Учреждение образования

«Белорусский государственный технологический университет»

**Отчет по лабораторной работе №6**

по дисциплине: «Защита информации и надежность информационных систем»

Исполнитель:

Воликов Д. А., ФИТ 4-7

Руководитель:

Асс. Савельева М. Г.

Минск 2024

# Лабораторная работа №6. Исследование потоковых шифров

**Цель**: изучение и приобретение практических навыков разработки и использования приложений для реализации потоковых шифров.

**Задачи**:

1. Закрепить теоретические знания по алгебраическому описанию, алгоритмам реализации операций зашифрования/расшифрования и оценке криптостойкости потоковых шифров.
2. Разработать приложение для реализации указанных преподавателем методов генерации ключевой информации и ее использования для потокового зашифрования/расшифрования.
3. Выполнить анализ криптостойкости потоковых шифров.
4. Оценить скорость зашифрования/расшифрования реализованных шифров.
5. Результаты выполнения лабораторной работы оформить в виде описания разработанного приложения, методики выполнения экспериментов с использованием приложения и результатов эксперимента.

## Теоретические сведения

Потоковый шифр (иногда говорят «поточный») – симметричный шифр, преобразующий каждый символ *m*i открытого текста в символ шифрованного *c*i, зависящий от ключа и расположения символа в тексте.

Термин «потоковый шифр» обычно используется в том случае, когда шифруемые символы открытого текста представляются одной буквой, битом или реже – байтом.

Все потоковые шифры делятся на 2 класса: синхронные и асинхронные (или самосинхронизирующиеся).

Основной задачей потоковых шифров является выработка некоторой последовательности (гаммы) для зашифрования, т. е. выходная гамма является ключевым потоком (ключом) для сообщения. В общем виде схема потокового шифра изображена на рис. 1.

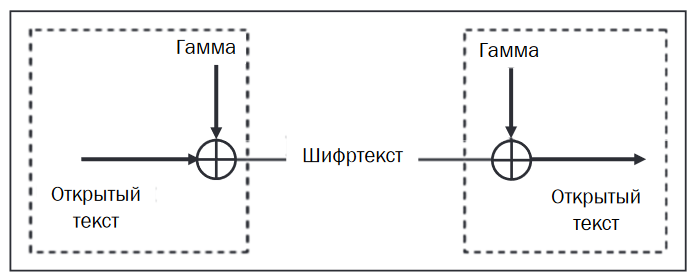


Рисунок 1 – Схема потокового шифра

Синхронные потоковые шифры (СПШ) характеризуются тем, что поток ключей генерируется независимо от открытого текста и шифртекста. Главное свойство СПШ – нераспространение ошибок. Ошибки отсутствуют, пока работают синхронно шифровальное и дешифровальное устройства отправителя и получателя информации. Один из методов борьбы с рассинхронизацией – разбить отрытый текст на отрезки, начало и конец которых выделить вставкой контрольных меток (специальных маркеров).

Синхронные потоковые шифры уязвимы к атакам на основе изменения отдельных битов шифртекста.

В самосинхронизирующихся потоковых шифрах символы ключевой гаммы зависят от исходного секретного ключа шифра и от конечного числа последних знаков зашифрованного текста. Основная идея заключается в том, что внутреннее состояние генератора потока ключей является функцией фиксированного числа предыдущих битов шифртекста. Поэтому генератор потока ключей на приемной стороне, приняв фиксированное число битов, автоматически синхронизируется с генератором гаммы.

Недостаток этих потоковых шифров – распространение ошибок, так как искажение одного бита в процессе передачи шифртекста приведет к искажению нескольких битов гаммы и, соответственно, расшифрованного сообщения.

Генератор псевдослучайных чисел на основе алгоритма RSA. Алгоритм RSA разработан для систем асимметричного зашифрования/расшифрования и будет более детально рассмотрен с практической точки зрения ниже.

Генератор же ПСП на основе RSA устроен следующим образом. Последовательность генерируется с использованием соотношения

*x*t = (*x*t - 1)e mod *n*

Начальными параметрами служат n, большие простые числа *p* и *q* (причем *n* = *p* ⋅ *q*), целое число *е*, взаимно простое с произведением (*р* – 1) ⋅ (*q* – 1), а также некоторое случайное начальное значение *x*0.

Потоковый шифр RC4. Алгоритм RC4 разработан Р. Ривестом в 1987 г. Он представляет собой потоковый шифр с переменным размером ключа. Здесь гамма не зависит от открытого текста.

Алгоритм RC4, как и любой потоковый шифр, строится на основе генератора псевдослучайных битов (генератора ПСП). На вход генератора записывается ключ, а на выходе читаются псевдослучайные биты. Длина ключа может составлять от 40 до 2048 битов.

Ядро алгоритма состоит из функции генерации ключевого потока. Другая часть алгоритма – функция инициализации, которая использует ключ переменной длины *K*i для создания начального состояния генератора ключевого потока.

