Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Лабораторная работа № 6

Компьютерная реализация электронной цифровой подписи на примере алгоритма ГОСТ 3410

Выполнил: Нетецкая Ю.В.

Проверил: Олисейчик В.В.

Минск 2021

**Постановка задачи**

В ходе данной лабораторной работы необходимо изучить теоретическую часть о электронной подписи и алгоритме ГОСТ 3410, на основании которых создать приложение, использующие данные технологии.

**Описание использованных алгоритмов**

1. В стандарте *ГОСТ 3410* используется хэш-функция ГОСТ 3411, которая создает хэш-код длиной 256 бит. Это во многом обуславливает требования к выбираемым простым числам p и q:
2. р должно быть простым числом в диапазоне

2509 < p < 2512либо 21020 < p < 21024

3. q должно быть простым числом в диапазоне 2254 < q < 2256

q также должно быть делителем (р-1).

4. Аналогично выбирается и параметр g.

При этом требуется, чтобы gq (mod p) = 1.

В соответствии с теоремой Ферма это эквивалентно условию в *DSS*,

что g = h(p-1)/q mod p.

5. Закрытым ключом является произвольное число х: 0 < x < q

6. Открытым ключом является число y: y = gx mod p

7. Для создания подписи выбирается случайное число k: 0 < k < q

Подпись состоит из двух чисел (r, s), вычисляемых по следующим формулам:

r = (gk mod p) mod q

s = (k H(M) + xr) mod q

**Хэш-функция ГОСТ 3411-94**

Алгоритм *ГОСТ 3411* является отечественным стандартом для хэш-функций. Длина хэш-кода, создаваемого алгоритмом *ГОСТ 3411*, равна 256 битам. Алгоритм разбивает сообщение на блоки, длина которых также равна 256 битам. Кроме того, параметром алгоритма является стартовый вектор хэширования *Н* - произвольное фиксированное значение длиной также 256 бит.

***Алгоритм обработки одного блока сообщения***

Сообщение обрабатывается блоками по 256 бит справа налево.

Каждый блок сообщения обрабатывается по следующему алгоритму.

1.           Генерация четырех ключей длиной 256 бит каждый.

2.           Шифрование 64-битных значений промежуточного хэш-кода H на ключах Ki(i = 1, 2, 3, 4) с использованием алгоритма ГОСТ 28147 в режиме простой замены.

3.           Перемешивание результата шифрования.

Для генерации ключей используются следующие данные:

·                промежуточное значение хэш-кода Н длиной 256 бит;

·                текущий обрабатываемый блок сообщения М длиной 256 бит;

·                параметры - три значения С2, С3 и С4 длиной 256 бит следующего вида: С2 и С4 состоят из одних нулей, а С3 равно

18 08 116 024 116 08 (08 18)2 18 08 (08 18)4 (18 08)4

где степень обозначает количество повторений 0 или 1.

Используются две формулы, определяющие перестановку и сдвиг.

Перестановка Р битов определяется следующим образом: каждое 256-битное значение рассматривается как последовательность тридцати двух 8-битных значений.

Перестановка Р элементов 256-битной последовательности выполняется по формуле y = φ(x), где x - порядковый номер 8-битного значения в исходной последовательности; y - порядковый номер 8-битного значения в результирующей последовательности.

φ(i + 1 + 4(k - 1)) = 8i + k ; i = 0 ÷ 3, k = 1 ÷ 8

Сдвиг А определяется по формуле

A (x) = (x1  x2) || x4 || x3 || x2

Где

|  |
| --- |
| xi - соответствующие 64 бита 256-битного значения х, |
| || обозначает конкатенацию. |

Присваиваются следующие начальные значения:

i = 1, U = H, V = M.

