Практичне завдання:

**“Реалізація криптографічного алгоритму”**

**Мета лабораторної роботи:** вивчити найбільш поширені криптографічні алгоритми та реалізувати один.

# Теоретичні відомості

Розвиток криптографії почався ще з давніх-давен, а її історія налічує понад 4 тисячі років. Спочатку криптографія вирішувала лише одне завдання – забезпечення конфіденційності під час передачі та зберігання даних. На даний момент криптографія також використовується для забезпечення цілісності та автентичності даних.

На сьогоднішній день криптографія ґрунтується на математичних операціях, визначених у таких розділах математики, як теорія чисел, теорія груп, кілець, полів тощо.

Сучасна криптографія складається із схем симетричного та асиметричного шифрування, схем хешування даних, схем цифрового підпису, методів управління ключами, схем доказів з нульовим розголошенням, методів криптоаналізу та постквантової криптографії тощо.

Основним завданням інформаційної безпеки є забезпечення:

* конфіденційність даних,
* цілісність даних,
* доступність даних,
* автентичність даних.

*Конфіденційність* передбачає, що неавторизовані особи не можуть отримати доступ до даних, що зберігаються або передаються. Ця послуга безпеки може забезпечуватись за допомогою алгоритмів шифрування.

*Цілісність* передбачає, що дані не було змінено під час передачі чи виконання операції над ними. Для перевірки цілісності даних використовують хеш-функції.

*Доступність* передбачає, що суб'єкти, які мають право доступу до інформації, гарантовано отримають доступ до неї. Ця послуга безпеки не може забезпечуватися під час використання криптографічних методів.

*Автентичність* передбачає можливість доказу, що дані справді були отримані від конкретного автора. Перевірка автентичності даних виконується за допомогою механізму цифрового підпису (або MAC кодів).

## 1.1 Шифрування

Процес шифрування перетворює вихідні дані в байтову послідовність (вміст якої не можна передбачити до виконання шифрування), вихідний зміст якої не може бути отриманий стороною не має відповідного ключа розшифрування. Зашифровані дані можуть бути перетворені назад у їхній початковий вигляд, за умови, якщо є відповідний ключ.

Існує два основних типи шифрування – симетричне та асиметричне.

* *Симетричне шифрування* використовує той самий ключ і для шифрування, і для розшифровування.
* *Асиметричне шифрування* використовує два різні ключі: один для шифрування (відкритий ключ), інший для розшифровування (особистий ключ).

## 1.3 Алгоритми цифрового підпису

Цифровий підпис дозволяє встановити автентифікацію відправника інформації. Підпис пов'язаний як з автором, так і з самим документом за допомогою криптографічних методів, і не може бути підроблений для іншого документа за допомогою звичайного копіювання.

Для генерації пари ключів (особистого та публічного) в алгоритмах Електронного цифрового підпису (ЕЦП) або кваліфікованого електронного підпису (КЕП) використовуються різні математичні схеми, що ґрунтуються на застосуванні односпрямованих функцій. Традиційні схеми поділяються на дві групи. В основі такого поділу лежать відомі складні обчислювальні завдання: факторизація великих цілих чисел та дискретного логарифмування.

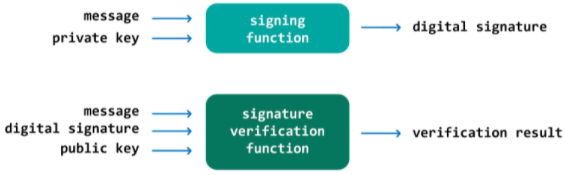


Рисунок 1.2 – Схема цифрового підпису

# Хід роботи

Для виконання лабораторної роботи ви можете використовувати будь-яку зручну для вас мову програмування. Завдання полягає у реалізації одного з запропонованих алгоритмів (можна обрати для реалізації такий алгоритм, який не описаний у даному методичному посібнику).

Опції виконання лабораторної роботи:

| Варіант | Алгоритм для реалізації | Basic level | Advanced level |
| --- | --- | --- | --- |
| 1 | Алгоритм шифрування | Vigenère Cipher (2.1) | AES (2.2) або будь-який інший алгоритм на власний розсуд. |
| 3 | Алгоритм цифрового підпису | RSA (2.3) | ECDSA(2.4), Schnorr signature (2.5), RingSig (2.6) або будь-який інший алгоритм на власний розсуд. |

Зазначимо, що вибір рівня виконання завдання не передбачає необхідність реалізації всіх зазначених алгоритмів. Реалізація алгоритмів у кількості більше за 1 заохочується, але не є необхідною.

