Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Пермский национальный исследовательский политехнический университет»

Электротехнический факультет

Кафедра «Информационные технологии и автоматизированные системы»

Направление 09.03.01 – «Информатика и вычислительная техника»

Дисциплина: «Защита информации»

Профиль: «Автоматизированные системы обработки информации и управления»

Семестр 7

ОТЧЕТ

по лабораторной работе №2

Тема: «Алгоритм RSA»

Выполнил: студент группы АСУ-14-1б

Калмыков В. А. \_\_\_\_\_\_\_\_\_\_\_

Проверил: доцент кафедры ИТАС

Шереметьев В. Г. \_\_\_\_\_\_\_\_\_\_\_

Дата \_\_\_\_\_\_

Пермь, 2017

**ЦЕЛЬ РАБОТЫ**

Получить практические навыки по использованию ассиметричных алгоритмов шифрования, на примере использования алгоритма RSA.

**ЗАДАНИЕ**

**Вариант №6**. Выполнить шифрование текста методом RSA, используя в качестве p и q простые числа с разрядностью не меньшей двадцати.

**ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

**Криптосистема RSA**

Алгоритм RSA (R.Rivest, A.Shamir, L.Adleman) был предложен еще в 1977 году. С тех пор он весьма упорно противостоит различным атакам, и сейчас является самым распространенным криптоалгоритмом в мире. Он входит во многие криптографические стандарты, используется во многих приложениях и секретных протоколах (включая PEM, S-HTTP и SSL).

**Основные принципы работы RSA**

Сначала пара математических определений. Целое число называют простым, если оно делится нацело только на единицу и на само себя, иначе его называют составным. Два целых числа называют взаимно простым, если их наибольший общий делитель (НОД) равен 1.

Алгоритм работы RSA таков. Сначала надо получить открытый и секретный ключи:

1. Выбираются два простых числа p и q
2. Вычисляется их произведение n(=p\*q)
3. Выбирается произвольное число e (e<n), такое, что НОД(e,(p-1)(q-1))=1, то есть e должно быть взаимно простым с числом (p-1)(q-1).
4. Методом Евклида решается в целых числах уравнение e\*d+(p-1)(q-1)\*y=1. Здесь неизвестными являются переменные d и y – метод Евклида как раз и находит множество пар (d,y), каждая из которых является решением уравнения в целых числах.
5. Два числа (e,n) – публикуются как открытый ключ.
6. Число d хранится в строжайшем секрете – это и есть закрытый ключ, который позволит читать все послания, зашифрованные с помощью пары чисел (e,n).

Как же производится собственно шифрование с помощью этих чисел:

1. Отправитель разбивает свое сообщение на блоки, равные k=[log2(n)] бит, где квадратные скобки обозначают взятие целой части от дробного числа.
2. Подобный блок, как Вы знаете, может быть интерпретирован как число из диапазона (0;2k-1). Для каждого такого числа (назовем его mi) вычисляется выражение ci=((mi)e)mod n. Блоки ci и есть зашифрованное сообщение Их можно спокойно передавать по открытому каналу, поскольку операция возведения в степень по модулю простого числа, является необратимой математической задачей. Обратная ей задача носит название «логарифмирование в конечном поле» и является на несколько порядков более сложной задачей. То есть даже если злоумышленник знает числа e и n, то по ci прочесть исходные сообщения mi он не может никак, кроме как полным перебором mi.

А вот на приемной стороне процесс дешифрования все же возможен, и поможет нам в этом хранимое в секрете число d. Достаточно давно была доказана теорема Эйлера, частный случай которой утверждает, что если число n представимо в виде двух простых чисел p и q, то для любого x имеет место равенство (x(p-1)(q-1))mod n = 1. Для дешифрования RSA-сообщений воспользуемся этой формулой. Возведем обе ее части в степень (-y) : (x(-y)(p-1)(q-1))mod n = 1(-y) = 1. Теперь умножим обе ее части на x : (x(-y)(p-1)(q-1)+1)mod n = 1\*x = x.

А теперь вспомним как мы создавали открытый и закрытый ключи. Мы подбирали с помощью алгоритма Евклида d такое, что e\*d+(p-1)(q-1)\*y=1, то есть e\*d=(-y)(p-1)(q-1)+1. А следовательно в последнем выражении предыдущего абзаца мы можем заменить показатель степени на число (e\*d). Получаем (xe\*d)mod n = x. То есть для того чтобы прочесть сообщение ci=((mi)e)mod n достаточно возвести его в степень d по модулю m : ((ci)d)mod n = ((mi)e\*d)mod n = mi.

На самом деле операции возведения в степень больших чисел достаточно трудоемки для современных процессоров, даже если они производятся по оптимизированным по времени алгоритмам. Поэтому обычно весь текст сообщения кодируется обычным блочным шифром (намного более быстрым), но с использованием ключа сеанса, а вот сам ключ сеанса шифруется как раз асимметричным алгоритмом с помощью открытого ключа получателя и помещается в начало файла.

