­Министерство высшего и профессионального образования РФ

Государственное образовательное учреждение

высшего профессионального образования

Ульяновский государственный технический университет

Кафедра «Вычислительная техника»

Дисциплина "МиСЗКИ"

Лабораторная работа №3

Использование систем шифрования с открытым ключом.

Выполнил:

студент группы БЭВМд-41

Ключников Дмитрий

Проверил:

Мартынов А. И.

Ульяновск 2012

***Задание:***

1. Изучить теоретические основы построения систем с открытым ключом (СОК) и схемы распределения открытых ключей.
2. Изучить алгоритмы оптимизации наиболее сложных вычислительных аспектов СОК (тесты Рабина-Миллера и Лемана, алгоритм Евклида, расширенный алгоритм Евклида, алгоритмы ускоренного умножения в конечном поле).
3. Реализовать следующие функциональные блоки:
   * **Генератор ключей** – модуль, предназначенный для генерации пары ключей (открытого и закрытого). *Входные данные* – левая граница диапазона, с которой начинается процесс поиска простых чисел. *Выходные данные* – файлы close\_key.txt и open\_key.txt, содержащие значения полученных ключей. *Требования по функциональности*: в процессе генерации ключей программа выдает информацию о состоянии процесса (Progress Bar или что-то подобное); пользователь должен иметь возможность прервать процесс генерации в любое время.
   * **Шифратор/Дешифратор** – модуль, осуществляющий кодирование/декодирование текстовых файлов по схеме, указанной в Таблице 1 согласно варианту. *Входные-выходные данные*: файл close\_key.txt или open\_key.txt в зависимости от режима использования; файл pass.txt или pass.cod в зависимости от режима использования. Файл pass.txt содержит текстовый пароль, который необходимо зашифровать для последующего использования и записать в файл pass.cod. Файл pass. cod содержит зашифрованный текстовый пароль, который необходимо расшифровать для последующего использования и записать в файл pass.txt. *Требования по функциональности*: в процессе кодирования/декодирования программа выдает информацию о состоянии процесса (Progress Bar или что-то подобное); пользователь должен иметь возможность прервать процесс кодирования/декодирования в любое время; время обработки для текстового файла размером 10-30 символов не должно превышать 20 сек.
   * **Блочный шифр** – берется из предыдущей лабораторной работы.
   * **Подсистему управления** – модуль, обеспечивающий частичную автоматизацию следующего сценария (эти пункты отмечены звездочкой) разбитого на шесть этапов:
     1. Пользователь создает в текущей директории две папки Sender (имитация отправителя) и Recipient (имитация получателя), копирует в эти папки разработанную программу, а также исходные данные: в папку Sender – файл pass.txt и файл message.txt.
     2. \*Запускает генератор ключей и распределяет ключи (файл close\_key.txt кладет в папку Recipient, а файл open\_key.txt в папку Sender).
     3. \*Запускает шифратор в папке Sender, на вход которого подает файл с текстовым паролем pass.txt и файл open\_key.txt и результат кодирования (файл pass.cod) сохраняет в папке и Recipient.
     4. \*Запускает дешифратор в папке Recipient, на вход которого подает файл с закодированным паролем pass.cod и файл close\_key.txt и результат декодирования (файл pass.txt) сохраняет в папке и Recipient.
     5. \*Запускает программу блочного шифрования в режиме кодирования в папке Sender, на вход которой подается пароль (файл pass.txt) и исходное сообщение (message.txt). Результат кодирования (файл message.cod) сохраняет в папке Recipient.
     6. \*Запускает программу блочного шифрования в режиме декодирования в папке Recipient, на вход которой подается пароль (файл pass.txt) и зашифрованное сообщение (message.cod). Результат декодирования (файл message.txt) сохраняет в папке Recipient.
     7. Пользователь сравнивает полученные результаты и если файлы message.txt в папках Recipient и Sender совпадают, то процесс кодирования считается успешно завершенным.

