БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра информатики

Факультет ИНО

Специальность ИиТП

Контрольная работа № 2

по дисциплине «Методы защиты информации»

Выполнил студент: Дегтярев А.А.

группа 393551

Зачетная книжка № 902021-26

Минск 2018

**КР №2**

#### Криптография с использованием эллиптических кривых.

В данной работе, я выполнил генерацию публичных/приватных ключей, для шифрования текста аналогично алгоритму Диффи-Хеллмана

Порядок использования программы:



Необходимо указать флаг –g для генерации ключей

**Использовать флаг ECC(ecc) для указания алгоритма**

Алгоритм использует эллиптические кривые

Для использования алгоритма в коде представлено API

**int** ecdh\_generate\_keys(uint8\_t\* publicKey, uint8\_t\* privateKey);  
**int** ecdh\_shared\_secret(**const** uint8\_t\* privateKey, **const** uint8\_t\* otherPublicKey, uint8\_t\* sharedSecret);

Пример использования:



Алгоритм генерирует два ключа для двух собеседников аналогично схеме Диффи-Хельмана, публичный/приватный, далее выполняется проверка переданного секретного сообщения. Они совпадают у обоих собеседников.

Исходные файлы:

ECC.h

#ifndef MZI\_ECC\_H  
#define MZI\_ECC\_H  
#include <cstdint>  
#include "bitvector.h"  
#include "eccfield.h"  
**int** ecdh\_generate\_keys(uint8\_t\* publicKey, uint8\_t\* privateKey);  
**int** ecdh\_shared\_secret(**const** uint8\_t\* privateKey, **const** uint8\_t\* otherPublicKey, uint8\_t\* sharedSecret);  
#endif //MZI\_ECC\_H

ECC.cpp

#include "ecc.h"  
**int** ecdh\_generate\_keys(uint8\_t\* publicKey, uint8\_t\* privateKey)  
{  
 //private should contain random data!  
 gf2point\_copy((uint32\_t\*)publicKey, (uint32\_t\*)(publicKey + BITVEC\_NBYTES), base\_x, base\_y);  
  
 **if** (bitvec\_degree((uint32\_t\*)privateKey) < (CURVE\_DEGREE / 2))  
 {  
 printf("\nERROR: Random is too small\n");  
 **return** 0;  
 }  
 **else** {  
 **int** nbits = bitvec\_degree(base\_order);  
 **int** i;  
  
 **for** (i = (nbits - 1); i < (BITVEC\_NWORDS \* 32); ++i)  
 {  
 /\* Clear bits > CURVE\_DEGREE in highest word to satisfy constraint 1 <= exp < n. \*/  
 bitvec\_clear\_bit((uint32\_t\*)privateKey, i);  
 }  
 gf2point\_mul((uint32\_t\*)publicKey, (uint32\_t\*)(publicKey + BITVEC\_NBYTES), (uint32\_t\*)privateKey);  
 **return** 1;  
 }  
}  
  
**int** ecdh\_shared\_secret(**const** uint8\_t\* privateKey, **const** uint8\_t\* otherPublicKey, uint8\_t\* sharedSecret)  
{  
 **if** (!gf2point\_is\_zero ((uint32\_t\*)otherPublicKey, (uint32\_t\*)(otherPublicKey + BITVEC\_NBYTES)) && gf2point\_on\_curve((uint32\_t\*)otherPublicKey, (uint32\_t\*)(otherPublicKey + BITVEC\_NBYTES)) )  
 {  
 **unsigned int** i;  
 **for** (i = 0; i < (BITVEC\_NBYTES \* 2); ++i)  
 {  
 sharedSecret[i] = otherPublicKey[i];  
 }  
 gf2point\_mul((uint32\_t\*)sharedSecret,(uint32\_t\*)(sharedSecret + BITVEC\_NBYTES), (**const** uint32\_t\*)privateKey); //otherPub\*ourKey  
 **return** 1;  
 }  
 **else** {  
 **return** 0;  
 }  
}