W = U V, K1 = Р (W)

Ключи K2, K3, K4 вычисляются последовательно по следующему алгоритму:

U = A(U)  Сi, V = A(A(V)),  W = U  V,  Ki = Р(W)

Далее выполняется шифрование 64-битных элементов текущего значения хэш-кода Н с ключами K1, K2, K3 и K4. При этом хэш-код Н рассматривается как последовательность 64-битных значений:

H = h4 || h3 || h2 || h1

Выполняется шифрование алгоритмом ГОСТ 28147:

si = EKi [hi]    i = 1, 2, 3, 4

S = s1 || s2 || s3 || s4

Наконец на заключительном этапе обработки очередного блока выполняется перемешивание полученной последовательности. 256-битное значение рассматривается как последовательность шестнадцати 16-битных значений. Сдвиг обозначается Ψ и определяется следующим образом:

|  |
| --- |
| η16 || η15 || ... || η1 - исходное значение |
| η1η2η3η4η13η16 || η16 || ... || η2 - результирующее значение |

Результирующее значение хэш-кода определяется следующим образом:

Χ(M, H) = ψ61 (H   ψ (M  ψ12(S)))

где

|  |
| --- |
| H - предыдущее значение хэш-кода, |
| М - текущий обрабатываемый блок, |
| Ψi - i-ая степень преобразования Ψ. |

Входными параметрами алгоритма являются:

·                исходное сообщение М произвольной длины;

·                стартовый вектор хэширования Н, длина которого равна 256 битам;

·                контрольная сумма Σ, начальное значение которой равно нулю и длина равна 256 битам;

·                переменная L, начальное значение которой равно длине сообщения.

Сообщение М делится на блоки длиной 256 бит и обрабатывается справа налево. Очередной блок i обрабатывается следующим образом:

1.           H = Χ(Mi, H)

2.           Σ = Σ  ' Mi

3.           L рассматривается как неотрицательное целое число, к этому числу прибавляется 256 и вычисляется остаток от деления получившегося числа на 2256. Результат присваивается L.

Где ' обозначает следующую операцию: Σ и Mi рассматриваются как неотрицательные целые числа длиной 256 бит. Выполняется обычное сложение этих чисел и находится остаток от деления результата сложения на 2256. Этот остаток и является результатом операции.

Самый левый, т.е. самый последний блок М' обрабатывается так:

1.           Блок добавляется слева нулями так, чтобы его длина стала равна 256 битам.

2.           Вычисляется Σ = Σ  ' Mi.

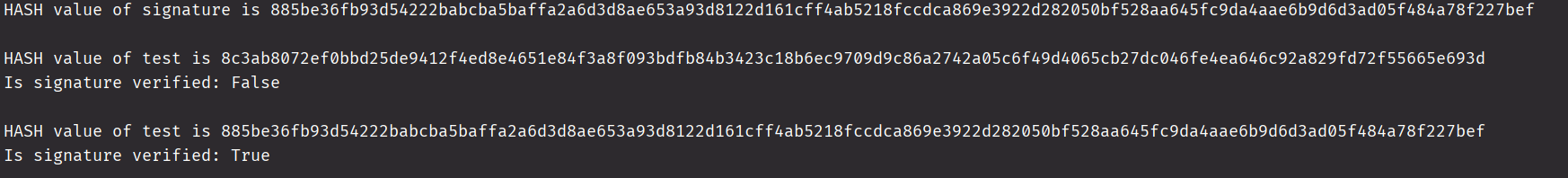
3.           L рассматривается как неотрицательное целое число, к этому числу прибавляется длина исходного сообщения М и находится остаток от деления результата сложения на 2256.

4.           Вычисляется Н = Χ(М', Н).

5.           Вычисляется Н = Χ(L, Н).

6.           Вычисляется Н = Χ(Σ, Н).

Значением функции хэширования является Н.