*Входные данные*: перед запуском управляющей программы папки должны содержать следующие исходные файлы с данными: в папке Sender файлы pass.txt и message.txt, в папке Recipient должно быть пусто.

*Входные данные*: после успешной работы программы в папке Recipient должны находится файлы close\_key.txt (закрытый ключ), pass.cod (закодированный пароль), pass.txt (расшифрованный пароль), message.cod (зашифрованное сообщение) и message.txt ( расшифрованное сообщение).

*Функциональные требования*: каждый шаг сценария запускается отдельно, для того чтобы можно было проконтролировать процесс обмена информацией.

|  |  |  |
| --- | --- | --- |
| **№ Этапа** | **Что добавляется в папку Sender** | **Что добавляется в папку Recipient** |
| 1 | pass.txt, message.txt |  |
| 2 | open\_key.txt | close\_key.txt |
| 3 |  | pass.cod |
| 4 |  | pass.txt |
| 5 |  | message.cod |
| 6 |  | message.txt |

1. Реализовать систему шифрования в соответствии с вариантами указанными в Таблице 1 (*А – Алгоритм СОК, В – Способ получения простых чисел*)

***Дополнительные требования к лабораторной работе:***

1. Паролем может быть любая последовательность символов (русских и английских, цифр, знаков препинания и т.д.).
2. Программа должна быть оформлена в виде удобной утилиты, позволяющей работать с любыми файлами.
3. Программа должна обеспечивать шифрование файлов произвольной длины.
4. Текст программы оформляется прилично (удобочитаемо, с описанием ВСЕХ функций, переменных и критических мест).
5. В процессе работы программа ОБЯЗАТЕЛЬНО выдает информацию о состоянии процесса кодирования/декодирования.
6. После завершения работы программы выдает информацию о скорости шифрования / дешифрования (символ /сек)
7. Интерфейс программы может быть произвольным, но удобным и понятным (разрешается использование библиотек VCL)
8. Среда разработки и язык программирования могут быть произвольными.

***Задание по варианту:***

Алгоритм СОК: Алгоритм RSA

Способ получения простых чисел: Тест Рабина-Миллера

**Алгоритм RSA**

В основу криптографической системы с открытым ключом RSA положена сложность задачи факторизации произведения двух больших простых чисел. Для шифрования используется операция возведения в степень по модулю большого числа. Для дешифрования за разумное время (обратной операции) необходимо уметь вычислять функцию Эйлера от данного большого числа, для чего необходимо знать разложения числа на простые множители.

В криптографической системе с открытым ключом каждый участник располагает как открытым ключом (англ. *public key*), так и закрытым ключом (англ. *private key*). В криптографической системе RSA каждый ключ состоит из пары целых чисел. Каждый участник создаёт свой открытый и закрытый ключ самостоятельно. Закрытый ключ каждый из них держит в секрете, а открытые ключи можно сообщать кому угодно или даже публиковать их. Открытый и закрытый ключи каждого участника обмена сообщениями в криптосистеме RSA образуют «согласованную пару» в том смысле, что они являются *взаимно обратными*, то есть:

\forall сообщения m \in M, где M — множество допустимых сообщений

\forall допустимых открытого и закрытого ключей P и S

\exist\, соответствующие функции шифрования E_p(x) и расшифрования D_s(x), такие что

m=D_s(E_p(m))=E_p(D_s(m)).

### Алгоритм создания открытого и секретного ключей

RSA-ключи генерируются следующим образом:

1. Выбираются два различных случайных простых числа p и q заданного размера (например, 1024 бита каждое).
2. Вычисляется их произведение n=pq, которое называется *модулем*.
3. Вычисляется значение функции Эйлера от числа n:

\varphi(n) = (p-1)(q-1).

