Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Лабораторная работа № 7

Криптография с использованием эллиптических кривых

Выполнил: Нетецкая Ю.В.

Проверил: Олисейчик В.В.

Минск 2021

**Постановка задачи**

Реализовать программное средство формирования электронной цифровой подписи на основе эллиптических кривых ECDSA и программное средство и программное средство, реализующее простой подход к шифрованию/дешифрованию с использованием *эллиптических кривых*.

**Описание использованных алгоритмов**

**Аналог алгоритма Диффи-Хеллмана обмена ключами**

Обмен ключами с использованием *эллиптических кривых* может быть выполнен следующим образом. Сначала выбирается простое число р ≈ 2180 и параметры a и b для уравнения *эллиптической кривой*. Это задает множество точек Ep (a,b). Затем в Ep (a,b) выбирается генерирующая точка G = (x1,y1). При выборе G важно, чтобы наименьшее значение n, при котором n × G = 0, оказалось очень большим простым числом. Параметры Ep (a,b) и G криптосистемы являются параметрами, известными всем участникам.

Обмен ключами между пользователями А и В производится по следующей схеме.

1.           Участник А выбирает целое число nA, меньшее n. Это число является закрытым ключом участника А. Затем участник А вычисляет открытый ключ PA = nA × G, который представляет собой некоторую точку на Ep (a,b).

2.           Точно так же участник В выбирает закрытый ключ nB и вычисляет открытый ключ PB.

3.           Участники обмениваются открытыми ключами, после чего вычисляют общий секретный ключ K

Участник А: K = nA × PB

       Участник В: K = nВ × PА

Следует заметить, что общий секретный ключ представляет собой пару чисел. Если данный ключ предполагается использовать в качестве сеансового ключа для алгоритма симметричного шифрования, то из этой пары необходимо создать одно значение.

**Алгоритм цифровой подписи на основе эллиптических кривых ECDSA**

Алгоритм ECDSA (Elliptic Curve Digest Signature Algorithm) принят в качестве стандартов ANSI X9F1 и IEEE P1363.

Создание ключей:

1.           Выбирается *эллиптическая кривая*   Ep (a,b). Число точек на ней должно делиться на большое целое n.

2.           Выбирается точка РEp (a,b).

3.           Выбирается случайное число d  [1, n-1].

4.           Вычисляется Q = d × P.

5.           Закрытым ключом является d, открытым ключом – (E, P, n, Q).

Создание подписи:

1.           Выбирается случайное число k [1, n-1].

2.           Вычисляется k × P = (x1, y1)   и  r = x1 (mod n).

Проверяется, чтобы r не было равно нулю, так как в этом случае подпись не будет зависеть от закрытого ключа. Если r = 0, то выбирается другое случайное число k.

3.           Вычисляется k-1 mod n

4.           Вычисляется s = k-1 (Н(M) + dr) (mod n)

Проверяется, чтобы s не было равно нулю, так как в этом случае необходимого для проверки подписи числа s-1 mod n не существует. Если s = 0, то выбирается другое случайное число k.

Подписью для сообщения М является пара чисел (r,s).

Проверка подписи:

1.           Проверить, что целые числа r и s принадлежат диапазону чисел [0, n-1]. В противном случае результат проверки отрицательный, и подпись отвергается.

2.           Вычислить w = s-1 (mod n) и H(M)

3.           Вычислить u1 = H(M) w (mod n), u2 = rw (mod n)

4.           Вычислить  u1P + u2Q = (x0, y0), v = x0 (mod n)

5.           Подпись верна в том и только том случае, когда v = r.

**Шифрование/дешифрование с использованием эллиптических кривых**

Рассмотрим самый простой подход к шифрованию/дешифрованию с использованием *эллиптических кривых*. Задача состоит в том, чтобы зашифровать сообщение М, которое может быть представлено в виде точки на эллиптической кривой Pm (x,y).