ECCFIELD.h

//  
// Created by xdegtyarev on 1/16/18.  
//  
  
#ifndef MZI\_ECCFIELD\_H  
#define MZI\_ECCFIELD\_H  
#include <cstdint>  
#include "utility.h"  
  
#define ECC\_PRV\_KEY\_SIZE 21  
**typedef** bitvec\_t gf2elem\_t;//field element type  
**typedef** bitvec\_t scalar\_t;//scalar element type  
  
  
#define NIST\_B163 1  
#define NIST\_K163 2  
  
#ifndef ECC\_CURVE  
#define ECC\_CURVE NIST\_B163  
#endif  
  
#if defined (ECC\_CURVE) && (ECC\_CURVE != 0)  
  
#if (ECC\_CURVE == NIST\_B163)  
#define coeff\_a 1  
#define cofactor 2  
**const** gf2elem\_t polynomial = { 0x000000c9, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000008 };  
**const** gf2elem\_t coeff\_b = { 0x4a3205fd, 0x512f7874, 0x1481eb10, 0xb8c953ca, 0x0a601907, 0x00000002 };  
**const** gf2elem\_t base\_x = { 0xe8343e36, 0xd4994637, 0xa0991168, 0x86a2d57e, 0xf0eba162, 0x00000003 };  
**const** gf2elem\_t base\_y = { 0x797324f1, 0xb11c5c0c, 0xa2cdd545, 0x71a0094f, 0xd51fbc6c, 0x00000000 };  
**const** scalar\_t base\_order = { 0xa4234c33, 0x77e70c12, 0x000292fe, 0x00000000, 0x00000000, 0x00000004 };  
#endif  
  
#if (ECC\_CURVE == NIST\_K163)  
#define coeff\_a 1  
 #define cofactor 2  
 **const** gf2elem\_t polynomial = { 0x000000c9, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000008 };  
 **const** gf2elem\_t coeff\_b = { 0x00000001, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000 };  
 **const** gf2elem\_t base\_x = { 0x5c94eee8, 0xde4e6d5e, 0xaa07d793, 0x7bbc11ac, 0xfe13c053, 0x00000002 };  
 **const** gf2elem\_t base\_y = { 0xccdaa3d9, 0x0536d538, 0x321f2e80, 0x5d38ff58, 0x89070fb0, 0x00000002 };  
 **const** scalar\_t base\_order = { 0x99f8a5ef, 0xa2e0cc0d, 0x00020108, 0x00000000, 0x00000000, 0x00000004 };  
#endif  
#endif  
  
**static void** gf2field\_set\_one(gf2elem\_t x)  
{  
 /\* Set first word to one \*/  
 x[0] = 1;  
 /\* .. and the rest to zero \*/  
 **int** i;  
 **for** (i = 1; i < BITVEC\_NWORDS; ++i)  
 {  
 x[i] = 0;  
 }  
}  
  
**static int** gf2field\_is\_one(**const** gf2elem\_t x)  
{  
 /\* Check if first word == 1 \*/  
 **if** (x[0] != 1)  
 {  
 **return** 0;  
 }  
 /\* ...and if rest of words == 0 \*/  
 **int** i;  
 **for** (i = 1; i < BITVEC\_NWORDS; ++i)  
 {  
 **if** (x[i] != 0)  
 {  
 **break**;  
 }  
 }  
 **return** (i == BITVEC\_NWORDS);  
}  
  
/\* galois field(2^m) addition is modulo 2, so XOR is used instead - 'z := a + b' \*/  
**static void** gf2field\_add(gf2elem\_t z, **const** gf2elem\_t x, **const** gf2elem\_t y)  
{  
 **int** i;  
 **for** (i = 0; i < BITVEC\_NWORDS; ++i)  
 {  
 z[i] = (x[i] ^ y[i]);  
 }  
}  
  