1. Выбирается целое число e (1 < e < \varphi(n)), взаимно простое со значением функции \varphi(n). Обычно в качестве e берут простые числа, содержащие небольшое количество единичных бит в двоичной записи, например, простые числа Ферма 17, 257 или 65537.
   * Число e называется *открытой экспонентой* (англ. *public exponent*)
   * Время, необходимое для шифрования с использованием быстрого возведения в степень, пропорционально числу единичных бит в e.
   * Слишком малые значения e, например 3, потенциально могут ослабить безопасность схемы RSA.
2. Вычисляется число d, мультипликативно обратное к числу e по модулю \varphi(n), то есть число, удовлетворяющее условию:

d e \equiv 1 \mod {\varphi(n)}.

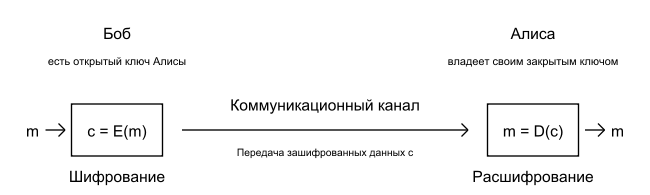
* + Число d называется *секретной экспонентой*. Обычно, оно вычисляется при помощи расширенного алгоритма Евклида.

1. Пара \left\{ e, n \right\} публикуется в качестве *открытого ключа RSA* (англ. *RSA public key*).
2. Пара \left\{ d, n \right\} играет роль *закрытого ключа RSA* (англ. *RSA private key*) и держится в секрете.

### Шифрование и расшифрование

Предположим, Боб хочет послать Алисе сообщение m.

Сообщениями являются целые числа в интервале от 0 до n - 1, т.е m \in \mathbb{Z}_{n}\,.

[](http://commons.wikimedia.org/wiki/File:Public_key_encryption,_transmission_and_decryption_light-ru-rendered.svg?uselang=ru)

|  |  |
| --- | --- |
| **Алгоритм**:   * Взять *открытый ключ* (e,n) Алисы * Взять *открытый текст* m * Зашифровать сообщение с использованием открытого ключа Алисы:   c = E(m) = m^e \mod n ~~~~ (1) | **Алгоритм**:   * Принять зашифрованное сообщение C \, * Взять свой *закрытый ключ* (d,n) * Применить закрытый ключ для расшифрования сообщения:   m = D(c) = c^d \mod n ~~~~(2) |

**Тест Миллера — Рабина**

**Тест Миллера — Рабина** — вероятностный полиномиальный тест простоты. Тест Миллера — Рабина позволяет эффективно определять, является ли данное число составным. Однако, с его помощью нельзя строго доказать простоту числа.

Свидетели простоты и теорема Рабина

Пусть ~m — нечётное число большее 1. Число ~m-1 однозначно представляется в виде m-1 = 2^s \cdot t, где ~t нечётно. Целое число ~a, ~1 < a < m, называется **свидетелем простоты**числа ~m, если выполняется одно из условий:

* ~a^t\equiv 1\pmod m

или

* существует целое число ~k, ~0\leq k<s, такое, что ~ a^{2^kt}\equiv m-1\pmod m.

**Теорема Рабина** утверждает, что составное нечётное число m имеет не более \varphi(m)/4 различных свидетелей простоты, где \varphi(m) — функция Эйлера.

**Алгоритмы нахождения взаимно простых больших чисел**

Два числа называются взаимно простыми, если у них нет общих множителей кроме 1. Иными словами, если наибольший общий делитель a и n равен 1. Это записывается как:

.

Взаимно просты числа 15 и 28. 15 и 27 не являются взаимно простыми, а 13 и 500 - являются. Простое число взаимно просто со всеми другими числами, кроме чисел, кратных данному простому числу. Одним из способов вычислить наибольший общий делитель двух чисел является алгоритм Эвклида.

**Алгоритм Евклида для целых чисел**

Пусть *a* и *b* — целые числа, не равные одновременно нулю, и последовательность чисел

 a > b > r_1 > r_2 > r_3 > r_4 > \cdots >r_n

определена тем, что каждое *rk* — это остаток от деления предпредыдущего числа на предыдущее, а предпоследнее делится на последнее нацело, то есть

Тогда НОД(*a*,*b*), наибольший общий делитель *a* и *b*, равен *rn*, последнему ненулевому члену этой последовательности.

