Учреждение образования

Белорусский государственный университет

информатики и радиоэлектроники

Факультет компьютерных систем и сетей

Кафедра информатики

Отчет по лабораторной работе №3

**Асимметричная криптография.**

**RSA**

|  |  |
| --- | --- |
| Выполнил:  студент группы 653501  Ковалевская Алина | Проверил:  Артемьев В.С. |

Минск, 2019

**Задача**

Реализовать программные средства шифрования и дешифрования текстовых файлов при помощи алгоритма RSA.

**Введение**

Как известно, в настоящее время одним из наиболее надежных и эффективных методов защиты информации является шифрование, представляющее собой метод преобразования информации в зашифрованный текст с той целью, чтобы доступ к ней смог получить лишь пользователь, у которого есть необходимый ключ для дешифровки.

Ранее информация шифровалась и расшифровывалась при помощи одного и того же криптографического ключа. Однако в этом случае существует проблема: как передать такой ключ получателю, чтобы он с его помощью смог расшифровать ваше сообщение? Он должен передаваться либо при личной встрече, либо по надежным защищенным каналам связи, что позволит предотвратить перехват ключа посторонними. Но это не всегда удобно и достаточно проблематично.

Данная проблема решается при помощи ассиметричной криптографии. В данном случае у пользователя имеется так называемая ключевая пара, состоящая из закрытого ключа (Private Key) и открытого ключа (Public Key). Открытый ключ предназначен для массового распространения — вы отправляете его другим пользователям, публикуете на открытых серверах ключей и т. д. для того чтобы все желающие могли зашифровать с его помощью сообщение для вас.

Интересно то, что после того, как сообщение зашифровано, расшифровать его сможет лишь владелец закрытого ключа, который находится в паре с открытым ключом, то есть только вы и никто другой, даже отправитель сообщения.

Это возможно по тому, что открытый и закрытый ключи связаны между собой по особой математической зависимости, но получить из открытого ключа закрытый не представляется возможным.

В основе ассиметричной криптографии лежит алгоритм Диффи-Хеллермана, двух ученых, сформулировавших модель криптографической системы с открытым ключом. Впоследствии три других ученых Р. Ривест, А. Шамир и Л. Адлеман создали ассиметричный алгоритм шифрования RSA (от первых букв фамилий его создателей), который сегодня получил повсеместное распространение и используется как для шифрования/дешифрования сообщений, так и для создания электронно-цифровых подписей.

Ассиметричная криптография сегодня еще известна как криптография на основе инфраструктуры открытых ключей (Public Key Infrastructure), она широко используется по всему миру как серьезными структурами и организациями (например, министерством обороны США), так и рядовыми пользователями.

**Алгоритм RSA**

RSA (аббревиатура от фамилий Rivest, Shamir и Adleman) — криптографический алгоритм с открытым ключом, основывающийся на вычислительной сложности задачи факторизации больших целых чисел.

Криптосистема RSA стала первой системой, пригодной и для шифрования, и для цифровой подписи. Алгоритм используется в большом числе криптографических приложений, включая PGP, S/MIME, TLS/SSL, IPSEC/IKE и других.

Криптографические системы с открытым ключом используют так называемые односторонние функции, которые обладают следующим свойством:

– если известно x, то f(x) вычислить относительно просто;

– если известно y=f(x), то для вычисления x нет простого (эффективного) пути.

Под односторонностью понимается не теоретическая однонаправленность, а практическая невозможность вычислить обратное значение, используя современные вычислительные средства, за обозримый интервал времени.

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 (обратной операции) за разумное время необходимо уметь вычислять функцию Эйлера от данного большого числа, для чего необходимо знать разложение 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:

Для любых допустимых пар открытого и закрытого ключей (p,s)

существуют соответствующие функции шифрования Ep(x) и расшифрования Ds(x) такие, что для любого сообщения m из M, где M — множество допустимых сообщений, m = Ds(Ep(m)) = Ep(Ds(m)).

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=p\*q, которое называется модулем.
3. Вычисляется значение функции Эйлера от числа n:

phi(n)=(p-1)\*(q-1)

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

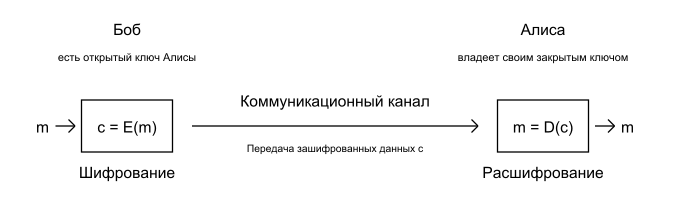
d\*e =1 (mod phi(n)). Число d называется секретной экспонентой. Обычно оно вычисляется при помощи расширенного алгоритма Евклида.

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

**Блок-схема алгоритма**

Предположим, Боб хочет послать Алисе сообщение m.

Сообщениями являются целые числа в интервале от 0 до n-1.