/\* increment element \*/  
**static void** gf2field\_inc(gf2elem\_t x)  
{  
 x[0] ^= 1;  
}  
  
  
/\* field multiplication 'z := (x \* y)' \*/  
**static void** gf2field\_mul(gf2elem\_t z, **const** gf2elem\_t x, **const** gf2elem\_t y)  
{  
 **int** i;  
 gf2elem\_t tmp;  
  
// *TODO: assert(z != y);* bitvec\_copy(tmp, x);  
  
 /\* LSB set? Then start with x \*/  
 **if** (bitvec\_get\_bit(y, 0) != 0)  
 {  
 bitvec\_copy(z, x);  
 }  
 **else** /\* .. or else start with zero \*/  
 {  
 bitvec\_set\_zero(z);  
 }  
  
 /\* Then add 2^i \* x for the rest \*/  
 **for** (i = 1; i < CURVE\_DEGREE; ++i)  
 {  
 /\* lshift 1 - doubling the value of tmp \*/  
 bitvec\_lshift(tmp, tmp, 1);  
  
 /\* Modulo reduction polynomial if degree(tmp) > CURVE\_DEGREE \*/  
 **if** (bitvec\_get\_bit(tmp, CURVE\_DEGREE))  
 {  
 gf2field\_add(tmp, tmp, polynomial);  
 }  
  
 /\* Add 2^i \* tmp if this factor in y is non-zero \*/  
 **if** (bitvec\_get\_bit(y, i))  
 {  
 gf2field\_add(z, z, tmp);  
 }  
 }  
}  
  
/\* field inversion 'z := 1/x' \*/  
**static void** gf2field\_inv(gf2elem\_t z, **const** gf2elem\_t x)  
{  
 gf2elem\_t u, v, g, h;  
 **int** i;  
  
 bitvec\_copy(u, x);  
 bitvec\_copy(v, polynomial);  
 bitvec\_set\_zero(g);  
 gf2field\_set\_one(z);  
  
 **while** (!gf2field\_is\_one(u))  
 {  
 i = (bitvec\_degree(u) - bitvec\_degree(v));  
  
 **if** (i < 0)  
 {  
 bitvec\_swap(u, v);  
 bitvec\_swap(g, z);  
 i = -i;  
 }  
  
 bitvec\_lshift(h, v, i);  
 gf2field\_add(u, u, h);  
 bitvec\_lshift(h, g, i);  
 gf2field\_add(z, z, h);  
 }  
}  
  
  
**static void** gf2point\_copy(gf2elem\_t x1, gf2elem\_t y1, **const** gf2elem\_t x2, **const** gf2elem\_t y2)  
{  
 bitvec\_copy(x1, x2);  
 bitvec\_copy(y1, y2);  
}  
  
**static void** gf2point\_set\_zero(gf2elem\_t x, gf2elem\_t y)  
{  
 bitvec\_set\_zero(x);  
 bitvec\_set\_zero(y);  
}  
  
**static int** gf2point\_is\_zero(**const** gf2elem\_t x, **const** gf2elem\_t y)  
{  
 **return** ( bitvec\_is\_zero(x)  
 && bitvec\_is\_zero(y));  
}  
  
/\* double the point (x,y) \*/  
**static void** gf2point\_double(gf2elem\_t x, gf2elem\_t y)  
{  
 /\* iff P = O (zero or infinity): 2 \* P = P \*/  
 **if** (bitvec\_is\_zero(x))  
 {  
 bitvec\_set\_zero(y);  
 }  
 **else** {  
 gf2elem\_t l;  
  
 gf2field\_inv(l, x);  
 gf2field\_mul(l, l, y);  
 gf2field\_add(l, l, x);  
 gf2field\_mul(y, x, x);  
 gf2field\_mul(x, l, l);  
  
 gf2field\_inc(l);  
 gf2field\_add(x, x, l);  
 gf2field\_mul(l, l, x);  
 gf2field\_add(y, y, l);  
 }  
}  
  