**Алгоритмы возведения в степень в конечном поле**

В системах с открытым ключом активно используется техника возведения в степень в конечном поле или, как ее еще называют, арифметика вычетов. Такой подход позволяет избежать ситуации переполнения разрядной сетки при работе с большими числами.

Арифметика вычетов очень похожа на обычную арифметику: она коммутативна, ассоциативна и дистрибутивна. Кроме того, приведение каждого промежуточного результата по модулю n дает тот же результат, как и выполнение всего вычисления с последующим приведением конечного результата по модулю n.

Вычисление mod n часто используется в криптографии, так как вычисление дискретных логарифмов и квадратных корней mod n может быть нелегкой проблемой. Арифметика вычетов, к тому же, легче реализуется на компьютерах, поскольку она ограничивает диапазон промежуточных значений и результата. Для k-битовых вычетов n, промежуточные результаты любого сложения, вычитание или умножения будут не длиннее, чем 2k бит.

Поэтому в арифметике вычетов мы можем выполнить возведение в степень без огромных промежуточных результатов. Вычисление степени некоторого числа по модулю другого числа, a x mod n, представляет собой просто последовательность умножений и делений, но существуют приемы, ускоряющие это действие. Один из таких приемов стремится минимизировать количество умножений по модулю, другой - оптимизировать отдельные

умножения по модулю. Так как операции дистрибутивны, быстрее выполнить возведение в степень как поток последовательных умножений, каждый раз получая вычеты. Сейчас вы не чувствуете разницы, но она будет заметна при умножении 200-битовых чисел.

Например, если вы хотите вычислить a 8 mod n, не выполняйте наивно семь умножений и одно приведение по модулю:

Вместо этого выполните три меньших умножения и три меньших приведения по модулю:

Точно также,

Вычисление ax , где x не является степенью 2, ненамного труднее.

Двоичная запись представляет x в виде суммы степеней двойки: 25 – это

бинарное 11001, поэтому 25 = 24 + 23 + 20. Поэтому:

С продуманным сохранением промежуточных результатов вам

понадобится только шесть умножений:

Такой прием называется цепочкой сложений, или методом двоичных

квадратов и умножения. Он использует простую и очевидную цепочку

сложений, в основе которой лежит двоичное представление числа.

**Код программы**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Numerics;

using System.Security.Cryptography;

using System.Text;

namespace CommonLibrary

{

// Miller-Rabin primality test as an extension method on the BigInteger type.

// Based on the Ruby implementation on this page.

public static class BigIntegerExtensions

{

public static bool IsProbablePrime(this BigInteger source, int certainty)

{

if (source == 2 || source == 3)

return true;

if (source < 2 || source % 2 == 0)

return false;

BigInteger t = source - 1;

int s = 0;

while (t % 2 == 0)

{

t /= 2;

s += 1;

}

// There is no built-in method for generating random BigInteger values.

// Instead, random BigIntegers are constructed from randomly generated

// byte arrays of the same length as the source.

RandomNumberGenerator rng = RandomNumberGenerator.Create();

byte[] bytes = new byte[source.ToByteArray().LongLength];

BigInteger a;

for (int i = 0; i < certainty; i++)

{

do

{

rng.GetBytes(bytes);

a = new BigInteger(bytes);

} while (a < 2 || a >= source - 2);

BigInteger x = BigInteger.ModPow(a, t, source);

if (x == 1 || x == source - 1)

continue;

for (int r = 1; r < s; r++)

{

x = BigInteger.ModPow(x, 2, source);

if (x == 1)

return false;

if (x == source - 1)

break;

}

if (x != source - 1)

return false;

}

return true;

}

public static BigInteger GenPrimeBigInteger(byte length)

{

RandomNumberGenerator rng = RandomNumberGenerator.Create();

byte[] bytes = new byte[length];

