для кожного блоку. Припустимо, що це буде [2, 3, 1]

**Отримаємо:**

* **Блок 1**: "Hel" За ключем перестановки [2, 3, 1] отримаємо: "lhe".
* **Блок 2:** "loW" За ключем перестановки [2, 3, 1] отримаємо: "wlo".
* **Блок 3:** "orl" За ключем перестановки [2, 3, 1] отримаємо: "lor".
* **Блок 4:** "dIL" За ключем перестановки [2, 3, 1] отримаємо: "ldi".
* **Блок 5:** "ove" За ключем перестановки [2, 3, 1] отримаємо: "eov".
* **Блок 6:** "Cry" За ключем перестановки [2, 3, 1] отримаємо: "ycr".
* **Блок 7:** "pTo" За ключем перестановки [2, 3, 1] отримаємо: "opt".
* **Блок 8:** "log" За ключем перестановки [2, 3, 1] отримаємо: "glo".
* **Блок 9:** "y" (Останній блок): Цей блок складається лише з одного символу, тому він залишається без змін: "y".

**Результат:** lHeWlolorLdIeovyCroptglo y

**Реалізація на Python**

**def block\_permutation\_cipher(text, block\_size=3, permutation\_key=[2, 3, 1]):**

**# Прибираємо пробіли вручну**

**text\_no\_spaces = ""**

**for char in text:**

**if char != " ":**

**text\_no\_spaces += char**

**print(f"Текст без пробілів: {text\_no\_spaces}")**

**# Розбиваємо текст на блоки по block\_size**

**blocks = []**

**for i in range(0, len(text\_no\_spaces), block\_size):**

**block = text\_no\_spaces[i:i + block\_size]**

**blocks.append(block)**

**print(f"Блоки тексту: {blocks}")**

**# Результат шифрування**

**encrypted\_text = ""**

**# Застосовуємо перестановку для кожного блоку**

**for block\_index, block in enumerate(blocks):**

**print(f"Блок {block\_index + 1} перед перестановкою: {block}")**

**# Якщо довжина блоку менша, ніж block\_size, доповнюємо його пробілами**

**if len(block) < block\_size:**

**block = block + " " \* (block\_size - len(block))**

**print(f"Блок {block\_index + 1} після доповнення пробілами {block}")**

**# Застосовуємо перестановку по ключу permutation\_key**

**permuted\_block = [""] \* block\_size**

**for i in range(block\_size):**

**permuted\_block[permutation\_key[i] - 1] = block[i]**

**permuted\_block = ''.join(permuted\_block)**

**print(f"Блок {block\_index + 1} після перестановки: {permuted\_block}")**

**# Додаємо зашифрований блок у кінцевий результат**

**encrypted\_text += permuted\_block**

**return encrypted\_text**

**Криптосистема Гольдвассера-Мікалі**

Шифрування ґрунтується на неможливості (без знання факторизації модуля) визначити, чи є певне число квадратичним лишком (тобто має квадратний корінь) по модулю великого складного числа.

**Хід шифрування**

1. Кожен символ перетворюємо у ASCII-код, потім у двійковий формат (8 біт):

**"h"** → ASCII 104 → 01101000  
**"e"** → ASCII 101 → 01100101  
і так далі...

Весь текст перетвориться у бітовий рядок довжиною N = кількість символів × 8 = 27 × 8 = 216 бітів.

1. Для простоти візьмемо маленькі приклади (у реальності числа — 1024 біти і більше):

* p = 7, q = 11 → n = pq = 77
* x = 10

Ми перевіряємо, що x=10 не є квадратичним лишком ні по 7, ні по 11.

**Публічний ключ:** (n = 77, x = 10)  
**Секретний ключ:** (p = 7, q = 11)

1. Для кожного біта mi ∈ {0,1}:

* Випадково обираємо ri ∈ ​ (тобто r не має спільних дільників з n)
* Рахуємо:

A black background with white letters

AI-generated content may be incorrect.