В основе алгоритма – размер блока или слова, определяемый параметром *n*. Обычно *n* = 8, но можно использовать и другие значения. Внутренне состояние шифра определяется массивом слов (*S*-блоком) размером 2 ⋅ *n*. При *n* = 8 элементы блока представляют собой перестановку чисел от 0 до 255, а сама перестановка зависит от ключа переменной длины. Другими элементами внутреннего состояния являются 2 счетчика (каждый размером в одно слово; обозначим их *i* и *j*) с нулевыми начальными значениями. В основе вычислений лежит операция по mod (2 ⋅ *n*).

Генератор ключевого потока RC4 переставляет значения, хранящиеся в *S*, и каждый раз выбирает различное значение из *S* в качестве результата. В одном цикле RC4 определяется одно *n*-битное слово *K* из ключевого потока, которое в последующем суммируется с исходным текстом для получения зашифрованного текста. Эта часть алгоритма называется генератором ПСП. При *n* = 8 для генерации случайного байта выполняются операции, представленные на рисунке 2.

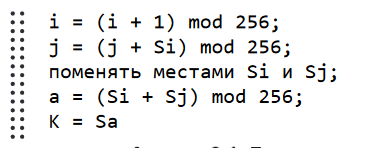


Рисунок 2 – Псевдокод для генерации байта ПСП

Байт *K* используется в операции XOR с открытым текстом для получения 8-битного шифртекста или для его расшифрования.

Так же достаточно проста и инициализация *S*-блока. Этот алгоритм использует ключ, который подается на вход пользователем. Сначала *S*-блок заполняется линейно: *S*0 = 0, *S*1 = 1, …, *S*255 = 255. Затем заполняется секретным ключом другой 256-байтный массив. Если необходимо, ключ повторяется многократно, чтобы заполнить весь массив: *K*0, *K*1, …, *K*255. Далее массив *S* перемешивается путем перестановок, определяемых ключом. Действия выполняются в соответствии с псевдокодом, представленным на рисунке 3.

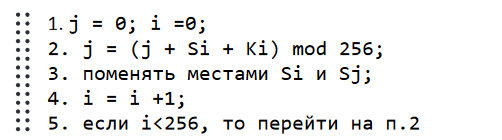


Рисунок 3 – Псевдокод для начального заполнения таблицы замен *S* (*S*-блока)

## 1.2 Практическое задание

**Вариант 2**. Разработать авторские приложения в соответствии с целью лабораторной работы. При этом можно воспользоваться готовыми библиотеками либо программными кодами, реализующими заданные алгоритмы.

Приложение 1 должно реализовывать генерацию ПСП на основе алгоритма RSA. Значения *p*, *q* и *e* должны иметь размер 256 бит.

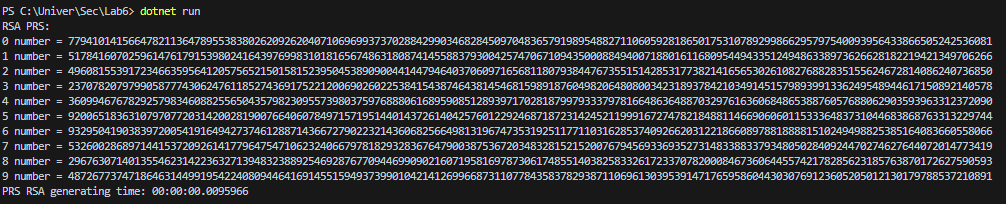


Рисунок 4 – Результат генерации ПСП на основе RSA

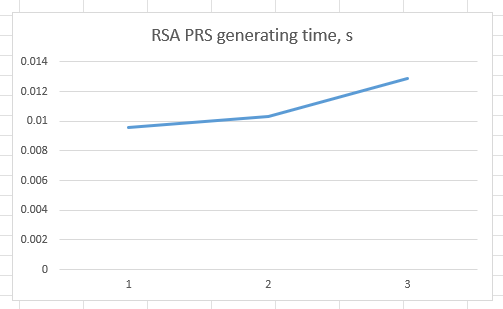


Рисунок 5 – График времени генерации ПСП на основе RSA

Приложение 2 должно реализовывать алгоритм RC4 (параметр *n* = 8, ключевая последовательность = «76, 111, 85, 54, 211»), а также дополнительно выполнять оценку скорости выполнения операций генерации ПСП.