**Результат работы программы**

**Код программы**

import os  
  
  
p\_bytes = "fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffdc7"  
q\_bytes = "ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff27e69532f48d89116ff22b8d4e0560609b4b38abfad2b85dcacdb1411f10b275"  
a\_bytes = "fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffdc4"  
b\_bytes = "e8c2505dedfc86ddc1bd0b2b6667f1da34b82574761cb0e879bd081cfd0b6265ee3cb090f30d27614cb4574010da90dd862ef9d4ebee4761503190785a71c760"  
x\_bytes = "00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000003"  
y\_bytes = "7503cfe87a836ae3a61b8816e25450e6ce5e1c93acf1abc1778064fdcbefa921df1626be4fd036e93d75e6a50e3a41e98028fe5fc235f5b889a589cb5215f2a4"  
  
  
P = [  
 252, 238, 221, 17, 207, 110, 49, 22, 251, 196, 250,  
 218, 35, 197, 4, 77, 233, 119, 240, 219, 147, 46,  
 153, 186, 23, 54, 241, 187, 20, 205, 95, 193, 249,  
 24, 101, 90, 226, 92, 239, 33, 129, 28, 60, 66,  
 139, 1, 142, 79, 5, 132, 2, 174, 227, 106, 143,  
 160, 6, 11, 237, 152, 127, 212, 211, 31, 235, 52,  
 44, 81, 234, 200, 72, 171, 242, 42, 104, 162, 253,  
 58, 206, 204, 181, 112, 14, 86, 8, 12, 118, 18,  
 191, 114, 19, 71, 156, 183, 93, 135, 21, 161, 150,  
 41, 16, 123, 154, 199, 243, 145, 120, 111, 157, 158,  
 178, 177, 50, 117, 25, 61, 255, 53, 138, 126, 109,  
 84, 198, 128, 195, 189, 13, 87, 223, 245, 36, 169,  
 62, 168, 67, 201, 215, 121, 214, 246, 124, 34, 185,  
 3, 224, 15, 236, 222, 122, 148, 176, 188, 220, 232,  
 40, 80, 78, 51, 10, 74, 167, 151, 96, 115, 30,  
 0, 98, 68, 26, 184, 56, 130, 100, 159, 38, 65,  
 173, 69, 70, 146, 39, 94, 85, 47, 140, 163, 165,  
 125, 105, 213, 149, 59, 7, 88, 179, 64, 134, 172,  
 29, 247, 48, 55, 107, 228, 136, 217, 231, 137, 225,  
 27, 131, 73, 76, 63, 248, 254, 141, 83, 170, 144,  
 202, 216, 133, 97, 32, 113, 103, 164, 45, 43, 9,  
 91, 203, 155, 37, 208, 190, 229, 108, 82, 89, 166,  
 116, 210, 230, 244, 180, 192, 209, 102, 175, 194, 57,  
 75, 99, 182,  
]  
  
BYTES\_IN\_BLOCK = 64  
  
A = [*int*(x, 16) for x in (  
 "8e20faa72ba0b470", "47107ddd9b505a38", "ad08b0e0c3282d1c", "d8045870ef14980e",  
 "6c022c38f90a4c07", "3601161cf205268d", "1b8e0b0e798c13c8", "83478b07b2468764", "a011d380818e8f40", "5086e740ce47c920", "2843fd2067adea10", "14aff010bdd87508", "0ad97808d06cb404", "05e23c0468365a02", "8c711e02341b2d01", "46b60f011a83988e", "90dab52a387ae76f", "486dd4151c3dfdb9", "24b86a840e90f0d2", "125c354207487869", "092e94218d243cba", "8a174a9ec8121e5d", "4585254f64090fa0", "accc9ca9328a8950",  
 "9d4df05d5f661451", "c0a878a0a1330aa6", "60543c50de970553", "302a1e286fc58ca7", "18150f14b9ec46dd", "0c84890ad27623e0", "0642ca05693b9f70", "0321658cba93c138", "86275df09ce8aaa8", "439da0784e745554", "afc0503c273aa42a", "d960281e9d1d5215", "e230140fc0802984", "71180a8960409a42", "b60c05ca30204d21", "5b068c651810a89e", "456c34887a3805b9", "ac361a443d1c8cd2", "561b0d22900e4669", "2b838811480723ba",  
 "9bcf4486248d9f5d", "c3e9224312c8c1a0", "effa11af0964ee50", "f97d86d98a327728", "e4fa2054a80b329c", "727d102a548b194e", "39b008152acb8227", "9258048415eb419d", "492c024284fbaec0", "aa16012142f35760", "550b8e9e21f7a530", "a48b474f9ef5dc18", "70a6a56e2440598e", "3853dc371220a247", "1ca76e95091051ad", "0edd37c48a08a6d8", "07e095624504536c", "8d70c431ac02a736", "c83862965601dd1b", "641c314b2b8ee083",  
)]  
  
