Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

ОТЧЁТ

по лабораторной работе №6

По дисциплине «Методы защиты информации»

Выполнил:

Студент гр. 853505

Лазарева Е.В.

Проверил:

Олисейчик В.В.

Минск 2021

**1. Постановка задачи**

Необходимо реализовать программные средства формирования электронной цифровой подписи на примере алгоритма ГОСТ 3410*.*

**ЗАДАНИЕ:**

1. Изучить теоретические сведения.
2. Реализовать программное стредство формирования и проверки ЭЦП на базе алгоритма ГОСТ 3410.

**2. Теоретическая** **сведения**

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:

р должно быть простым числом в диапазоне

2509 < p < 2512либо 21020 < p < 21024

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

q также должно быть делителем (р-1).

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.

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

Открытым ключом является число y: y = gx mod p

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

Обратим внимание на отличия *DSS* и *ГОСТ 3410*.

1)                Используются разные хэш-функции: в *ГОСТ 3410* применяется отечественный стандарт на хэш-функции ГОСТ 3411, в *DSS* используется SHA-1, которые имеют разную длину хэш-кода. Отсюда и разные требования на длину простого числа q: в *ГОСТ 3410* длина q должна быть от 254 бит до 256 бит, а в *DSS* длина q должна быть от 159 бит до 160 бит.

2)                По-разному вычисляется компонента s подписи.

В *ГОСТ 3410* компонента s вычисляется по формуле:

s = (k H(M) + xr) mod q

В *DSS* компонента s вычисляется по формуле:

s = [k-1 (H(M) + xr)] mod q

Последнее отличие приводит к соответствующим отличиям в формулах для проверки подписи.

Получатель вычисляет w = H(M)-1 mod q

u1 = w s mod q

u2 = (q-r) w mod q

v = [(gu1 yu2) mod p] mod q

Подпись корректна, если v = r.

Структура обоих алгоритмов довольно интересна. Заметим, что значение r совсем не зависит от сообщения. Вместо этого r есть функция от k и трех общих компонент открытого ключа. Мультипликативная инверсия k (mod p) (в случае *DSS*) или само значение k (в случае ГОСТ 3410) подается в функцию, которая, кроме того, в качестве входа имеет хэш-код сообщения и закрытый ключ пользователя. Эта функция такова, что получатель может вычислить r, используя входное сообщение, подпись, открытый ключ пользователя и общий открытый ключ.

В силу сложности вычисления дискретных логарифмов нарушитель не может восстановить k из r или х из s.

Другое важное замечание заключается в том, что экспоненциальные вычисления при создании подписи необходимы только для gk mod p. Так как это значение от подписываемого сообщения не зависит, оно может быть вычислено заранее. Пользователь может заранее просчитать некоторое количество значений r и использовать их по мере необходимости для подписи документов. Еще одна задача состоит в определении мультипликативной инверсии k-1 (в случае *DSS*). Эти значения также могут быть вычислены заранее.

**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.

1. **Пример работы программы**



**4. Код программы**

from os import urandom  
from hashlib import sha1  
from codecs import getdecoder  
from codecs import getencoder  
  
  
\_hexdecoder = getdecoder("hex")  
\_hexencoder = getencoder("hex")  
  
def hexdec(data):  
 return \_hexdecoder(data)[0]  
  
def hexenc(data):  
 return \_hexencoder(data)[0].decode("ascii")  
  
  
def modinvert(a, n):  
 if a < 0:  
 # k^-1 = p - (-k)^-1 mod p  
 return n - modinvert(-a, n)  
 t, newt = 0, 1  
 r, newr = n, a  
 while newr != 0:  
 quotinent = r // newr  
 t, newt = newt, t - quotinent \* newt  
 r, newr = newr, r - quotinent \* newr  
 if r > 1:  
 return -1  
 if t < 0:  
 t = t + n  
 return t  
  
def bytes2long(raw):  
 return int(hexenc(raw), 16)  
  
  
def long2bytes(n, size=32):  
 res = hex(int(n))[2:].rstrip("L")  
 if len(res) % 2 != 0:  
 res = "0" + res  
 s = hexdec(res)  
 if len(s) != size:  
 s = (size - len(s)) \* b"\x00" + s  
 return s  
  
class GOST3410Curve(object):  
 def \_\_init\_\_(self, p, q, a, b, x, y, e=None, d=None):  
 self.p = p  
 self.q = q  
 self.a = a  
 self.b = b  
 self.x = x  
 self.y = y  
 self.e = e  
 self.d = d  
 r1 = self.y \* self.y % self.p  
 r2 = ((self.x \* self.x + self.a) \* self.x + self.b) % self.p  
 if r1 != self.pos(r2):  
 raise ValueError("Invalid parameters")  
 self.\_st = None  
  
 def pos(self, v):  
 if v < 0:  
 return v + self.p  
 return v  
  
 def \_add(self, p1x, p1y, p2x, p2y):  
 if p1x == p2x and p1y == p2y:  
 # double  
 t = ((3 \* p1x \* p1x + self.a) \* modinvert(2 \* p1y, self.p)) % self.p  
 else:  
 tx = self.pos(p2x - p1x) % self.p  
 ty = self.pos(p2y - p1y) % self.p  
 t = (ty \* modinvert(tx, self.p)) % self.p  
 tx = self.pos(t \* t - p1x - p2x) % self.p  
 ty = self.pos(t \* (p1x - tx) - p1y) % self.p  
 return tx, ty  
  