**Наприклад:**

* Біт: 1, обрали r = 3
  + c = 10 ⋅ 32 mod  77 = 10 ⋅ 9 = 90 mod  77 = 13
* Біт: 0, обрали r = 4
  + c = 42 mod  77 = 16

Це робиться **для кожного біта**, в результаті отримаємо масив шифротекстів c1,c2,...,c216, що і буде нашим зашифрованим текстом.

**Реалізація на Python**

**# --- Helper Functions ---**

**def is\_quadratic\_residue(a, p):**

**return pow(a, (p - 1) // 2, p) == 1**

**def is\_residue\_mod\_n(a, p, q):**

**return is\_quadratic\_residue(a % p, p) and is\_quadratic\_residue(a % q, q)**

**def generate\_random\_r(n):**

**while True:**

**r = random.randint(2, n - 2)**

**if gcd(r, n) == 1:**

**return r**

**def encrypt\_bit(bit, public\_key):**

**n, x = public\_key**

**r = generate\_random\_r(n)**

**if bit == 0:**

**c = pow(r, 2, n)**

**else:**

**c = (pow(r, 2, n) \* x) % n**

**return c**

**def decrypt\_bit(ciphertext, private\_key):**

**p, q = private\_key**

**return 0 if is\_residue\_mod\_n(ciphertext, p, q) else 1**

**def to\_binary\_string(message):**

**return ''.join(format(ord(char), '08b') for char in message)**

**def from\_binary\_string(binary\_str):**

**chars = [chr(int(binary\_str[i:i+8], 2)) for i in range(0, len(binary\_str), 8)]**

**return ''.join(chars)**

**def is\_prime(n):**

**"""Simple primality test"""**

**if n <= 1:**

**return False**

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

**if n % i == 0:**

**return False**

**return True**

**def find\_non\_residue(p, q, n):**

**"""Find a suitable x (non-residue modulo n = p \* q)"""**

**x = 2 # Start checking from 2**

**while x < n:**

**if not is\_residue\_mod\_n(x, p, q):**

**return x**

**x += 1**

**return None # In case no valid x is found (unlikely in practical scenarios)**

**Реалізація на C++**

**bool isPrime(BigInt prime) {**

**for (BigInt i = 0;i < prime / 2; i++) {**

**cout << i;**

**if (prime % i != 0) {**

**return false;**

**}**

**}**

**return true;**

**}**

**BigInt binary\_string\_to\_bigint(const std::string& bin) {**

**BigInt result = 0;**

**for (char bit : bin) {**

**result = result \* 2 + (bit - '0');**

**}**

**return result;**

**}**

**namespace my\_gcd {**

**BigInt gcd(BigInt a, BigInt b) {**

**while (b != 0) {**

**BigInt t = b;**

**b = a % b;**

**a = t;**

**}**

**return a;**

**}**

**}**

**// symbol Jacobi**

**int jacobi(const BigInt& a\_input, const BigInt& n\_input) {**

**BigInt a = a\_input;**

**BigInt n = n\_input;**

**int result = 1;**

**// If a = 0, then Jacobi is always 0**

**if (a == 0) return 0;**

**// If a < 0, then we change the sign**

**if (a < 0) {**

**a = -a;**

**if (n % 4 == 3) result = -result;**

**}**

**while (a != 0) {**

**// Remove all factors of 2 from a**

**while ((a % 2) == 0) {**

**a /= 2;**

**// If n ≡ 3 (mod 8), change the sign**

**if (n % 8 == 3 || n % 8 == 5) result = -result;**

**}**

**swap(a, n);**

**// If a ≡ 3 (mod 4) and n ≡ 3 (mod 4), change the sign**

**if (a % 4 == 3 && n % 4 == 3) result = -result;**

**// a mod n**

**a %= n;**

**}**

**// If n = 1, then return the result, otherwise 0**

**return (n == 1) ? result : 0;**

**}**

**// Exponentiation by modulo**

**BigInt mod\_pow(BigInt base, BigInt