tau = (  
 0, 8, 16, 24, 32, 40, 48, 56,  
 1, 9, 17, 25, 33, 41, 49, 57,  
 2, 10, 18, 26, 34, 42, 50, 58,  
 3, 11, 19, 27, 35, 43, 51, 59,  
 4, 12, 20, 28, 36, 44, 52, 60,  
 5, 13, 21, 29, 37, 45, 53, 61,  
 6, 14, 22, 30, 38, 46, 54, 62,  
 7, 15, 23, 31, 39, 47, 55, 63,  
)  
  
C = [  
 [7, 69, 166, 242, 89, 101, 128, 221, 35, 77, 116, 204, 54, 116, 118, 5, 21, 211, 96, 164, 8, 42, 66, 162, 1, 105, 103, 146, 145, 224, 124, 75, 252, 196, 133, 117, 141, 184, 78, 113, 22, 208, 69, 46, 67, 118, 106, 47, 31, 124, 101, 192, 129, 47, 203, 235, 233, 218, 202, 30, 218, 91, 8, 177],  
 [183, 155, 177, 33, 112, 4, 121, 230, 86, 205, 203, 215, 27, 162, 221, 85, 202, 167, 10, 219, 194, 97, 181, 92, 88, 153, 214, 18, 107, 23, 181, 154, 49, 1, 181, 22, 15, 94, 213, 97, 152, 43, 35, 10, 114, 234, 254, 243, 215, 181, 112, 15, 70, 157, 227, 79, 26, 47, 157, 169, 138, 181, 163, 111],  
 [178, 10, 186, 10, 245, 150, 30, 153, 49, 219, 122, 134, 67, 244, 182, 194, 9, 219, 98, 96, 55, 58, 201, 193, 177, 158, 53, 144, 228, 15, 226, 211, 123, 123, 41, 177, 20, 117, 234, 242, 139, 31, 156, 82, 95, 94, 241, 6, 53, 132, 61, 106, 40, 252, 57, 10, 199, 47, 206, 43, 172, 220, 116, 245],  
 [46, 209, 227, 132, 188, 190, 12, 34, 241, 55, 232, 147, 161, 234, 83, 52, 190, 3, 82, 147, 51, 19, 183, 216, 117, 214, 3, 237, 130, 44, 215, 169, 63, 53, 94, 104, 173, 28, 114, 157, 125, 60, 92, 51, 126, 133, 142, 72, 221, 228, 113, 93, 160, 225, 72, 249, 210, 102, 21, 232, 179, 223, 31, 239],  
 [87, 254, 108, 124, 253, 88, 23, 96, 245, 99, 234, 169, 126, 162, 86, 122, 22, 26, 39, 35, 183, 0, 255, 223, 163, 245, 58, 37, 71, 23, 205, 191, 189, 255, 15, 128, 215, 53, 158, 53, 74, 16, 134, 22, 31, 28, 21, 127, 99, 35, 169, 108, 12, 65, 63, 154, 153, 71, 71, 173, 172, 107, 234, 75],  
 [110, 125, 100, 70, 122, 64, 104, 250, 53, 79, 144, 54, 114, 197, 113, 191, 182, 198, 190, 194, 102, 31, 242, 10, 180, 183, 154, 28, 183, 166, 250, 207, 198, 142, 240, 154, 180, 154, 127, 24, 108, 164, 66, 81, 249, 196, 102, 45, 192, 57, 48, 122, 59, 195, 164, 111, 217, 211, 58, 29, 174, 174, 79, 174],  
 [147, 212, 20, 58, 77, 86, 134, 136, 243, 74, 60, 162, 76, 69, 23, 53, 4, 5, 74, 40, 131, 105, 71, 6, 55, 44, 130, 45, 197, 171, 146, 9, 201, 147, 122, 25, 51, 62, 71, 211, 201, 135, 191, 230, 199, 198, 158, 57, 84, 9, 36, 191, 254, 134, 172, 81, 236, 197, 170, 238, 22, 14, 199, 244],  
