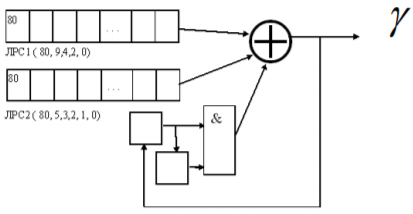
|  |
| --- |
| Университет итмо |
| Лабораторная работа №2 по дисциплине «Информационная безопасность» |
| «Поточное симметричное шифрование» |
| Вариант 21  Группа Р3402 |
| **Выполнили:**  **Лалетина Екатерина Александровна**  **Орлова Кристина Александровна** |
| **Преподаватель: Маркина Татьяна Анатольевна** |

|  |
| --- |
| *05.04.20* |

**Цель работы:** изучение структуры и основных принципов работы современных алгоритмов поточного симметричного шифрования, приобретение навыков программной реализации поточных симметричных шифров.

**Задание**

Реализовать в программе поточное кодирование текста, вводимого с клавиатуры, с помощью заданной нелинейной схемы, использующей дополнительные ячейки памяти РС.



**Листинг разработанной программы**

**Класс Main.java:**

**import** Structure.Register;  
  
**import** java.io.PrintStream;  
**import** java.util.Scanner;  
  
**public class** Main {  
 **public static void** main(String[] args) **throws** java.io.UnsupportedEncodingException {  
 Scanner in = **new** Scanner(System.***in***, **"UTF-8"**);  
 System.***out***.println(**"Source: "**);  
  
 String source = in.nextLine();  
 PrintStream ps = **new** PrintStream(System.***out***, **true**, **"UTF-8"**);  
 String plainText = *convertToBinary*(**new** StringBuilder(source));  
 String key = *convertToBinary*(**new** StringBuilder(**"keyword"**));  
 String cipherText = *encode*(plainText, key);  
  
 plainText = *encode*(*convertToBinary*(**new** StringBuilder(cipherText)), key);  
  
 ps.println(**"Encoded: "** + *replaceControlCharacters*(cipherText));  
 ps.println(**"Decoded: "** + plainText);  
 }  
  
 */\*\*  
 \* Шифрование  
 \* Работает в обе стороны: шифрование / расшифрование  
 \*  
 \** ***@param plainText*** *- открытый текст, который необходимо зашифровать  
 \** ***@param key*** *- ключ  
 \** ***@return*** *- возвращает шифртекст  
 \*/* **private static** String encode(String plainText, String key) {  
 **final int** registerLength = 81;  
 **final int**[] register1BitPositions = {80, 9, 4, 2, 0};  
 **final int**[] register2BitPositions = {80, 5, 3, 2, 1, 0};  
 Register register1 = **new** Register(registerLength, register1BitPositions, key);  
 Register register2 = **new** Register(registerLength, register2BitPositions, key);  
 StringBuilder cipherText = **new** StringBuilder();  
 **int** lastGamma = 0;  
  
 **for** (**int** i = 0; i < registerLength; i++) { *// Прогоняем столько тактов, сколько длина регистра, вхолостую* lastGamma = *generateGamma*(register1, register2, lastGamma);  
 }  
  
 **for** (**int** i = 0; i < plainText.length(); i++) {  
 **int** gamma = *generateGamma*(register1, register2, lastGamma);  
 **int** encoded = (Integer.*parseInt*(String.*valueOf*(plainText.charAt(i))) + gamma) % 2;  
  
 lastGamma = gamma;  
 cipherText.append(encoded);  
 }  
  
 **return** *convertToText*(cipherText);  
 }  
  
 */\*\*  
 \* Гамма-генератор  
 \* Генерирует гамму по заданному алгоритму  
 \*  
 \** ***@param register1*** *- первый регистр  
 \** ***@param register2*** *- второй регистр  
 \** ***@param gamma*** *- прошлое значение гаммы  
 \** ***@return*** *возвращает новое значение гаммы  
 \*/* **private static int** generateGamma(Register register1, Register register2, **int** gamma) {  
 gamma = (gamma + register1.getBit() + register2.getBit()) % 2;  
  
 register1.moveToRight();  
 register2.moveToRight();  
  
 **return** gamma;  
 }  
  
 */\*\*  
 \* Перевод символьной строки в бинарную  
 \*  
 \** ***@param text*** *- символьная строка  
 \** ***@return*** *возвращает бинарную строку  
 \*/* **private static** String convertToBinary(StringBuilder text) {  
 StringBuilder binary = **new** StringBuilder();  
  
 text = *fixCyrillic*(text);  
  