Алгоритм шифрования:

1. Взять открытый ключ (e,n) Алисы
2. Взять открытый текст m
3. Зашифровать сообщение с использованием открытого ключа Алисы: c=E(m)=me mod n

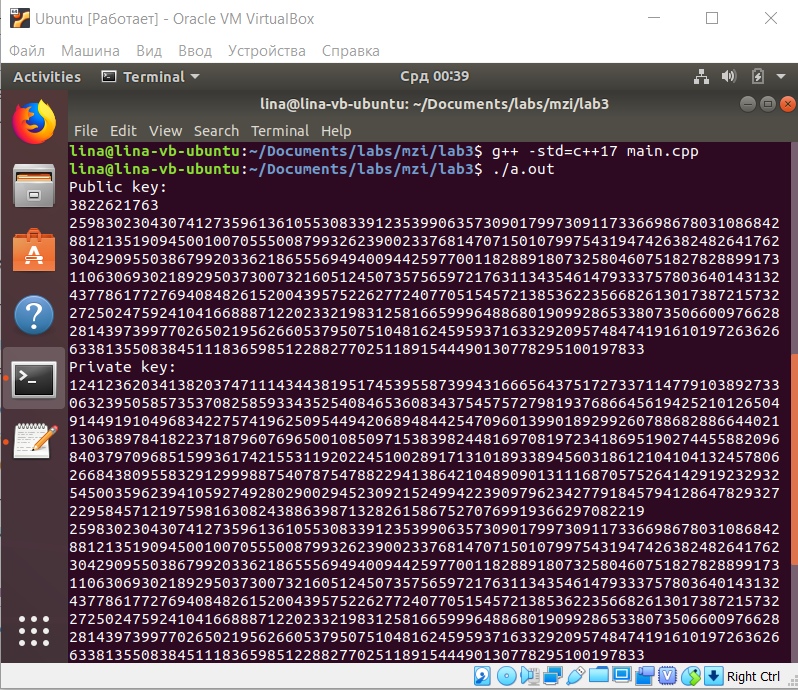
Алгоритм расшифрования:

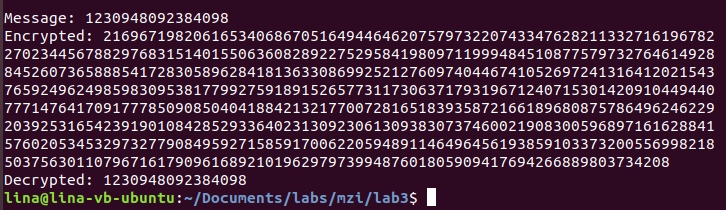
1. Принять зашифрованное сообщение c.
2. Взять свой закрытый ключ (d,n).
3. Применить закрытый ключ для расшифрования сообщения: m=D(c)= cd mod n.

Данная схема на практике не используется по причине того, что она не является практически надёжной (semantically secured). Действительно, односторонняя функция E(m) является детерминированной — при одних и тех же значениях входных параметров (ключа и сообщения) выдаёт одинаковый результат. Это значит, что не выполняется необходимое условие практической (семантической) надёжности шифра.

Наиболее используемым в настоящее время является смешанный алгоритм шифрования, в котором сначала шифруется сеансовый ключ, а потом уже с его помощью участники шифруют свои сообщения симметричными системами. После завершения сеанса сеансовый ключ, как правило, уничтожается.

**Демонстрация работы программы**





**Вывод**

На данный момент асимметричное шифрование на основе открытого ключа RSA (расшифровывается, как Rivest, Shamir and Aldeman - создатели алгоритма) использует большинство продуктов на рынке информационной безопасности.

Его криптостойкость основывается на сложности разложения на множители больших чисел, а именно - на исключительной трудности задачи определить секретный ключ на основании открытого, так как для этого потребуется решить задачу о существовании делителей целого числа. Наиболее криптостойкие системы используют 1024-битовые и большие числа.

**Программный код**

**main.cpp**

#include <boost/multiprecision/cpp\_int.hpp>

#include <chrono>

#include <ratio>

#include <thread>

#include <atomic>

#include <vector>

#include <random>

#include <utility>

using uint1024 = boost::multiprecision::number<boost::multiprecision::cpp\_int\_backend<4096, 4096,

boost::multiprecision::signed\_magnitude, boost::multiprecision::unchecked, void>>;

struct TRSAKeys {

std::pair<uint1024, uint1024> PublicKey;

std::pair<uint1024, uint1024> PrivateKey;

TRSAKeys(uint1024 openKey, uint1024 privateKey, uint1024 module)

: PublicKey{openKey, module}

, PrivateKey{privateKey, module}

{}

};

uint1024 BinPower(const uint1024& base, uint1024 power, const uint1024& mod) {

if (power == 0) {

return 1;

} else {

auto value = BinPower(base, power / 2, mod);

value = (value \* value) % mod;

if (power % 2 != 0) {

value = (value \* base) % mod;

}

return value;

}

}

template<class Function>

void TestPerfomance(Function function) {

auto start = std::chrono::high\_resolution\_clock::now();

function();

auto finish = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double, std::milli> time\_ms = finish - start;

std::cout << "function took: " << std::fixed << time\_ms.count() << " ms" << std::endl;