 [30, 231, 2, 191, 212, 13, 127, 164, 217, 168, 81, 89, 53, 194, 172, 54, 47, 196, 165, 209, 43, 141, 209, 105, 144, 6, 155, 146, 203, 43, 137, 244, 154, 196, 219, 77, 59, 68, 180, 137, 30, 222, 54, 156, 113, 248, 183, 78, 65, 65, 110, 12, 2, 170, 231, 3, 167, 201, 147, 77, 66, 91, 31, 155],  
 [219, 90, 35, 131, 81, 68, 97, 114, 96, 42, 31, 203, 146, 220, 56, 14, 84, 156, 7, 166, 154, 138, 43, 123, 177, 206, 178, 219, 11, 68, 10, 128, 132, 9, 13, 224, 183, 85, 217, 60, 36, 66, 137, 37, 27, 58, 125, 58, 222, 95, 22, 236, 216, 154, 76, 148, 155, 34, 49, 22, 84, 90, 143, 55],  
 [237, 156, 69, 152, 251, 199, 180, 116, 195, 182, 59, 21, 209, 250, 152, 54, 244, 82, 118, 59, 48, 108, 30, 122, 75, 51, 105, 175, 2, 103, 231, 159, 3, 97, 51, 27, 138, 225, 255, 31, 219, 120, 138, 255, 28, 231, 65, 137, 243, 243, 228, 178, 72, 229, 42, 56, 82, 111, 5, 128, 166, 222, 190, 171],  
 [27, 45, 243, 129, 205, 164, 202, 107, 93, 216, 111, 192, 74, 89, 162, 222, 152, 110, 71, 125, 29, 205, 186, 239, 202, 185, 72, 234, 239, 113, 29, 138, 121, 102, 132, 20, 33, 128, 1, 32, 97, 7, 171, 235, 187, 107, 250, 216, 148, 254, 90, 99, 205, 198, 2, 48, 251, 137, 200, 239, 208, 158, 205, 123],  
 [32, 215, 27, 241, 74, 146, 188, 72, 153, 27, 178, 217, 213, 23, 244, 250, 82, 40, 225, 136, 170, 164, 29, 231, 134, 204, 145, 24, 157, 239, 128, 93, 155, 159, 33, 48, 212, 18, 32, 248, 119, 29, 223, 188, 50, 60, 164, 205, 122, 177, 73, 4, 176, 128, 19, 210, 186, 49, 22, 241, 103, 231, 142, 55],  
]  
  
  
def sum\_two(*a*, *b*):  
 carry = 0  
 res = *bytearray*(64)  
 for i in *range*(64):  
 res[i] = (*a*[i] + *b*[i] + carry)  
 carry = res[i] // 256  
 res[i] %= 256  
 return *list*(res)  
  
  
def xor\_two(*a*, *b*):  
 xor = *bytearray*(*min*(*len*(*a*), *len*(*b*)))  
 for i in *range*(*min*(*len*(*a*), *len*(*b*))):  
 xor[i] = *a*[i] ^ *b*[i]  
 return *bytes*(xor)  
  
  
def to\_bytearray(*n*):  
 res = *bytearray*(8)  
 for i in *range*(8):  
 res[i] = *n* % 256  
 *n* //= 256  
 return res  
  
  
def g(*n*, *hash*, *message*):  
 result = E(LPS(xor\_two(*hash*[:8], to\_bytearray(*n*)) + *hash*[8:]), *message*)  
 return xor\_two(xor\_two(result, *hash*), *message*)  
  
