Лабораторна робота № 2.
Реалізація алгоритмів генерації ключів гібридних криптосистем.

Виконали: *Бараніченко Андрій, Гаврилова Анастасія, Дрозд Софія, Зібаров Дмитро, Колесник Андрій*

Мета роботи: Дослідження алгоритмів генерації псевдовипадкових послідовностей, тестування простоти чисел і генерації простих чисел із точки зору їхньої ефективності за часом і можливості використання для гененерації ключів асиметричних криптосистем.

Завдання:

Підгрупа 2А. Бібліотека OpenSSL під Windows платформу.

Код

```
#include <openssl/rand.h>
#include <openssl/evp.h>
#include <iostream>
#include <vector>
#include <chrono>
#include <random>
#include <algorithm>
#include <set>
#include <cmath>
#include <numeric>
#include <map>
#include <iomanip>

using namespace std;

double monteCarloPiTest(const vector<unsigned char>& data) {
    int insideCircle = 0;
    int totalPoints = data.size() / 2; // Кожна точка представлена двома байтами
    for (int i = 0; i < totalPoints; ++i) {
        double x = (double)data[2 * i] / 255;
        double y = (double)data[2 * i + 1] / 255;
        if (x * x + y * y <= 1.0) {
            ++insideCircle;
        }
    }
    double piEstimate = (double)insideCircle / totalPoints * 4;
    return piEstimate;
}

double sampleMeanTest(const vector<unsigned char>& data) {
    double sum = accumulate(data.begin(), data.end(), 0.0);
    double mean = sum / data.size();
    return mean;
}


int runsTest(const vector<unsigned char>& data) {
    int runs = 1; // Почнемо з першого елемента
```

```
    for (size_t i = 1; i < data.size(); ++i) {
        if ((data[i] > 127 && data[i - 1] <= 127) || (data[i] <= 127 && data[i - 1] >
127)) {
            ++runs;
        }
    }
    return runs;
}

// Функція для генерації псевдовипадкових чисел з використанням OpenSSL
vector<unsigned char> generateRandomBytesOpenSSL(int numBytes) {
    vector<unsigned char> buffer(numBytes);
    if (RAND_bytes(buffer.data(), numBytes) != 1) {
        throw runtime_error("RAND nums generation error");
    }
    return buffer;
}

// Функція для генерації ключа з використанням OpenSSL
vector<unsigned char> generateKeyOpenSSL(int keyLength) {
    vector<unsigned char> key(keyLength);
    if (RAND_bytes(key.data(), keyLength) != 1) {
        throw runtime_error("Key generation error");
    }
    return key;
}

// Функція для генерації псевдовипадкових чисел з використанням стандартної бібліотеки
vector<unsigned char> generateRandomBytesStd(int numBytes) {
    vector<unsigned char> buffer(numBytes);
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dis(0, 255);
    generate(buffer.begin(), buffer.end(), [&]() { return dis(gen); });
    return buffer;
}

// Функція для обчислення ентропії Шеннона
double calculateShannonEntropy(const vector<unsigned char>& data) {
    map<unsigned char, int> freq;
    for (auto byte : data) {
        freq[byte]++;
    }
    double entropy = 0.0;
    for (const auto& pair : freq) {
        double p = (double)pair.second / data.size();
        entropy -= p * log2(p);
    }
    return entropy;
}

// Тест хі-квадрат
double chiSquareTest(const vector<unsigned char>& data) {
    map<unsigned char, int> freq;
    for (auto byte : data) {
        freq[byte]++;
    }
    double chiSquare = 0.0;
    double expected = (double)data.size() / 256;
    for (const auto& pair : freq) {
```

```
        chiSquare += pow(pair.second - expected, 2) / expected;
    }
    return chiSquare;
}

int main() {
    try {
        int numBytes = 4096; // Кількість байтів для генерації
        int keyLength = 32; // Довжина ключа в байтах

        // Генерація псевдовипадкових чисел з OpenSSL
        auto start = chrono::high_resolution_clock::now();