Как и в случае обмена ключом, в системе шифрования/дешифрования в качестве параметров рассматривается *эллиптическая кривая*   Ep (a,b) и точка G на ней. Участник B выбирает закрытый ключ nB и вычисляет открытый ключ PB = nB × G. Чтобы зашифровать сообщение Pm используется открытый ключ получателя B   PB. Участник А выбирает случайное целое положительное число k и вычисляет зашифрованное сообщение Cm, являющееся точкой на *эллиптической кривой*.

Cm = {k × G, Pm + k × PB}

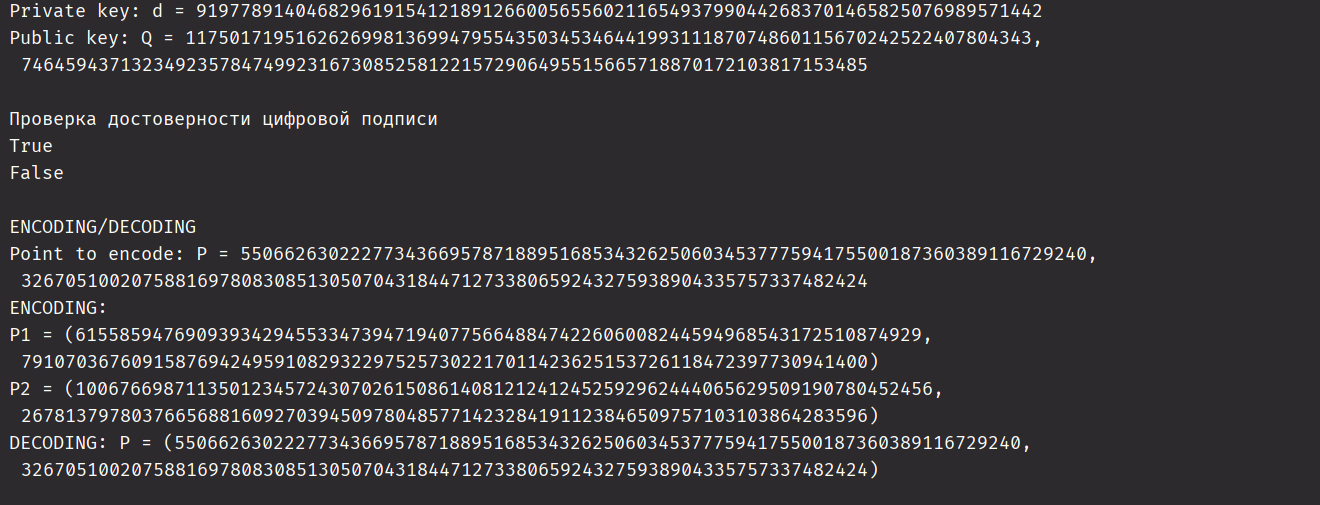
Чтобы дешифровать сообщение, участник В умножает первую координату точки на свой закрытый ключ и вычитает результат из второй координаты:

Pm + k × PB - nB × (k × G) = Pm + k × (nB × G) - nB × (k × G) = Pm

Участник А зашифровал сообщение Pm добавлением к нему kxPB. Никто не знает значения k, поэтому, хотя PB и является открытым ключом, никто не знает k × PB. Противнику для восстановления сообщения придется вычислить k, зная G и k × G. Сделать это будет нелегко.

Получатель также не знает k, но ему в качестве подсказки посылается k × G. Умножив k × G на свой закрытый ключ, получатель получит значение, которое было добавлено отправителем к незашифрованному сообщению. Тем самым получатель, не зная k, но имея свой закрытый ключ, может восстановить незашифрованное сообщение.