  
def E(*k*, *message*):  
 for i in *range*(12):  
 message = LPS(xor\_two(*k*, *message*))  
 k = (xor\_two(*k*, C[i]))  
 result = xor\_two(*k*, *message*)  
 return result  
  
  
def LPS(*data*):  
 res = L(PS(*bytearray*(*data*)))  
 return res  
  
  
def PS(*data*):  
 res = *bytearray*(BYTES\_IN\_BLOCK)  
 for i in *range*(BYTES\_IN\_BLOCK):  
 res[tau[i]] = P[*data*[i]]  
 return res  
  
  
def L(*data*):  
 res = []  
 for i in *range*(8):  
 val = *int*(*data*[i \* 8:i \* 8 + 8][::-1].hex(), 16)  
 current\_result = 0  
 for j in *range*(BYTES\_IN\_BLOCK):  
 if val & 0x8000000000000000:  
 current\_result ^= A[j]  
 val \*= 2  
 res.append(to\_bytearray(current\_result))  
 return b''.join(res)  
  
  
class Gost3411:  
 def *\_\_init\_\_*(*self*):  
 pass  
  
 def get\_hash(*self*, *data*):  
 H = BYTES\_IN\_BLOCK \* b'\x00'  
 SUM = BYTES\_IN\_BLOCK \* b'\x00'  
  
 n = 0  
 for i in *range*(0, *len*(*data*) // BYTES\_IN\_BLOCK \* BYTES\_IN\_BLOCK, BYTES\_IN\_BLOCK):  
 cur\_data = *data*[i:i + BYTES\_IN\_BLOCK]  
 H = g(n, H, cur\_data)  
 SUM = sum\_two(SUM, cur\_data)  
 n += BYTES\_IN\_BLOCK \* 8  
  
 adding\_bytes\_cnt = *len*(*data*) \* 8 - n  
 *data* += b'\x01'  
 zeros\_to\_add = (BYTES\_IN\_BLOCK - *len*(*data*) % BYTES\_IN\_BLOCK) % BYTES\_IN\_BLOCK  
 *data* += b'\x00' \* zeros\_to\_add  
  
 H = g(n, H, *data*[-BYTES\_IN\_BLOCK:])  
 SUM = sum\_two(SUM, *data*[-BYTES\_IN\_BLOCK:])  
 n += adding\_bytes\_cnt  
 H = g(0, H, to\_bytearray(n) + 56 \* b'\x00')  
 H = g(0, H, SUM)  
 return H  
  
  
def inv(*x*, *mod*):  
 assert *x* >= 0  
 t, newt = 0, 1  
 r, newr = *mod*, *x* while newr != 0:  
 q = r // newr  
 t, newt = newt, t - q \* newt  
 r, newr = newr, r - q \* newr  
 assert r <= 1  
 if t < 0:  
 t = t + *mod* return t  
  
  
class Curve:  
 def *\_\_init\_\_*(*self*):  
 *self*.p = 13407807929942597099574024998205846127479365820592393377723561443721764030073546976801874298166903427690031858186486050853753882811946569946433649006083527  
 *self*.q = 13407807929942597099574024998205846127479365820592393377723561443721764030073449232318290585817636498049628612556596899500625279906416653993875474742293109  
 *self*.a = 13407807929942597099574024998205846127479365820592393377723561443721764030073546976801874298166903427690031858186486050853753882811946569946433649006083524  
 *self*.b = 12190580024266230156189424758340094075514844064736231252208772337825397464478540423418981074322718899427039088997221609947354520590448683948135300824418144  
 *self*.x = 3  
 *self*.y = 6128567132159368375550676650534153371826708807906353132296049546866464545472607119134529221703336921516405107369028606191097747738367571924466694236795556  
  