В 1990-х годах с помощью распределенных вычислений через Internet предпринимались удачные попытки факторизовать некоторые произвольные большие числа максимальное факторизованное число имело 140 десятичных разрядов (1999 год), на что ушло около 2000 MY (Mips/Year – годовая работа компьютера мощностью в миллион целочисленных операций в секунду). Учитывая это, сейчас специалисты (Лаборатория RSA, www. rsa.com / rsalabs ) рекомендуют использовать минимальную длину ключа n, не менее чем 768 бит (~230 десятичных рзрядов) для малосекретной информации, 1024 бит для обычной и 2048 для особо секретной. Использующаяся в старых продуктах длина ключа в 512 бит (~160 разрядов) уже под угрозой взлома. Известно, что RSA имеет низкую криптостойкость при шифровании коротких блоков. В таких случаях злоумышленник может взять от блока шифротекста корень степени e по модулю n, что в данном случае будет намного быстрее факторизации. Поэтому короткие блоки обязательно надо «набивать» дополнительными битами. Еще один интересный вопрос касается простых чисел. Для построения ключей алгоритму RSA необходимо найти два простых числа. Благо, среди чисел простые попадаются довольно часто: на отрезке от 1 до n примерно n/ln(n) чисел являются простыми. Поэтому можно просто брать псевдослучайные числа нужной длины и проверять их на простоту. Проверку числа на простоту можно делать двумя способами: «в лоб», перебором всех его делителей (от 2 до округленного корня из n), или с помощью более «хитрых» тестов на делимость. При переборе всех делителей мы гарантированно можем утверждать, что прошедшее такую проверку число является простым. Однако, время работы такой процедуры будет очень велико. Среди «хитрых» тестов надо выделить тест Миллера–Рабина, так как он на сегодняшний день является наиболее лучшим по всем параметрам. Так вот, проверка ллера–Рабина работает намного быстрее перебора делителей, но в отличие от него, число, выдаваемое им как простое, с некоторой очень маленькой вероятностью может оказаться составным. На практике обычно применяют последовательно несколько разных «хитрых» тестов, минимизируя тем самым вероятность ошибки. Программная реализация RSA работает медленнее примерно на два порядка по сравнению с симметричными алгоритмами. RSA часто используют вместе с каким-нибудь симметричным шифром. При таком способе все сообщения шифруются с помощью более быстрого симметричного алгоритма, а для пересылки сессионного секретного ключа этого симметричного алгоритма используется RSA. Получается вычислительно дешево и сердито.

Проясним использование алгоритма RSA на конкретном примере. Выбираем два простые числа p=7; q=17 (на практике эти числа во много раз длиннее). В этом случае n = p\*q будет равно 119. Теперь необходимо выбрать e, выбираем e=5. Следующий шаг связан с формированием числа d так, чтобы (d\*e) mod [(p-1)(q-1)] =1. d=77 (использован расширенный алгоритм Эвклида). d – секретный ключ, а e и n характеризуют открытый ключ. Пусть текст, который нам нужно зашифровать представляется M=19. С = Memod n. Получаем зашифрованный текст C=66. Этот «текст» может быть послан соответствующему адресату. Получатель дешифрует полученное сообщение, используя М= Cdmod n и C=66. В результате получается M=19.

На практике общедоступные ключи могут помещаться в специальную базу данных. При необходимости послать партнеру зашифрованное сообщение можно сделать сначала запрос его открытого ключа. Получив его, можно запустить программу шифрации, а результат ее работы послать адресату. На использовании общедоступных ключей базируется и так называемая электронная подпись, которая позволяет однозначно идентифицировать отправителя. Сходные средства могут применяться для предотвращения внесения каких-либо корректив в сообщение на пути от отправителя к получателю. Быстродействующие аппаратные 512-битовые модули могут обеспечить скорость шифрования на уровне 64 кбит в сек. Готовятся ИС, способные выполнять такие операции со скоростью 1 Мбайт/сек. Разумный выбор параметра e позволяет заметно ускорить реализацию алгоритма.

**ХОД РАБОТЫ**

На рисунке 1 представлена главная форма программы.

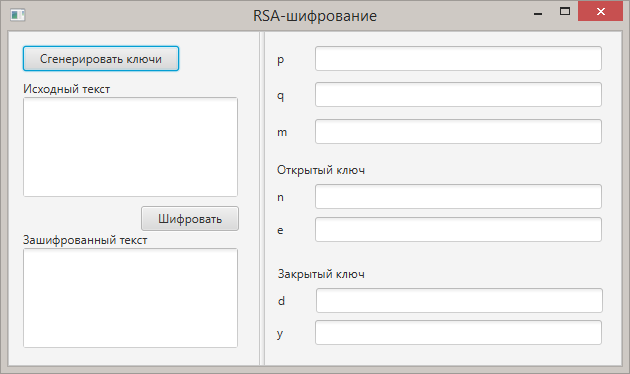


Рисунок 1 – Главная форма программы.