**Результат работы программы**



**Код программы**

from hashlib import sha256  
import random  
  
  
class Point(*object*):  
 def *\_\_init\_\_*(*self*, *x*, *y*):  
 *self*.x, *self*.y = *x*, *y  
 self*.inf = False  
  
 @classmethod  
 def atInfinity(*cls*):  
 P = cls(0, 0)  
 P.inf = True  
 return P  
  
 @classmethod  
 def secp256k1(*cls*):  
 return *cls*(55066263022277343669578718895168534326250603453777594175500187360389116729240,  
 32670510020758816978083085130507043184471273380659243275938904335757337482424)  
  
 def *\_\_eq\_\_*(*self*, *other*):  
 if *self*.inf:  
 return *other*.inf  
 elif *other*.inf:  
 return *self*.inf  
 else:  
 return *self*.x == *other*.x and *self*.y == *other*.y  
  
 def is\_infinite(*self*):  
 return *self*.inf  
  
  
class Curve(*object*):  
 *# Set attributes of a general Weierstrass cubic y^2 = x^3 + ax^2 + bx + c over any field.* def *\_\_init\_\_*(*self*, *a*, *b*, *c*, *char*, *exp*):  
 *self*.a, *self*.b, *self*.c = *a*, *b*, *c  
 self*.char, *self*.exp = *char*, *exp* def mult(*self*, *P*, *k*):  
 if *P*.is\_infinite():  
 return *P* elif *k* == 0:  
 return Point.atInfinity()  
 elif *k* < 0:  
 return *self*.mult(*self*.invert(*P*), -*k*)  
 else:  
 b = *bin*(*k*)[2:]  
 return *self*.repeat\_additions(*P*, b, 1)  
  
 def repeat\_additions(*self*, *P*, *b*, *n*):  
 if *b* == '0':  
 return Point.atInfinity()  
 elif *b* == '1':  
 return *P* elif *b*[-1] == '0':  
 return *self*.repeat\_additions(*self*.add(*P*, *P*), *b*[:-1], *n* + 1)  
 elif *b*[-1] == '1':  
 return *self*.add(*P*, *self*.repeat\_additions(*self*.add(*P*, *P*), *b*[:-1], *n* + 1))  
  
  
class CurveOverFp(Curve):  
 *# Construct a Weierstrass cubic y^2 = x^3 + ax^2 + bx + c over Fp.* def *\_\_init\_\_*(*self*, *a*, *b*, *c*, *p*):  
 Curve.*\_\_init\_\_*(*self*, *a*, *b*, *c*, *p*, 1)  
  
 *# The secp256k1 curve.* @classmethod  
 def secp256k1(*cls*):  
 return cls(0, 0, 7, 2 \*\* 256 - 2 \*\* 32 - 2 \*\* 9 - 2 \*\* 8 - 2 \*\* 7 - 2 \*\* 6 - 2 \*\* 4 - 1)  
  
 def contains(*self*, *P*):  
 if *P*.is\_infinite():  
 return True  
 else:  
 return (*P*.y \* *P*.y) % *self*.char == (*P*.x \* *P*.x \* *P*.x + *self*.a \* *P*.x \* *P*.x + *self*.b \* *P*.x + *self*.c) % *self*.char  
  
 def invert(*self*, *P*):  
 if *P*.is\_infinite():  
 return *P* else:  
 return Point(*P*.x, -*P*.y % *self*.char)  
  
 def add(*self*, *P\_1*, *P\_2*):  
 y\_diff = (*P\_2*.y - *P\_1*.y) % *self*.char  
 x\_diff = (*P\_2*.x - *P\_1*.x) % *self*.char  
 if *P\_1*.is\_infinite():  
 return *P\_2* elif *P\_2*.is\_infinite():  
 return *P\_1* elif x\_diff == 0 and y\_diff != 0:  
 return Point.atInfinity()  
 elif x\_diff == 0 and y\_diff == 0:  
 if *P\_1*.y == 0:  
 return Point.atInfinity()  
 else:  
 ld = ((3 \* *P\_1*.x \* *P\_1*.x + 2 \* *self*.a \* *P\_1*.x + *self*.b) \* mult\_inv(2 \* *P\_1*.y, *self*.char)) % *self*.char  
 else:  
 ld = (y\_diff \* mult\_inv(x\_diff, *self*.char)) % *self*.char  
 nu = (*P\_1*.y - ld \* *P\_1*.x) % *self*.char  
 x = (ld \* ld - *self*.a - *P\_1*.x - *P\_2*.x) % *self*.char  
 y = (-ld \* x - nu) % *self*.char  
 return Point(x, y)  
  