 def add(*self*, *x1*, *y1*, *x2*, *y2*):  
 if *x1* == *x2* and *y1* == *y2*:  
 t = ((3 \* *x1* \* *x1* + *self*.a) \* inv(2 \* *y1*, *self*.p)) % *self*.p  
 else:  
 tx = ((*x2* - *x1*) % *self*.p + *self*.p) % *self*.p  
 ty = ((*y2* - *y1*) % *self*.p + *self*.p) % *self*.p  
 t = (ty \* inv(tx, *self*.p)) % *self*.p  
 tx = ((t \* t - *x1* - *x2*) % *self*.p + *self*.p) % *self*.p  
 ty = (t \* (*x1* - tx) - *y1*) % *self*.p  
 return tx, ty  
  
 def exp(*self*, *n*, *x*=None, *y*=None):  
 x = *x* or *self*.x  
 y = *y* or *self*.y  
 ax = *x* ay = *y  
 n* -= 1  
 assert *n* != 0  
  
 while *n* > 0:  
 if *n* % 2:  
 ax, ay = *self*.add(ax, ay, *x*, *y*)  
 x, y = *self*.add(*x*, *y*, *x*, *y*)  
 *n* //= 2  
 return ax, ay  
  
  
def public\_key(*curve*, *private*):  
 return *curve*.exp(*private*)  
  
  
def private\_key():  
 return *int*(os.urandom(128).hex(), 16)  
  
  
class Gost3410:  
 def *\_\_init\_\_*(*self*):  
 pass  
  
 *# создание подписи* def sign(*self*, *curve*, *private*, *z*):  
  
 q = *curve*.q  
 e = *z* % q  
 if e == 0:  
 e = 1  
  
 s = 0  
 r = 0  
 while s == 0:  
 k = *int*(os.urandom(64).hex(), 16) % (q - 1) + 1  
 r, \_ = *curve*.exp(k)  
 r %= q  
 if r == 0:  
 continue  
 s = (r \* *private* + k \* e) % q  
 return r, s  
  
 *# проверка подписи* def verify(*self*, *curve*, *pub*, *z*, *signature*):  
  
 r, s = *signature* q = *curve*.q  
  
 if r <= 0 or r >= q or s <= 0 or s >= q:  
 return False  
  
 e = *z* % *curve*.q  
 if e == 0:  
 e = 1  
 v = inv(e, q)  
 z1 = s \* v % q  
 z2 = (q - r \* v % q) % q  
 p1x, p1y = *curve*.exp(z1)  
 q1x, q1y = *curve*.exp(z2, *pub*[0], *pub*[1])  
  
 R, \_ = *curve*.add(p1x, p1y, q1x, q1y)  
 R %= q  
 return R == r  
  
  
def prepare\_hash(*bytes\_object*):  
 gost3411 = Gost3411()  
 return gost3411.get\_hash(*bytes\_object*).hex()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 hash\_value\_sign = prepare\_hash(b'001ab01')  
 *print*('HASH value of signature is ' + hash\_value\_sign)  
  
 curve = Curve()  
 privkey = private\_key()  
 pubkey = public\_key(curve, privkey)  
  
 gost = Gost3410()  
 signature = gost.sign(curve, privkey, *int*(hash\_value\_sign, 16))  
  
 hash\_value\_test\_false = prepare\_hash(b'001sssb01')  
 *print*('\nHASH value of test is ' + hash\_value\_test\_false)  
 is\_verified\_false = gost.verify(curve, pubkey, *int*(hash\_value\_test\_false, 16), signature)  
 *print*('Is signature verified: ' + *str*(is\_verified\_false))  
  
 hash\_value\_test\_true = prepare\_hash(b'001ab01')  
 *print*('\nHASH value of test is ' + hash\_value\_test\_true)  
 is\_verified\_true = gost.verify(curve, pubkey, *int*(hash\_value\_test\_true, 16), signature)  
 *print*('Is signature verified: ' + *str*(is\_verified\_true))

**Вывод**

В рамках данной лабораторной работы были рассмотрены цифровая подпись и ГОСТ 3410.

Для демонстрации работы описанных выше алгоритмов было разработано и реализовано приложение с использованием указанного выше алгоритма.