        vector<unsigned char> randomBytesOpenSSL = generateRandomBytesOpenSSL(numBytes);
        auto end = chrono::high_resolution_clock::now();
        chrono::duration<double> duration = end - start;
        cout << "Time to generate pseudorand numbers (OpenSSL): " << duration.count() << "
seconds\n";
        cout << "Pseudorand numbers (OpenSSL): ";
        for (unsigned char byte : randomBytesOpenSSL) {
            cout << hex << static_cast<int>(byte) << " ";
        }
        cout << endl;

        // Генерація ключа з OpenSSL
        start = chrono::high_resolution_clock::now();
        vector<unsigned char> keyOpenSSL = generateKeyOpenSSL(keyLength);
        end = chrono::high_resolution_clock::now();
        duration = end - start;
        cout << "Time to generate key (OpenSSL): " << duration.count() << " seconds\n";
        cout << "Generated key (OpenSSL): ";
        for (unsigned char byte : keyOpenSSL) {
            cout << hex << static_cast<int>(byte) << " ";
        }
        cout << endl;

        // Генерація псевдовипадкових чисел з використанням стандартної бібліотеки
        start = chrono::high_resolution_clock::now();
        vector<unsigned char> randomBytesStd = generateRandomBytesStd(numBytes);
        end = chrono::high_resolution_clock::now();
        duration = end - start;
        cout << "Time to generate pseudorand numbers (std): " << duration.count() << "
seconds\n";
        cout << "Pseudorand numbers (std): ";
        for (unsigned char byte : randomBytesStd) {
            cout << hex << static_cast<int>(byte) << " ";
        }
        cout << endl;

        // Перевірка випадковості результатів
        cout << "Checking randomness...\n\n";
        auto countUniqueBytes = [](const vector<unsigned char>& data) {
            set<unsigned char> uniqueBytes(data.begin(), data.end());
            return uniqueBytes.size();
            };
        size_t uniqueOpenSSL = countUniqueBytes(randomBytesOpenSSL);
        size_t uniqueStd = countUniqueBytes(randomBytesStd);
        cout << "Unique bytes (OpenSSL): " << uniqueOpenSSL << "\n";
        cout << "Unique bytes (std): " << uniqueStd << "\n\n";

        // Ентропія Шеннона
```

```
        double entropyOpenSSL = calculateShannonEntropy(randomBytesOpenSSL);
        double entropyStd = calculateShannonEntropy(randomBytesStd);
        cout << "Shannon Entropy (OpenSSL): " << entropyOpenSSL << "\n";
        cout << "Shannon Entropy (std): " << entropyStd << "\n\n";

        // Тест хі-квадрат
        double chiSquareOpenSSL = chiSquareTest(randomBytesOpenSSL);
        double chiSquareStd = chiSquareTest(randomBytesStd);
        cout << "Chi-Square Test (OpenSSL): " << chiSquareOpenSSL << "\n";
        cout << "Chi-Square Test (std): " << chiSquareStd << "\n\n";

        double piEstimateOpenSSL = monteCarloPiTest(randomBytesOpenSSL);
        double piEstimateStd = monteCarloPiTest(randomBytesStd);
        cout << "Monte Carlo Pi Estimate (OpenSSL): " << piEstimateOpenSSL << "\n";
        cout << "Monte Carlo Pi Estimate (std): " << piEstimateStd << "\n\n";

        double sampleMeanOpenSSL = sampleMeanTest(randomBytesOpenSSL);
        double sampleMeanStd = sampleMeanTest(randomBytesStd);
        cout << "Sample Mean (OpenSSL): " << sampleMeanOpenSSL << "\n";
        cout << "Sample Mean (std): " << sampleMeanStd << "\n\n";

        //int runsOpenSSL = runsTest(randomBytesOpenSSL);
        //int runsStd = runsTest(randomBytesStd);
        //cout << "Runs Test (OpenSSL): " << runsOpenSSL << "\n";
        //cout << "Runs Test (std): " << runsStd << "\n";
    }
    catch (const exception& e) {
        cerr << "Error occurred: " << e.what() << endl;
    }
    return 0;
}
```