  
def hash\_and\_cut(*message*, *n*):  
 h = hash(*message*)  
 b = *bin*(h)[2:*len*(*bin*(*n*))]  
 return *int*(b, 2)  
  
  
def hash(*message*):  
 return *int*(sha256(*message*).hexdigest(), 16)  
  
  
def mult\_inv(*a*, *n*):  
 g, x, y = euclid(*a*, *n*)  
 return x % *n*def euclid(*sml*, *big*):  
 if *sml* == 0:  
 return *big*, 0, 1  
 else:  
 g, y, x = euclid(*big* % *sml*, *sml*)  
 return g, x - (*big* // *sml*) \* y, y  
  
  
*# 1 создание ключей*def prepare\_keys(*curve*, *P*, *n*):  
 d = random.randrange(1, *n*)  
 Q = *curve*.mult(*P*, d)  
 *print*("Private key: d = " + *str*(d))  
 *print*("Public key: Q = " + *str*(Q.x) + ',' + *str*(Q.y))  
 return d, Q  
  
  
*# 2 создание подписи*def sign(*message*, *curve*, *P*, *n*, *private*, *public*):  
 d, Q = *private*, *public* z = hash\_and\_cut(*message*, *n*)  
 r, s = 0, 0  
 while r == 0 or s == 0:  
 k = 4  
 R = *curve*.mult(*P*, k) *# (x1,y1)* r = R.x % *n* s = (mult\_inv(k, *n*) \* (z + r \* d)) % *n* return Q, r, s  
  
  
*# проверка цифровой подписи*def verify(*message*, *curve*, *P*, *n*, *signature*):  
 Q, r, s = *signature* if Q.is\_infinite() or not *curve*.contains(Q):  
 return False  
 if not *curve*.mult(Q, *n*).is\_infinite():  
 return False  
 if r > *n* or s > *n*:  
 return False  
 z = hash\_and\_cut(*message*, *n*)  
 w = mult\_inv(s, *n*) % *n* u\_1, u\_2 = z \* w % *n*, r \* w % *n* C\_1, C\_2 = *curve*.mult(*P*, u\_1), *curve*.mult(Q, u\_2)  
 C = *curve*.add(C\_1, C\_2)  
 return r % *n* == C.x % *n*def encode(*curve*, *G*, *Pm*, *Pb*):  
 k = random.randrange(1, 10000)  
 return *curve*.mult(*G*, k), *curve*.add(*curve*.mult(*Pb*, k), *Pm*)  
  
  
def decode(*curve*, *P1*, *P2*, *nb*):  
 return *curve*.add(*P2*, *curve*.invert(*curve*.mult(*P1*, *nb*)))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 *# y^2 = x^3 + 7* C = CurveOverFp.secp256k1()  
 P = Point.secp256k1()  
 n = 115792089237316195423570985008687907852837564279074904382605163141518161494337  
  
 priv, public = prepare\_keys(C, P, n)  
 msg = b'001'  
 sig = sign(msg, C, P, n, priv, public)  
 *print*("\nПроверка достоверности цифровой подписи ")  
 *print*(verify(b'001', C, P, n, sig))  
 *print*(verify(b'224', C, P, n, sig))  
 *print*('\nENCODING/DECODING')  
 *print*('Point to encode: P = {}, {}'.format(P.x, P.y))  
 enc\_pts = encode(C, P, P, public)  
 *print*('ENCODING: \nP1 = ({}, {})\nP2 = ({}, {})'.format(enc\_pts[0].x, enc\_pts[0].y, enc\_pts[1].x, enc\_pts[1].y))  
 decpt = decode(C, enc\_pts[0], enc\_pts[1], priv)  
 *print*('DECODING: P = ({}, {})'.format(decpt.x, decpt.y))

**Вывод**

Особое достоинство криптосистем на эллиптических кривых состоит в том, что по сравнению с системами на основе алгоритма RSA они обеспечивают существенно более высокую стойкость при равной трудоемкости или, наоборот, существенно меньшую трудоемкость при равной стойкости.