Пример работы программы представлен на рисунке 2.

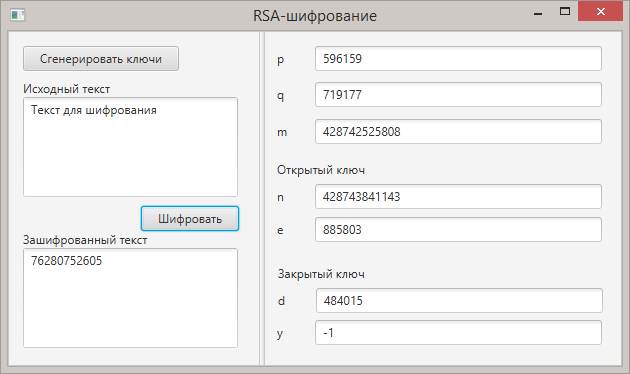


Рисунок 2 – Пример работы программы.

**ПРИЛОЖЕНИЕ А**

**Листинг класса RSA.java**

package Models;

import java.math.BigInteger;

import java.util.Objects;

import java.util.Random;

public class RSA {

private BigInteger p;

private BigInteger q;

private BigInteger m;

private BigInteger y = new BigInteger("-1");

// Открытый ключ

private BigInteger n;

private BigInteger e;

// Закрытый ключ

private BigInteger privateKey;

// Разрядность

private short cap = 20;

public RSA() {

Random rnd = new Random();

BigInteger one = BigInteger.ONE;

BigInteger gcd;

this.p = BigInteger.probablePrime(this.cap, rnd);

do {

this.q = BigInteger.probablePrime(this.cap, rnd);

} while (Objects.equals(this.p, this.q));

this.n = this.p.multiply(this.q);

this.m = this.p.subtract(one).multiply(this.q.subtract(one));

do {

this.e = BigInteger.probablePrime(this.cap, rnd);

gcd = this.e.gcd(this.m);

} while (!gcd.equals(one));

this.privateKey = one.add(this.m).divide(this.e);

}

public BigInteger getP() {

return this.p;

}

public BigInteger getQ() {

return q;

}

public BigInteger getPrivateKey() {

return privateKey;

}

public BigInteger getE() {

return e;

}

public BigInteger getN() {

return n;

}

public BigInteger getM() {

return m;

}

public BigInteger getY() {

return y;

}

public BigInteger encrypt(BigInteger message) {

return message.modPow(this.e, this.n);

}

}

**ПРИЛОЖЕНИЕ Б**

**Листинг класса SampleController.java**

package Forms;

import Models.RSA;

import javafx.fxml.FXML;

import javafx.scene.control.TextArea;

import javafx.scene.control.TextField;

import java.math.BigInteger;

public class SampleController {

@FXML

private TextArea txtText;

@FXML

private TextArea txtEncryptText;

@FXML

private TextField txtP;

@FXML

private TextField txtQ;

@FXML

private TextField txtM;

@FXML

private TextField txtN;

@FXML

private TextField txtE;

@FXML

private TextField txtD;

@FXML

private TextField txtY;

private RSA rsa;

public SampleController() {

}

@FXML

private void initialize() {

}

@FXML

private void handleGenerateKeys() {

this.rsa = new RSA();

this.txtP.setText(this.rsa.getP().toString());

this.txtQ.setText(this.rsa.getQ().toString());

this.txtM.setText(this.rsa.getM().toString());

this.txtN.setText(this.rsa.getN().toString());

this.txtE.setText(this.rsa.getE().toString());

this.txtD.setText(this.rsa.getPrivateKey().toString());

this.txtY.setText(this.rsa.getY().toString());

}

@FXML

private void handleEncryptText() {

this.txtEncryptText.setText("");

BigInteger message = new BigInteger(this.txtText.getText().getBytes());

BigInteger encryptMessage = this.rsa.encrypt(message);

this.txtEncryptText.setText(encryptMessage.toString());

}

}

**ПРИЛОЖЕНИЕ В**

**Листинг класса Main.java**

import javafx.application.Application;

import javafx.fxml.FXMLLoader;

import javafx.scene.Parent;

import javafx.scene.Scene;

import javafx.stage.Stage;

public class Main extends Application {

@Override

public void start(Stage primaryStage) throws Exception {

Parent root = FXMLLoader.load(getClass().getResource("/Forms/sample.fxml"));

primaryStage.setTitle("RSA-шифрование");

primaryStage.setScene(new Scene(root));

primaryStage.show();

}

public static void main(String[] args) {

launch(args);

}

}