  
/\* add two points together (x1, y1) := (x1, y1) + (x2, y2) \*/  
**static void** gf2point\_add(gf2elem\_t x1, gf2elem\_t y1, **const** gf2elem\_t x2, **const** gf2elem\_t y2)  
{  
 **if** (!gf2point\_is\_zero(x2, y2))  
 {  
 **if** (gf2point\_is\_zero(x1, y1))  
 {  
 gf2point\_copy(x1, y1, x2, y2);  
 }  
 **else** {  
 **if** (bitvec\_equals(x1, x2))  
 {  
 **if** (bitvec\_equals(y1, y2))  
 {  
 gf2point\_double(x1, y1);  
 }  
 **else** {  
 gf2point\_set\_zero(x1, y1);  
 }  
 }  
 **else** {  
 /\* Arithmetic with temporary variables \*/  
 gf2elem\_t a, b, c, d;  
  
 gf2field\_add(a, y1, y2);  
 gf2field\_add(b, x1, x2);  
 gf2field\_inv(c, b);  
 gf2field\_mul(c, c, a);  
 gf2field\_mul(d, c, c);  
 gf2field\_add(d, d, c);  
 gf2field\_add(d, d, b);  
  
 gf2field\_inc(d);  
 gf2field\_add(x1, x1, d);  
 gf2field\_mul(a, x1, c);  
 gf2field\_add(a, a, d);  
 gf2field\_add(y1, y1, a);  
 bitvec\_copy(x1, d);  
 }  
 }  
 }  
}  
  
/\* point multiplication via double-and-add algorithm \*/  
**static void** gf2point\_mul(gf2elem\_t x, gf2elem\_t y, **const** scalar\_t exp)  
{  
 gf2elem\_t tmpx, tmpy;  
 **int** i;  
 **int** nbits = bitvec\_degree(exp);  
  
 gf2point\_set\_zero(tmpx, tmpy);  
  
 **for** (i = (nbits - 1); i >= 0; --i)  
 {  
 gf2point\_double(tmpx, tmpy);  
 **if** (bitvec\_get\_bit(exp, i))  
 {  
 gf2point\_add(tmpx, tmpy, x, y);  
 }  
 }  
 gf2point\_copy(x, y, tmpx, tmpy);  
}  
  
  
**static int** gf2point\_on\_curve(**const** gf2elem\_t x, **const** gf2elem\_t y)  
{  
 gf2elem\_t a, b;  
  
 **if** (gf2point\_is\_zero(x, y))  
 {  
 **return** 1;  
 }  
 **else** {  
 gf2field\_mul(a, x, x);  
 gf2field\_mul(b, a, x);  
 gf2field\_add(a, a, b);  
 gf2field\_add(a, a, coeff\_b);  
 gf2field\_mul(b, y, y);  
 gf2field\_add(a, a, b);  
 gf2field\_mul(b, x, y);  
 **return** bitvec\_equals(a, b);  
 }  
}  
#endif //MZI\_ECCFIELD\_H

BITVEC.h  
#ifndef MZI\_BITVECTOR\_H  
#define MZI\_BITVECTOR\_H  
  
#include <cstdint>  
  
#define CURVE\_DEGREE 163  
#define BITVEC\_MARGIN 3 // margin for overhead needed in intermediate calculations  
#define BITVEC\_NBITS (CURVE\_DEGREE + BITVEC\_MARGIN)  
#define BITVEC\_NWORDS ((BITVEC\_NBITS + 31) / 32)  
#define BITVEC\_NBYTES (**sizeof**(uint32\_t) \* BITVEC\_NWORDS)  
  
#define ECC\_PRV\_KEY\_SIZE 21  
#define ECC\_PUB\_KEY\_SIZE (2 \* ECC\_PRV\_KEY\_SIZE)  
  