 **for** (**int** i = 0; i < text.length(); i++) {  
 **final char** c = text.charAt(i);  
 StringBuilder charBinary = **new** StringBuilder(Integer.*toBinaryString*(c));  
  
 **while** (charBinary.length() < 8) {  
 charBinary.insert(0, **"0"**);  
 }  
  
 binary.append(charBinary);  
 }  
  
 **return** binary.toString();  
 }  
  
 */\*\*  
 \* Перевод бинарной строки в символьную  
 \*  
 \** ***@param binary*** *- бинарная строка  
 \** ***@return*** *возвращает символьную строку  
 \*/* **private static** String convertToText(StringBuilder binary) {  
 StringBuilder text = **new** StringBuilder();  
  
 **for** (String s : binary.toString().split(**"(?<=\\G.{8})"**)) {  
 text.append((**char**) Integer.*parseInt*(s, 2));  
 }  
  
 **return** *fixCyrillic*(text).toString();  
 }  
  
 */\*\*  
 \* Замена управляющих символов  
 \* Управляющие символы при выводе на консоль не отображаются корректно и способны "сломать" вывод, в связи с чем  
 \* осуществляется их замена на символ ♞  
 \*  
 \** ***@param source*** *- строка, в которой необходимо произвести замену  
 \** ***@return*** *возвращает строку с заменами  
 \*/* **private static** String replaceControlCharacters(String source) {  
 StringBuilder fixed = **new** StringBuilder(source);  
  
 **for** (**int** i = 0; i < fixed.length(); i++) {  
 **if** (fixed.charAt(i) < 32) {  
 fixed.setCharAt(i, **'♞'**);  
 }  
 }  
  
 **return** fixed.toString();  
 }  
  
 */\*\*  
 \* Исправление записи кириллических символов  
 \* Коды символов кириллицы > 1000, тогда как остальные символы располагаются в таблице UTF-8 в диапазоне от 0 до 127  
 \* Исходные коды символов кириллицы заменяются на соответствующие, которые они могли бы занять в UTF-8 в диапазоне от 128 до 255  
 \* Обратно временные коды заменяются на исходные  
 \*  
 \** ***@param source*** *- строка, в которой необходимо произвести исправление  
 \** ***@return*** *возвращает исправленную строку  
 \*/* **private static** StringBuilder fixCyrillic(StringBuilder source) {  
 StringBuilder fixed = **new** StringBuilder(source);  
  
 **for** (**int** i = 0; i < fixed.length(); i++) {  
 **final char** c = fixed.charAt(i);  
  
 **if** (c > 127 && c < 256) {  
 fixed.setCharAt(i, (**char**) (c + 128 \* 7));  
 }  
  
 **if** (c > 1023 && c < 1152) {  
 fixed.setCharAt(i, (**char**) (c % 128 + 128));  
 }  
 }  
  
 **return** fixed;  
 }  
}

**Класс Register.java:**

**package** Structure;  
  
**import** java.util.LinkedList;  
  
*/\*\*  
 \* Линейный регистр сдвига  
 \*/***public class** Register {  
 **private** LinkedList<String> **buffer** = **new** LinkedList<>();  
 **private int**[] **bitPositions**;  
 **private int length**;  
  
 **public** Register(**int** length, **int**[] bitPositions, String value) {  
 **this**.**length** = length;  
 **this**.**bitPositions** = bitPositions;  
 fill(value);  
 }  
  
 */\*\*  
 \* Сдвиг регистра вправо на 1  
 \* Биты в регистре расположены в порядке с последнего по 0  
 \*/* **public void** moveToRight() {  
 **int** newBit = 0;  
  
 **for** (**int** bitPosition : **bitPositions**) {  
 newBit += Integer.*parseInt*(**buffer**.get(bitPosition));  
 }  
  
 **buffer**.add(String.*valueOf*(newBit % 2));  
 **buffer**.removeFirst();  
 }  
  
 */\*\*  
 \* Возврат бита регистра, необходимого для алгоритма вычисления гаммы  
 \*  
 \** ***@return*** *возвращает нулевой бит  
 \*/* **public int** getBit() {  
 **return** Integer.*parseInt*(String.*valueOf*(**buffer**.getFirst()));  
 }  
  
 */\*\*  
 \* Заполнение регистра ключом  
 \*  
 \** ***@param value*** *- ключ  
 \*/* **private void** fill(String value) {  
 **for** (**int** i = 0; i < **length**; i++) {  
 **buffer**.add(String.*valueOf*(value.charAt(i % value.length())));  
 }  
 }  
}

**Результаты работы программы**