Результат роботи

```
Time to generate pseudorand numbers (OpenSSL): 0.0056852 seconds
Pseudorand numbers (OpenSSL): 2e 72 f1 a5 c1 55 63 8d 1 38 95 5c f0 81 4c b4 f 80
94 2d 57 3a 98 28 bc 77 fb 55 22 8b 10 13 25 d4 95 6b 5b 4a 38 a7 7a 20 5f f4 ad
40 71 c6 fd 50 34 c1 b7 64 63 9f ee 1c da ca fa 8b ea f f7 34 5a 8 70 71 e ac 9 3b
53 80 5c d3 93 63 5d 97 8f cd 54 36 85 69 69 64 17 aa ed 6a 6f 4e 51 e6 7f fc 5d
f9 49 c3 f5 82 8c 98 6d 49 4 23 98 f8 a2 97 89 42 6a c1 46 5b 52 2d 2e 13 df b6
Time to generate key (OpenSSL): 3.45e-05 seconds
Generated key (OpenSSL): 58 7a 7a f8 29 bf a9 98 47 4d ea 3a 12 b1 14 16 12 4e 68
77 5b c1 2a 22 aa 2 c 3d d dd 38 e8
Time to generate pseudorand numbers (std): 9.73e-05 seconds
Pseudorand numbers (std): 9f 7c ab 4f 78 92 67 9b 62 57 b0 a7 82 91 41 f0 80 96 5a
2e 34 f5 db cc f1 33 1 4a e2 a0 5e d5 5e c7 e3 ff 63 67 53 a6 95 16 c a4 be 4 21
82 d1 9a b3 9d 33 4d 69 98 a4 9e c0 ef 6c c1 1 66 15 76 5e 43 fa 60 2 ca b5 db 7
b1 b6 be 9f 45 a5 f2 f6 a2 d de b4 e2 d0 75 f3 fd 4d 8 c6 bf 4f 2b 74 6 e3 45 2f
a0 3c 3e a7 1c bd b8 ec 81 3e e1 1b bb e d0 86 38 b4 91 1f 84 81 a8 c6 8a
Checking randomness...

Unique bytes (OpenSSL): 67
Unique bytes (std): 69

Shannon Entropy (OpenSSL): 6.59168
```

```
Shannon Entropy (std): 6.63473

Chi-Square Test (OpenSSL): 163.5
Chi-Square Test (std): 148.5

Monte Carlo Pi Estimate (OpenSSL): 3.1875
Monte Carlo Pi Estimate (std): 3.0625

Sample Mean (OpenSSL): 121.898
Sample Mean (std): 133.992
```

Опис функції генерації ПСП та ключів, бібліотеки з описом алгоритму, вхідних і вихідних даних, кодів повернення:

Функція	**generateRandomBytes**	**generateKey**
Опис	призначена для генерації псевдовипадкових чисел заданої кількості байтів	призначена для генерації криптографічного ключа заданої довжини
Алгоритм	Створюється буфер довжиною numBytes. Функція RAND_bytes із бібліотеки OpenSSL використовується для заповнення буфера псевдовипадковими числами. Якщо RAND_bytes повертає 1, буфер успішно заповнено псевдовипадковими числами. В іншому випадку, викликається виключення.	Створюється вектор довжиною keyLength. Функція RAND_bytes із бібліотеки OpenSSL використовується для заповнення вектора випадковими байтами. Якщо RAND_bytes повертає 1, вектор успішно заповнено випадковими байтами. В іншому випадку, викликається виключення.
Вхідні дані	numBytes: Кількість байтів для генерації.	keyLength: Довжина ключа в байтах
Вихідні дані	Вектор std::vector<unsigned char>, що містить псевдовипадкові числа.	Вектор std::vector<unsigned char>, що містить криптографічний ключ
Коди повернення	Повертає вектор із псевдовипадковими числами. Викликає виключення std::runtime_error у разі помилки.	Повертає вектор із криптографічним ключем. Викликає виключення std::runtime_error у разі помилки.