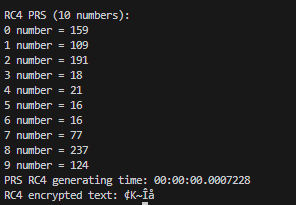


Рисунок 6 – Результат генерации ПСП и шифрования на основе RС4

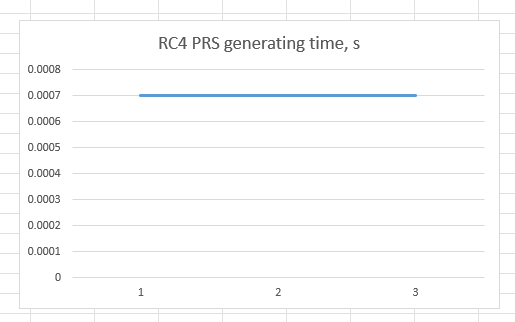


Рисунок 7 – График времени генерации ПСП на основе RC4

Вывод: в ходе этой лабораторной работы я изучил принцип работы потоковых шифров. Написал программу, которая рассчитывает ПСП на основе двух алгоритмов: RSA и RC4. Программы выводит рассчитанные значения, а также время генерации в секундах.

# Листинг «Лабораторная работа №5»

using Lab6.Generators;

var rsaGenerator = new RSAGenerator();

var rc4Generator = new RC4Generator(8);

Console.WriteLine("RSA PRS:");

var watch = System.Diagnostics.Stopwatch.StartNew();

var rsaSequence = rsaGenerator.Generate(10);

watch.Stop();

for (int i = 0; i < rsaSequence.Length; i++) {

Console.WriteLine($"{i} number = {rsaSequence[i]}");

}

Console.WriteLine($"PRS RSA generating time: {watch}\n");

Console.WriteLine("RC4 PRS (10 numbers):");

watch = System.Diagnostics.Stopwatch.StartNew();

var rc4Sequence = rc4Generator.GeneratePRS([76, 111, 85, 54, 211]);

watch.Stop();

for (int i = 0; i < 10; i++) {

Console.WriteLine($"{i} number = {rc4Sequence[i]}");

}

Console.WriteLine($"PRS RC4 generating time: {watch}");

var rc4EncryptedText = rc4Generator.Encrypt("Hello", [76, 111, 85, 54, 211]);

Console.WriteLine($"RC4 encrypted text: {rc4EncryptedText}");

Листинг 6.1 – Код программы

using System.Numerics;

namespace Lab6.Generators

{

public class RSAGenerator

{

private readonly BigInteger P;

private readonly BigInteger Q;

private readonly BigInteger N;

private readonly BigInteger E;

public RSAGenerator()

{

P = BigInteger.Parse("92575699611712814098324868827691115100832465461551981488684012097885956824763"); // Я не хочу писать генератор простых чисел

Q = BigInteger.Parse("112067608275005981840867909737467414898838159295533174762192122642924053837623");

E = BigInteger.Parse("102575699611712814098324868827691115100832465461551981488684012097885956824761");

N = P \* Q;

}

public BigInteger[] Generate(int length)

{

BigInteger x0 = new(new Random().Next());

BigInteger[] sequence = new BigInteger[length];

for (int i = 0; i < length; i++) {

sequence[i] = BigInteger.ModPow(i == 0 ? x0 : sequence[i - 1], E, N);

}

return sequence;

}

}

}

Листинг 6.2 – Код генератора RSA

namespace Lab6.Generators

{

public class RC4Generator(int n)

{

public int N { get; private set; } = n;

public string Encrypt(string message, int[] key)

{

var (sTable, kTable) = GenerateTables(key);

var result = "";

int i = 0;

int j = 0;

for (int k = 0; k < message.Length; k++)

{

i = (i + 1) % (int)Math.Pow(2, N);

j = (j + sTable[i]) % (int)Math.Pow(2, N);

(sTable[j], sTable[i]) = (sTable[i], sTable[j]);

result += (char)(message[k] ^ sTable[(sTable[i] + sTable[j]) % (int)Math.Pow(2, N)]);

}

return result;

}

public int[] GeneratePRS(int[] key)

{

var (sTable, kTable) = GenerateTables(key);

int i = 0;

int j = 0;

for (int k = 0; k < (int)Math.Pow(2, N); k++) {

j = (j + sTable[i]) % (int)Math.Pow(2, N);

(sTable[j], sTable[i]) = (sTable[i], sTable[j]);

int a = (sTable[i] + sTable[j]) % (int)Math.Pow(2, N);

kTable[k] = sTable[a];

i++;

}

return kTable;

}

private (int[], int[]) GenerateTables(int[] key)

{

var sTable = new int[(int)Math.Pow(2, N)];

var kTable = new int[(int)Math.Pow(2, N)];

for (int k = 0; k < (int)Math.Pow(2, N); k++) {

sTable[k] = k;

kTable[k] = key[k % key.Length];

}

int i = 0;

int j = 0;

while (i < (int)Math.Pow(2, N)) {

j = (j + sTable[i] + kTable[i]) % (int)Math.Pow(2, N);

(sTable[j], sTable[i]) = (sTable[i], sTable[j]);

i++;

}

return (sTable, kTable);}}}

Листинг 6.3 – Код генератора RC4