**typedef** uint32\_t bitvec\_t[BITVEC\_NWORDS];  
  
**static int** bitvec\_get\_bit(**const** bitvec\_t x, **const** uint32\_t idx)  
{  
 **return** ((x[idx / 32] >> (idx & 31) & 1));  
}  
  
**static void** bitvec\_clear\_bit(bitvec\_t x, **const** uint32\_t idx)  
{  
 x[idx / 32] &= ~(1 << (idx & 31));  
}  
  
**static void** bitvec\_copy(bitvec\_t x, **const** bitvec\_t y)  
{  
 **int** i;  
 **for** (i = 0; i < BITVEC\_NWORDS; ++i)  
 {  
 x[i] = y[i];  
 }  
}  
  
**static void** bitvec\_swap(bitvec\_t x, bitvec\_t y)  
{  
 bitvec\_t tmp;  
 bitvec\_copy(tmp, x);  
 bitvec\_copy(x, y);  
 bitvec\_copy(y, tmp);  
}  
  
**static int** bitvec\_equals(**const** bitvec\_t x, **const** bitvec\_t y)  
{  
 **int** i;  
 **for** (i = 0; i < BITVEC\_NWORDS; ++i)  
 {  
 **if** (x[i] != y[i])  
 {  
 **return** 0;  
 }  
 }  
 **return** 1;  
}  
  
**static void** bitvec\_set\_zero(bitvec\_t x)  
{  
 **int** i;  
 **for** (i = 0; i < BITVEC\_NWORDS; ++i)  
 {  
 x[i] = 0;  
 }  
}  
  
**static int** bitvec\_is\_zero(**const** bitvec\_t x)  
{  
 uint32\_t i = 0;  
 **while** (i < BITVEC\_NWORDS)  
 {  
 **if** (x[i] != 0)  
 {  
 **break**;  
 }  
 i += 1;  
 }  
 **return** (i == BITVEC\_NWORDS);  
}  
  
/\* return the number of the highest one-bit + 1 \*/  
**static int** bitvec\_degree(**const** bitvec\_t x)  
{  
 **int** i = BITVEC\_NWORDS \* 32;  
  
 /\* Start at the back of the vector (MSB) \*/  
 x += BITVEC\_NWORDS;  
  
 /\* Skip empty / zero words \*/  
 **while** ( (i > 0)  
 && (\*(--x)) == 0)  
 {  
 i -= 32;  
 }  
 /\* Run through rest if count is not multiple of bitsize of DTYPE \*/  
 **if** (i != 0)  
 {  
 uint32\_t u32mask = ((uint32\_t)1 << 31);  
 **while** (((\*x) & u32mask) == 0)  
 {  
 u32mask >>= 1;  
 i -= 1;  
 }  
 }  
 **return** i;  
}  
  
/\* left-shift by 'count' digits \*/  
**static void** bitvec\_lshift(bitvec\_t x, **const** bitvec\_t y, **int** nbits)  
{  
 **int** nwords = (nbits / 32);  
  
 /\* Shift whole words first if nwords > 0 \*/  
 **int** i,j;  
 **for** (i = 0; i < nwords; ++i)  
 {  
 /\* Zero-initialize from least-significant word until offset reached \*/  
 x[i] = 0;  
 }  
 j = 0;  
 /\* Copy to x output \*/  
 **while** (i < BITVEC\_NWORDS)  
 {  
 x[i] = y[j];  
 i += 1;  
 j += 1;  
 }  
  
 /\* Shift the rest if count was not multiple of bitsize of DTYPE \*/  
 nbits &= 31;  
 **if** (nbits != 0)  
 {  
 /\* Left shift rest \*/  
 **int** i;  
 **for** (i = (BITVEC\_NWORDS - 1); i > 0; --i)  
 {  
 x[i] = (x[i] << nbits) | (x[i - 1] >> (32 - nbits));  
 }  
 x[0] <<= nbits;  
 }  
}  
  