exp, BigInt mod) {**

**BigInt result = 1;**

**base = base % mod;**

**while (exp > 0) {**

**if ((exp % 2) == 1)**

**result = (result \* base) % mod;**

**exp = exp / 2;**

**base = (base \* base) % mod;**

**}**

**return result;**

**}**

**BigInt generate\_prime(int bits) {**

**random\_device rd;**

**mt19937 gen(rd());**

**uniform\_int\_distribution<int> dist(0, 1);**

**while (true) {**

**string num\_str = "1"; // most significant bit = 1**

**for (int i = 0; i < bits - 2; ++i) {**

**num\_str += to\_string(dist(gen));**

**}**

**num\_str += "1"; // least significant bit = 1 (odd)**

**BigInt num = binary\_string\_to\_bigint(num\_str);**

**bool is\_prime = true;**

**for (int i = 0; i < 5; ++i) {**

**int a\_int = 2 + rand() % 10; // small values for testing**

**BigInt a = a\_int;**

**if (mod\_pow(a, num - 1, num) != 1) {**

**is\_prime = false;**

**break;**

**}**

**}**

**if (is\_prime && (num % 4 == 3)) return num;**

**}**

**}**

**// Keys struct**

**struct GMKeys {**

**BigInt n;**

**BigInt y;**

**BigInt p, q;**

**};**

**int bit\_length(const BigInt& num) {**

**BigInt n = num;**

**int bits = 0;**

**while (n > 0) {**

**n /= 2;**

**++bits;**

**}**

**return bits;**

**}**

**BigInt random\_bigint\_less\_than(const BigInt& n) {**

**random\_device rd;**

**mt19937 gen(rd());**

**uniform\_int\_distribution<int> dist(0, 1);**

**int bits = bit\_length(n);**

**BigInt result;**

**do {**

**string bin = "1"; //Start from 1 to avoid zero**

**for (int i = 1; i < bits; ++i) {**

**bin += to\_string(dist(gen));**

**}**

**result = binary\_string\_to\_bigint(bin);**

**} while (result >= n); // repeat until we get into the range**

**return result;**

**}**

**GMKeys generate\_keys(int bits = 64) {**

**BigInt p = generate\_prime(bits / 2);**

**BigInt q = generate\_prime(bits / 2);**

**BigInt n = p \* q;**

**BigInt y;**

**do {**

**y = random\_bigint\_less\_than(n);**

**cout << jacobi(y, q) << " " << jacobi(y, p) << " " << jacobi(y, n) << endl; // debug y**

**} while (jacobi(y, q) != -1 || jacobi(y, p) != -1);**

**return { n, y, p, q };**

**}**

**// encrypting**

**vector<BigInt> encrypt(const string& msg, const GMKeys& key) {**

**vector<BigInt> ciphertext;**

**for (char c : msg) {**

**bitset<8> bits(c);**

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

**BigInt r;**

**do {**

**r = rand() % key.n;**

**} while (my\_gcd::gcd(r, key.n) != 1);**

**BigInt ciph = mod\_pow(r, 2, key.n);**

**if (bits[i]) {**

**ciph = (ciph \* key.y) % key.n;**

**}**

**ciphertext.push\_back(ciph);**

**}**

**}**

**return ciphertext;**

**}**

**string decrypt(const vector<BigInt>& ciphertext, const BigInt& p) {**

**string result;**

**for (size\_t i = 0; i < ciphertext.size(); i += 8) {**

**char c = 0;**

**for (int j = 0; j < 8; ++j) {**

**int bit = jacobi(ciphertext[i + j], p);**

**cout << "Jacobi(" << ciphertext[i + j] << ", " << p << ") = " << bit ;**

**cout << " Bit " << j << ": " << ((bit == 1) ? 0 : 1) << endl; // debug output**

**c = (c << 1) | ((bit == 1) ? 0 : 1);**

**}**

**result += c;**

**cout << "Decrypted char: " << c << " (" << bitset<8>(c) << ")\n";**

**}**

**return result;**

**}**