Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №4

По дисциплине «Методы защиты информации»

По теме «Ассиметричная криптография. Алгоритм Эль-Гамаля»

Выполнил:

Студент гр. 653501

Никитинская А. С.

Проверил:

Артемьев В. С.

Минск 2019

Содержание

[1. Постановка задачи 3](#_Toc26718068)

[2. Краткие теоретические сведения 4](#_Toc26718069)

[3. Блок-схемы алгоритма 6](#_Toc26718070)

[5. Вывод 7](#_Toc26718071)

[Приложение 1. Исходный код программы 8](#_Toc26718072)

[Приложение 2. Скриншот работы программы 13](#_Toc26718073)

## 1. Постановка задачи

1) Изучить теоретические сведения.

2) Реализовать программные средства шифрования и дешифрования текстовых файлов при помощи алгоритма Эль-Гамаля.

## Краткие теоретические сведения

Схема Эль-Гамаля (Elgamal) — криптосистема с открытым ключом, основанная на трудности вычисления дискретных логарифмов в конечном поле. Криптосистема включает в себя алгоритм шифрования и алгоритм цифровой подписи. Схема Эль-Гамаля лежит в основе бывших стандартов электронной цифровой подписи в США (DSA) и России (ГОСТ Р 34.10-94).

Схема была предложена Тахером Эль-Гамалем в 1985 году. Эль-Гамаль разработал один из вариантов алгоритма Диффи-Хеллмана. Он усовершенствовал систему Диффи-Хеллмана и получил два алгоритма, которые использовались для шифрования и для обеспечения аутентификации. В отличие от RSA алгоритм Эль-Гамаля не был запатентован и, поэтому, стал более дешевой альтернативой, так как не требовалась оплата взносов за лицензию. Считается, что алгоритм попадает под действие патента Диффи-Хеллмана.

Генерация ключей:

1. Генерируется случайное простое число p.
2. Выбирается целое число g — первообразный корень p.
3. Выбирается случайное целое число x такое, что 1<x<p-1.
4. Вычисляется y = gx mod p.
5. Открытым ключом является y, закрытым ключом — число x.

Шифросистема Эль-Гамаля является фактически одним из способов выработки открытых ключей Диффи — Хеллмана. Шифрование по схеме Эль-Гамаля не следует путать с алгоритмом цифровой подписи по схеме Эль-Гамаля.

Шифрование

Сообщение M должно быть меньше числа p. Сообщение шифруется следующим образом:

1. Выбирается сессионный ключ — случайное целое число k такое, что 1<k<p-1.
2. Вычисляются числа a=gk mod p и b=yk \* M mod p.
3. Пара чисел (a,b) является шифротекстом.

Нетрудно увидеть, что длина шифротекста в схеме Эль-Гамаля длиннее исходного сообщения M вдвое.

Расшифрование

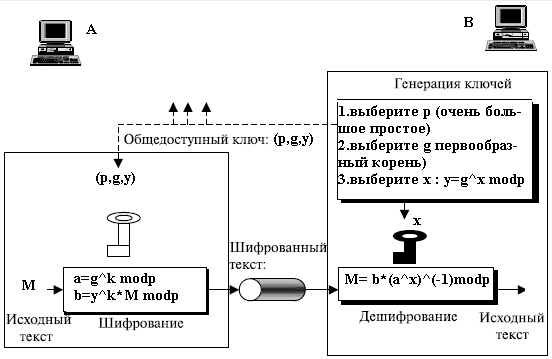
Зная закрытый ключ x, исходное сообщение можно вычислить из шифротекста (a,b) по формуле: M=b\*(ax)-1 mod p

При этом нетрудно проверить, что (ax)-1 = g –kx mod p

и поэтому b\*(ax)-1 = (yk\*M) \* g –kx = (gxk \* M) \* g –kx = M (mod p).

Для практических вычислений больше подходит следующая формула: M=b (ax)-1 = ba p-1-x (mod p).

## 3. Блок-схемы алгоритма



## 5. Вывод

Алгоритм Эль-Гамаля может использоваться для формирования электронной подписи или для шифрования данных. Он базируется на трудности вычисления дискретного логарифма. В настоящее время криптосистемы с открытым ключом считаются наиболее перспективными. К ним относится и схема Эль-Гамаля, криптостойкость которой основана на вычислительной сложности проблемы дискретного логарифмирования.