#endif //MZI\_BITVECTOR\_H

Фрагмент файла main.cpp для использования функции  
}**else if**(config.type == *ECC*){  
 printf("\nECC Generate Keys\n");  
 **static** uint8\_t puba[ECC\_PUB\_KEY\_SIZE];  
 **static** uint8\_t prva[ECC\_PRV\_KEY\_SIZE];  
 **static** uint8\_t seca[ECC\_PUB\_KEY\_SIZE];  
 **static** uint8\_t pubb[ECC\_PUB\_KEY\_SIZE];  
 **static** uint8\_t prvb[ECC\_PRV\_KEY\_SIZE];  
 **static** uint8\_t secb[ECC\_PUB\_KEY\_SIZE];  
 uint32\_t i;  
  
 **static int** initialized = 0;  
 **if** (!initialized)  
 {  
 prng\_init((0xbad ^ 0xc0ffee ^ 42) | 0xcafebabe | 666);  
 initialized = 1;  
 }  
  
 /\* 1.Alice picks a (secret) random natural number 'a', calculates P = a \* g and sends P to Bob. \*/  
 printf("\nAlice rnd prvkeys: \n");  
 **for** (i = 0; i < ECC\_PRV\_KEY\_SIZE; ++i)  
 {  
 prva[i] = prng\_next();  
 printf("%2.2x",prva[i]);  
 }  
 printf("\n\n");  
 **if**(ecdh\_generate\_keys(puba, prva)){  
 printf("\nAlice keys generated: \n");  
 printf("\nPUBLIC\n");  
 **for**(**int** i = 0; i<ECC\_PUB\_KEY\_SIZE; i++){  
 printf("%2.2x",puba[i]);  
 }  
 printf("\nPRIVATE\n");  
 **for**(**int** i = 0; i<ECC\_PRV\_KEY\_SIZE; i++){  
 printf("%2.2x",prva[i]);  
 }  
 }  
 printf("\n\n");  
 /\* 2. Bob picks a (secret) random natural number 'b', calculates Q = b \* g and sends Q to Alice. \*/  
 printf("\nBOB rnd prvkeys: \n");  
 **for** (i = 0; i < ECC\_PRV\_KEY\_SIZE; ++i)  
 {  
 prvb[i] = prng\_next();  
 printf("%2.2x",prvb[i]);  
 }  
 printf("\n\n");  
 **if**(ecdh\_generate\_keys(pubb, prvb)){  
 printf("\nBob keys generated: \n");  
 printf("\nPUBLIC\n");  
 **for**(**int** i = 0; i<ECC\_PUB\_KEY\_SIZE; i++){  
 printf("%2.2x",pubb[i]);  
 }  
 printf("\nPRIVATE\n");  
 **for**(**int** i = 0; i<ECC\_PRV\_KEY\_SIZE; i++){  
 printf("%2.2x",prvb[i]);  
 }  
 }  
 printf("\n\n");  
 **if**(ecdh\_shared\_secret(prva, pubb, seca)){  
 /\*S = a \* Q = a \* (b \* g). \*/  
 printf("\nAlice sharedSecret:\n");  
 **for**(**int** i = 0; i<ECC\_PRV\_KEY\_SIZE; i++){  
 printf("%2.2x",seca[i]);  
 }  
 }  
  
 printf("\n\n");  
  
 **if**(ecdh\_shared\_secret(prvb, puba, secb)){  
 /\*T = b \* P = b \* (a \* g). \*/  
 printf("\nBob sharedSecret:\n");  
 **for**(**int** i = 0; i<ECC\_PRV\_KEY\_SIZE; i++){  
 printf("%2.2x",secb[i]);  
 }  
 }  
  
 printf("\n\n");  
  
 **for** (i = 0; i < ECC\_PUB\_KEY\_SIZE; ++i)  
 {  
 assert(seca[i] == secb[i]);  
 }  
 printf("Success: both recepients calculated same value");  
}