}

namespace {

void PrintKeys(const TRSAKeys& rsaKeys) {

std::cout << "Public key:\n" << rsaKeys.PublicKey.first << "\n" << rsaKeys.PublicKey.second;

std::cout << std::endl;

std::cout << "Private key:\n" << rsaKeys.PrivateKey.first << "\n" << rsaKeys.PrivateKey.second;

std::cout << std::endl;

}

}

namespace {

// Miller–Rabin primality test

bool IsPrime(uint1024 value) {

constexpr static std::array<uint32\_t, 10> primes = {3, 5, 7, 11, 13, 17, 19, 23, 29, 31};

const static uint1024 temp{1};

for (auto p : primes) {

if (value % p == 0) {

return false;

}

}

const uint1024 base = 2;

--value;

uint32\_t power = 0;

while((value & (temp << power)) == 0) {

++power;

}

++value;

uint1024 q{value / (temp << power)};

auto surplus = BinPower(base, q, value);

if (surplus == 1 || surplus == value - 1) {

return true;

}

for (uint32\_t i = 1; i < power; ++i) {

surplus = (surplus \* surplus) % value;

if (surplus == value - 1) {

return true;

}

}

return false;

}

uint1024 GetRandBits(int bits, std::default\_random\_engine\* engine) {

std::uniform\_int\_distribution<int> uniform\_dist(0, 1);

uint1024 value = 0;

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

if ((uniform\_dist(\*engine) & 1) == 0) {

value |= (uint1024{1} << i);

}

}

return value;

}

uint1024 FPrime(int32\_t bits) {

std::random\_device r;

std::default\_random\_engine engine(r());

uint1024 e = BinPower(2, 16, 1000000000) + 1;

uint1024 mask = (uint1024{3} << (bits - 2)) | 1;

int count = 0;

while (true) {

++count;

auto value = GetRandBits(bits, &engine) | mask;

if (value % e != 1 && IsPrime(value)) {

return value;

}

}

}

}

uint1024 GeneratePrime(uint32\_t bitsCount) {

return FPrime(bitsCount);

}

uint1024 TransformFromBits(const std::string& bits, int bitsCount) {

if (bitsCount != bits.length()) {

std::cerr << "Warning. Incorrect sizes\n";

}

uint1024 value = 0;

for (int i = 0, j = bits.length() - 1; j >= 0 && i < bitsCount; --j, ++i) {

if (bits[j] == '0') {

continue;

}

value += uint1024{1} << i;

}

return value;

}

std::string TransformToBits(const std::string& value\_p, int bitsCount) {

uint1024 value{value\_p};

std::string bits;

for (int i = bitsCount - 1; i >= 0; --i) {

if (value & (uint1024{1} << i)) {

bits += '1';

} else {

bits += '0';

}

}

return bits;

}

namespace {

std::pair<uint1024, uint1024> ExtendedEuclidian(const uint1024& e, const uint1024& phi) {

if (e == 0) {

return std::make\_pair(0, 1);

}

auto tmp = ExtendedEuclidian(phi % e, e);

auto x1 = tmp.first;

auto y1 = tmp.second;

return std::make\_pair(y1 - (phi / e) \* x1 ,x1);

}

}

TRSAKeys GenerateRSAKeys(const uint1024& p, const uint1024& q, int bitsCount) {

uint1024 moduleValue = p \* q;

uint1024 phi = (p - 1) \* (q - 1);

uint1024 openExp;

uint1024 privateExp = -1;

while (privateExp < 0) {

openExp = GeneratePrime(std::min(bitsCount, 32));

auto temp = ExtendedEuclidian(openExp, phi);

auto privateExpValue = temp.first;

privateExp = std::move(privateExpValue);

}

if ((openExp \* privateExp) % phi != 1) {

std::cerr << "Warning. Incorrect exps\n";

}

return TRSAKeys{openExp, privateExp, moduleValue};

}

uint1024 Encode(uint1024 value, const std::pair<uint1024, uint1024>& publicKey) {

if (value >= publicKey.second) {

std::cerr << "Block is too large.\n";

}

return BinPower(value, publicKey.first, publicKey.second);

}

uint1024 Decode(uint1024 value, const std::pair<uint1024, uint1024>& privateKey) {

return BinPower(value, privateKey.first, privateKey.second);

}

int main() {

constexpr int bitsCount = 1024;

auto p = GeneratePrime(bitsCount);

auto q = GeneratePrime(bitsCount);

auto rsaKeys = GenerateRSAKeys(p, q, bitsCount);

PrintKeys(rsaKeys);

uint1024 value{"1230948092384098"};

auto encrypted = Encode(value, rsaKeys.PublicKey);

auto decrypted = Decode(encrypted, rsaKeys.PrivateKey);

std::cout << "\nMessage: " << value << std::endl;

std::cout << "Encrypted: " << encrypted << std::endl;

std::cout << "Decrypted: " << decrypted << std::endl;

return 0;

}