 def exp(self, degree, x=None, y=None):  
 x = x or self.x  
 y = y or self.y  
 tx = x  
 ty = y  
 if degree == 0:  
 raise ValueError("Bad degree value")  
 degree -= 1  
 while degree != 0:  
 if degree & 1 == 1:  
 tx, ty = self.\_add(tx, ty, x, y)  
 degree = degree >> 1  
 x, y = self.\_add(x, y, x, y)  
 return tx, ty  
  
 def st(self):  
 *"""Compute s/t parameters for twisted Edwards curve points conversion  
 """* if self.e is None or self.d is None:  
 raise ValueError("non twisted Edwards curve")  
 if self.\_st is not None:  
 return self.\_st  
 self.\_st = (  
 self.pos(self.e - self.d) \* modinvert(4, self.p) % self.p,  
 (self.e + self.d) \* modinvert(6, self.p) % self.p,  
 )  
 return self.\_st  
  
def public\_key(curve, prv):  
 *""" Generate public key from the private one"""* return curve.exp(prv)  
  
  
def sign(curve, prv, digest):  
 *""" Calculate signature for provided digest"""* size = 64  
 q = curve.q  
 e = bytes2long(digest) % q  
 if e == 0:  
 e = 1  
 while True:  
 k = bytes2long(urandom(size)) % q  
 if k == 0:  
 continue  
 r, \_ = curve.exp(k)  
 r %= q  
 if r == 0:  
 continue  
 d = prv \* r  
 k \*= e  
 s = (d + k) % q  
 if s == 0:  
 continue  
 break  
 return long2bytes(s, size) + long2bytes(r, size)  
  
  
def verify(curve, pub, digest, signature):  
 *""" Verify provided digest with the signature"""* size = 64  
 if len(signature) != size \* 2:  
 raise ValueError("Invalid signature length")  
 q = curve.q  
 p = curve.p  
 s = bytes2long(signature[:size])  
 r = bytes2long(signature[size:])  
 if r <= 0 or r >= q or s <= 0 or s >= q:  
 return False  
 e = bytes2long(digest) % curve.q  
 if e == 0:  
 e = 1  
 v = modinvert(e, q)  
 z1 = s \* v % q  
 z2 = q - r \* v % q  
 p1x, p1y = curve.exp(z1)  
 q1x, q1y = curve.exp(z2, pub[0], pub[1])  
 lm = q1x - p1x  
 if lm < 0:  
 lm += p  
 lm = modinvert(lm, p)  
 z1 = q1y - p1y  
 lm = lm \* z1 % p  
 lm = lm \* lm % p  
 lm = lm - p1x - q1x  
 lm = lm % p  
 if lm < 0:  
 lm += p  
 lm %= q  
 # This is not constant time comparison!  
 return lm == r  
  
  
def prv\_unmarshal(prv):  
 *"""Unmarshal private key"""* return bytes2long(prv[::-1])  
  
  
def pub\_marshal(pub):  
 *"""Marshal public key"""* size = 32  
 return (long2bytes(pub[1], size) + long2bytes(pub[0], size))[::-1]  
  
  
def pub\_unmarshal(pub):  
 *"""Unmarshal public key"""* size = 32  
 pub = pub[::-1]  
 return (bytes2long(pub[size:]), bytes2long(pub[:size]))  
  
  
def uv2xy(curve, u, v):  
 *"""Convert twisted Edwards curve U,V coordinates to Weierstrass X,Y"""* s, t = curve.st()  
 k1 = (s \* (1 + v)) % curve.p  
 k2 = curve.pos(1 - v)  
 x = t + k1 \* modinvert(k2, curve.p)  
 y = k1 \* modinvert(u \* k2, curve.p)  
 return x % curve.p, y % curve.p  
  
  
def xy2uv(curve, x, y):  
 *"""Convert Weierstrass X,Y coordinates to twisted Edwards curve U,V"""* s, t = curve.st()  
 xmt = curve.pos(x - t)  
 u = xmt \* modinvert(y, curve.p)  
 v = curve.pos(xmt - s) \* modinvert(xmt + s, curve.p)  
 return u % curve.p, v % curve.p  
  
curve = GOST3410Curve(  
 p=bytes2long(hexdec("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD97")),  
 q=bytes2long(hexdec("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF6C611070995AD10045841B09B761B893")),  
 a=bytes2long(hexdec("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD94")),  
 b=bytes2long(hexdec("00000000000000000000000000000000000000000000000000000000000000a6")),  
 x=bytes2long(hexdec("0000000000000000000000000000000000000000000000000000000000000001")),  
 y=bytes2long(hexdec("8D91E471E0989CDA27DF505A453F2B7635294F2DDF23E3B122ACC99C9E9F1E14")),  
)  
prv\_raw = urandom(32)  
prv = prv\_unmarshal(prv\_raw)  
pub = public\_key(curve, prv)  
print("Public key is:", hexenc(pub\_marshal(pub)))  
data\_for\_signing = b"some data"  
dgst = sha1(data\_for\_signing).digest()  
signature = sign(curve, prv, dgst)  
verify(curve, pub, dgst, signature)

**5. Вывод**

Криптографическая стойкость первых стандартов цифровой подписи ГОСТ 3410 была основана на задаче дискретного логарифмирования в мультипликативной группе простого конечного поля большого порядка. Начиная с ГОСТ Р 34.10-2001 стойкость алгоритма основана на более сложной задаче вычисления дискретного логарифма в группе точек эллиптической кривой. Также стойкость алгоритма формирования цифровой подписи основана на стойкости соответствующей хеш-функции.