BigInteger a;

int i = 0;

do

{

i++;

rng.GetBytes(bytes);

a = new BigInteger(bytes);

} while (!a.IsProbablePrime(1));

return a;

}

public static BigInteger GenBigInteger(byte length)

{

RandomNumberGenerator rng = RandomNumberGenerator.Create();

byte[] bytes = new byte[length];

BigInteger a;

do

{

rng.GetBytes(bytes);

a = new BigInteger(bytes);

} while (a.Sign != 1);

return a;

}

public static int gcd(int a, int b)

{

while (b != 0)

b = a % (a = b);

return a;

}

public static BigInteger gcd(BigInteger a, BigInteger b)

{

var aa = new BigInteger();

aa = BigInteger.Add(aa, a);

var bb = new BigInteger();

bb = BigInteger.Add(bb, b);

while (bb != 0)

{

var temp = BigInteger.ModPow(aa, 1, bb);

aa = bb;

bb = temp;

}

return aa;

}

public static BigInteger Pow(BigInteger x, BigInteger y)

{

if (y.CompareTo(BigInteger.Zero) < 0)

throw new ArgumentException();

BigInteger z = x; // z will successively become x^2, x^4, x^8, x^16, x^32...

BigInteger result = BigInteger.One;

byte[] bytes = y.ToByteArray();

for (int i = bytes.Length - 1; i >= 0; i--)

{

byte bits = bytes[i];

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

{

if ((bits & 1) != 0)

result = BigInteger.Multiply(result, z);

// short cut out if there are no more bits to handle:

if ((bits >>= 1) == 0 && i == 0)

return result;

z = BigInteger.Multiply(z, z);

}

}

return result;

}

public static Tuple<BigInteger, BigInteger, BigInteger, BigInteger> GenRSAValues(byte lengthKey)

{

var P = BigIntegerExtensions.GenPrimeBigInteger(lengthKey);

var Q = BigIntegerExtensions.GenPrimeBigInteger(lengthKey);

var N = BigInteger.Multiply(P, Q);

var M = BigInteger.Multiply(P - 1, Q - 1);

var a = 8;

var b = 19;

var r = BigIntegerExtensions.gcd(a, b);

BigInteger D = BigInteger.One;

BigInteger E = BigInteger.One;

BigInteger X = BigInteger.One;

int i = 2000000;

do

{

if (i < 1000000)

X = (i++) \* M + 1;

else

{

do

{

D = BigIntegerExtensions.GenBigInteger((byte)(3));

} while (BigIntegerExtensions.gcd(M, D) != 1);

i = 2;

}

} while (BigInteger.ModPow(X, 1, D) != 0);

E = X / D;

return new Tuple<BigInteger, BigInteger, BigInteger, BigInteger>(D, N, E, N);

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace CommonLibrary

{

public static class RabinMiller

{

public static bool IsPrime(int n, int k)

{

if (n < 2)

{

return false;

}

if (n != 2 && n % 2 == 0)

{

return false;

}

int s = n - 1;

while (s % 2 == 0)

{

s >>= 1;

}

Random r = new Random();

for (int i = 0; i < k; i++)

{

double a = r.Next((int)n - 1) + 1;

int temp = s;

int mod = (int)Math.Pow(a, (double)temp) % n;

while (temp != n - 1 && mod != 1 && mod != n - 1)

{

mod = (mod \* mod) % n;

temp = temp \* 2;

}

if (mod != n - 1 && temp % 2 == 0)

{

return false;

}

}

return true;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Numerics;

using System.Text;

namespace CommonLibrary

{

public static class RSAEx

{

public static BigInteger EnCrypt(BigInteger data, BigInteger D, BigInteger N)

{

var newDatas = BigInteger.ModPow(data, D, N);

return newDatas;

}

public static BigInteger DeCrypt(BigInteger data, BigInteger D, BigInteger N)

{

var newDatas = BigInteger.ModPow(data, D, N);

return newDatas;

}

}

}