## Приложение 1. Исходный код программы

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

#include <chrono>

#include <ratio>

#include <thread>

#include <atomic>

#include <vector>

#include <array>

#include <random>

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

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

struct Keys {

uint1024 PublicKey;

uint1024 PrivateKey;

uint1024 Root;

uint1024 Prime;

template<class T>

Keys(T&& publicKey, T&& privateKey, T&& root, T&& prime)

: PublicKey(std::forward<T>(publicKey))

, PrivateKey(std::forward<T>(privateKey))

, Root(std::forward<T>(root))

, Prime(std::forward<T>(prime))

{}

};

struct Text {

uint1024 KeyPart;

uint1024 TextPart;

template<class T>

Text(T&& keyPart, T&& textPart)

: KeyPart(std::forward<T>(keyPart))

, TextPart(std::forward<T>(textPart))

{}

};

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 {

// 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;

}

int GetBitsCount(uint1024 value) {

int bitsCount = 0;

while (value != 0) {

value >>= 1;

++bitsCount;

}

return bitsCount;

}

namespace {

auto GeneratePrimes() {

constexpr int count = 7500000;

std::array<bool, count> used{};

for (auto& i : used) {

i = false;

}

for (int i = 3; i < count; i += 2) {

if (used[i] == true) {

continue;

}

uint64\_t j = i + i;

while (j < count) {

used[j] = true;

j += i;

}

}

std::vector<uint64\_t> primes{{2}};

primes.reserve(1000);

for (int i = 3; i < count; i += 2) {

if (used[i] == false) {

primes.push\_back(i);

}

}

return primes;

}

uint1024 PrimitiveRootModulo(const uint1024& prime) {

auto primes = GeneratePrimes();

uint1024 phi{prime - 1};

std::vector<uint1024> factorization;

uint1024 n{phi};

for (int i = 0; (primes[i] \* primes[i]) <= n && i < primes.size(); ++i) {

if (n % primes[i] == 0) {

factorization.push\_back(primes[i]);

while (n % primes[i] == 0) {

n /= primes[i];

}

}

}

if (n > 1) {

factorization.push\_back(n);

}

for (uint64\_t result = 2; result <= prime; ++result) {

bool ok = true;

for (const auto& div : factorization) {

ok &= BinPower(result, phi / div, prime) != 1;

if (ok == false) {

break;

}

}

if (ok == true) {

return result;

}

}

return -1;

}

}

Keys GenerateElGamalKeys(int bitsCount) {

uint1024 prime{}, root{-1};

for (int i = 0; i < 1000; ++i) {

prime = GeneratePrime(bitsCount);

root = PrimitiveRootModulo(prime);

if (root != -1 && BinPower(root, prime - 1, prime) == 1) {

break;

}

}

uint1024 privateKey = GeneratePrime(bitsCount - 1);

uint1024 publicKey = BinPower(root, privateKey, prime);

return Keys{publicKey, privateKey, root, prime};

}

Text Encode(const uint1024& sourceText, const uint1024& publicKey, const uint1024& root, const uint1024& prime) {

int bitsCount = GetBitsCount(publicKey);

uint1024 sessionKey = GeneratePrime(bitsCount - 1);

uint1024 keyPart = BinPower(root, sessionKey, prime);

uint1024 textPart = (BinPower(publicKey, sessionKey, prime) \* sourceText) % prime;

return Text{std::move(keyPart), std::move(textPart)};

}

uint1024 Decode(const Text& text, const uint1024& privateKey, const uint1024& root, const uint1024& prime) {

uint1024 reverse = BinPower(text.KeyPart, prime - 1 - privateKey, prime);

uint1024 message = (text.TextPart \* reverse) % prime;

return message;

}

int main() {

Keys keys = GenerateElGamalKeys(1024);

std::cout << "Public key: " << keys.PublicKey << "\nPrivate key: " << keys.PrivateKey << std::endl;

std::cout << "Root: " << keys.Root << "\nPrime: " << keys.Prime << std::endl << std::endl;

uint1024 value{"12039832589634576987123"};

auto encoded = Encode(value, keys.PublicKey, keys.Root, keys.Prime);

auto decoded = Decode(encoded, keys.PrivateKey, keys.Root, keys.Prime);

std::cout << "Message: " << value << std::endl;

std::cout << "Encrypted a: " << encoded.KeyPart << "\n b: " << encoded.TextPart << std::endl;

std::cout << "Decoded: " << decoded << std::endl;

return 0;

}

## Приложение 2. Скриншот работы программы

