Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Лабораторная работа № 7

“Криптография с использованием эллиптических кривых.”

Выполнил студент гр. 753504

Горбачёнок К. Н.

Проверил

Протько М.И.

Минск, 2020

# 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 алгоритма

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.

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}

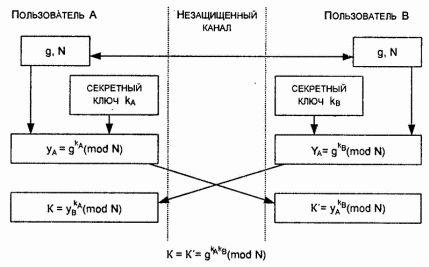
Чтобы дешифровать сообщение, участник В умножает первую координату точки на свой закрытый ключ и вычитает результат из второй координаты:

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.

# Блок-схема алгоритма



***Блок-схема алгоритма ECDSA***

# Результаты работы программы

# 

***Исходный файл***

******

***Результат работы программы***

# Прграммный код

from hashlib import sha256

from .signature import Signature

from .math import Math

from .utils.binary import BinaryAscii

from .utils.integer import RandomInteger

from .utils.compatibility import \*

class Ecdsa:

@classmethod

def sign(cls, message, privateKey, hashfunc=sha256):

hashMessage = hashfunc(toBytes(message)).digest()

numberMessage = BinaryAscii.numberFromString(hashMessage)

curve = privateKey.curve

r, s, randSignPoint = 0, 0, None

while r == 0 or s == 0:

randNum = RandomInteger.between(1, curve.N - 1)

randSignPoint = Math.multiply(curve.G, n=randNum, A=curve.A, P=curve.P, N=curve.N)

r = randSignPoint.x % curve.N

s = ((numberMessage + r \* privateKey.secret) \* (Math.inv(randNum, curve.N))) % curve.N

recoveryId = randSignPoint.y & 1

if randSignPoint.y > curve.N:

recoveryId += 2

return Signature(r=r, s=s, recoveryId=recoveryId)

@classmethod

def verify(cls, message, signature, publicKey, hashfunc=sha256):

hashMessage = hashfunc(toBytes(message)).digest()

numberMessage = BinaryAscii.numberFromString(hashMessage)

curve = publicKey.curve

sigR = signature.r

sigS = signature.s

inv = Math.inv(sigS, curve.N)

u1 = Math.multiply(curve.G, n=(numberMessage \* inv) % curve.N, A=curve.A, P=curve.P, N=curve.N)

u2 = Math.multiply(publicKey.point, n=(sigR \* inv) % curve.N, A=curve.A, P=curve.P, N=curve.N)

add = Math.add(u1, u2, P=curve.P, A=curve.A)

return sigR == add.x

from .utils.compatibility import \*

from .utils.base import Base64

from .utils.binary import BinaryAscii

from .utils.der import encodeSequence, encodeInteger, removeSequence, removeInteger

class Signature:

def \_\_init\_\_(self, r, s, recoveryId=None):

self.r = r

self.s = s

self.recoveryId = recoveryId

def toDer(self, withRecoveryId=False):

encodedSequence = encodeSequence(encodeInteger(self.r), encodeInteger(self.s))

if not withRecoveryId:

return encodedSequence

return chr(27 + self.recoveryId) + encodedSequence

def toBase64(self, withRecoveryId=False):

return toString(Base64.encode(toBytes(self.toDer(withRecoveryId=withRecoveryId))))

@classmethod

def fromDer(cls, string, recoveryByte=False):

recoveryId = None

if recoveryByte:

recoveryId = string[0] if isinstance(string[0], intTypes) else ord(string[0])

recoveryId -= 27

string = string[1:]

rs, empty = removeSequence(string)

if len(empty) != 0:

raise Exception("trailing junk after DER signature: %s" % BinaryAscii.hexFromBinary(empty))

r, rest = removeInteger(rs)

s, empty = removeInteger(rest)

if len(empty) != 0:

raise Exception("trailing junk after DER numbers: %s" % BinaryAscii.hexFromBinary(empty))

return Signature(r=r, s=s, recoveryId=recoveryId)

@classmethod

def fromBase64(cls, string, recoveryByte=False):

der = Base64.decode(string)

return cls.fromDer(der, recoveryByte)

from algorithms import ellipticcurve

from helpers.colored\_text import colored\_text

from helpers.file\_info import read\_text

from algorithms.ellipticcurve.ecdsa import Ecdsa

from algorithms.ellipticcurve.privateKey import PrivateKey

def main():

text = read\_text(filename="resources/lab7.txt")

print(colored\_text("TEXT:"), text)

private\_key = PrivateKey()

public\_key = private\_key.publicKey()

print(colored\_text("PRIVATE KEY"), private\_key.toPem())

print(colored\_text("PUBLIC KEY"), public\_key.toPem())

signature = Ecdsa.sign(text, private\_key)

print(colored\_text("SIGNATURE:"), signature.toBase64())

print(colored\_text("IS SIGNATURE VERIFIED:"), Ecdsa.verify(text, signature, public\_key))

if \_\_name\_\_ == "\_\_main\_\_":

main()

# Вывод

В ходе написания лабораторной работы был изучен алгоритмы цифровой подписи ECDSA c использованием эллиптических кривых, а также написана программная реализация.