Контрольні приклади роботи з функціями (використання обох функцій для генерації 16 псевдовипадкових чисел і 32-байтового криптографічного ключа, у разі виникнення помилки буде виведено повідомлення з описом помилки):

generateRandomBytes	generateKey

std::vector<unsigned char> generateRandomBytes(int numBytes) {
    std::vector<unsigned char> buffer(numBytes);
    if (RAND_bytes(buffer.data(), numBytes) != 1) {
        throw std::runtime_error("RAND nums generation error");
    }
    return buffer;
}
	int main() {
	    try {
	        int numBytes = 16; // Кількість байтів для генерації
	        int keyLength = 32; // Довжина ключа в байтах
	
	        std::vector<unsigned char> randomBytes = generateRandomBytes(numBytes);
	        std::cout << "Pseudorand numbers: ";
	        for (unsigned char byte : randomBytes) {
	            std::cout << std::hex << static_cast<int>(byte) << " ";
	        }
	        std::cout << std::endl;
	
	        std::vector<unsigned char> key = generateKey(keyLength);
	        std::cout << "Generated key: ";
	        for (unsigned char byte : key) {
	            std::cout << std::hex << static_cast<int>(byte) << " ";
	        }
	        std::cout << std::endl;
	    } catch (const std::exception& e) {
	        std::cerr << "Error occured: " << e.what() << std::endl;
	    }
	    return 0;
	}

Порівняння

Час генерації:

OpenSSL повільніше генерує псевдовипадкові числа (0.0056852 секунд) порівняно зі стандартною бібліотекою (0.0000973 секунд). OpenSSL оптимізований для

криптографічних функцій, проте в нього значно більше функцій, тому він дещо повільніший.

Псевдовипадкові числа:

І OpenSSL, і стандартна бібліотека генерували випадкові числа, але їх розподіли трохи відрізняються.

Унікальні байти:

OpenSSL: 67 унікальних байтів
Стандартна бібліотека: 69 унікальних байтів

Обидва методи показують високу різноманітність байтів, що є добрим показником випадковості.

Ентропія Шеннона:

OpenSSL: 6.59168
Стандартна бібліотека: 6.63473

Ентропія Шеннона є мірою випадковості. Значення, близькі до 8, є ідеальними для 8-бітового набору даних. Обидва значення високі, що свідчить про добру випадковість.

Тест хі-квадрат:

OpenSSL: 163.5
Стандартна бібліотека: 148.5

Тест хі-квадрат вимірює, наскільки очікувані та фактичні частоти байтів співпадають. Нижче значення зазвичай означає краще співпадіння, але обидва значення досить високі, що свідчить про відхилення від очікуваної однорідності.

Оцінка числа Пі за методом Монте-Карло:

OpenSSL: 3.1875
Стандартна бібліотека: 3.0625

Обидві оцінки близькі до справжнього значення числа Пі (3.14159...), причому оцінка OpenSSL трохи точніша.

Середнє значення:

OpenSSL: 121.898
Стандартна бібліотека: 133.992

Середнє значення випадкових байтів ідеально становить приблизно 127.5 для 8-бітового діапазону (0-255). Середнє значення OpenSSL ближче до цього ідеального значення.

Висновки

У ході виконання даної лабораторної роботи ми навчилися, використовуючи мову програмування C++ та бібліотеки OpenSSL, генерувати псевдовипадкові числа заданої кількості байтів за допомогою функції generateRandomBytes і генерувати криптографічні ключі заданої довжини за допомогою функції generateKey, також за допомогою функцій із бібліотеки chrono ми виконали аналіз часу, за котрий генеруються псевдовипадкові числа та криптографічні ключі в нашій програмі.

Враховуючи порівняння можна зробити висновок, що випадкова генерація в OpenSSL не настільки краща за стандартні бібліотеки, як каже реклама, проте показники все ще дуже достойні.