## 

## 2.1 Vigenère Cipher

Шифр Віженера використовує просту форму поліалфавітної заміни. Поліалфавітний шифр — це будь-який шифр, що базується на заміні з використанням кількох алфавітів зміщення. Шифрування вихідного тексту здійснюється за допомогою таблиці Віженера. Таблиця складається з алфавітів (масиву символів), виписаних 26 разів у різні рядки (для англійського алфавіту), кожен алфавіт циклічно зсувається вліво порівняно з попереднім алфавітом.

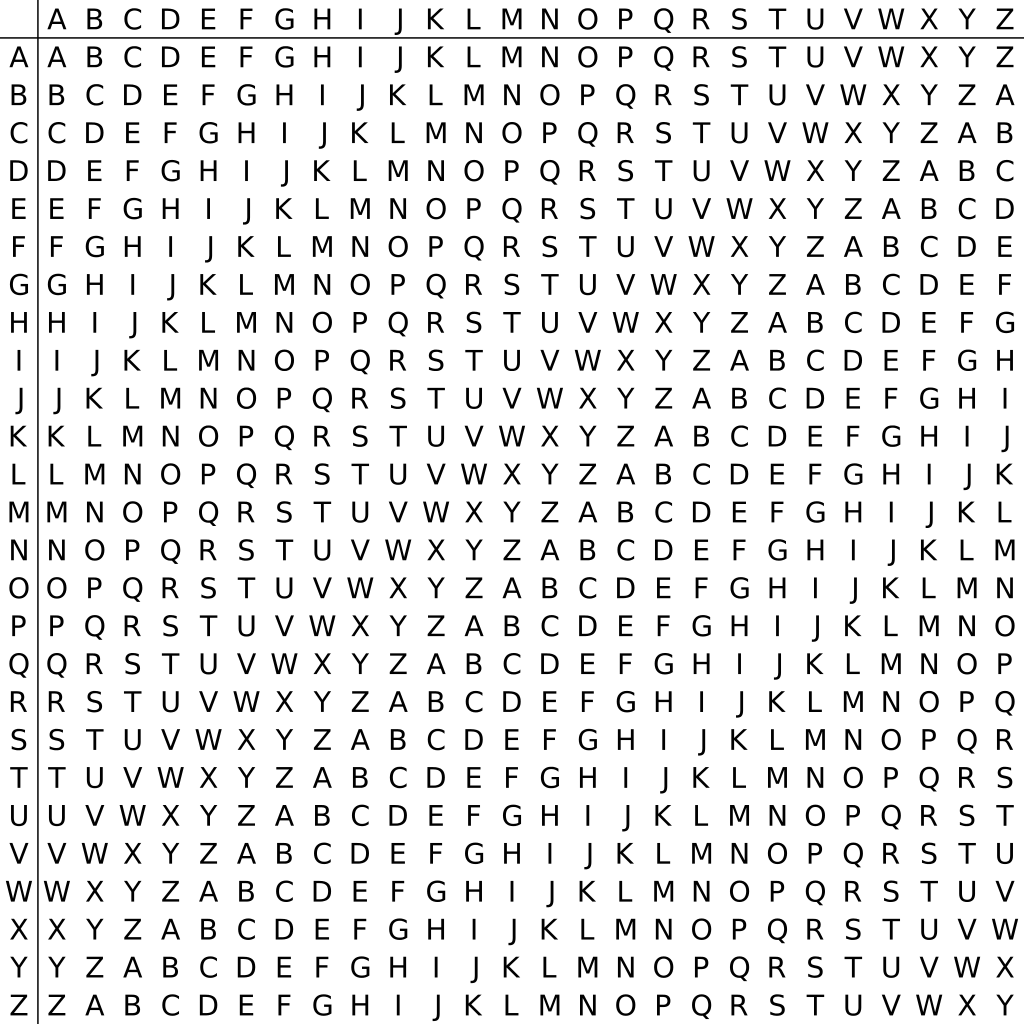


Рисунок 2.1 – Таблиця Віженера

Шифрування виконується таким чином:

Ci = (Pi + Kj) mod 26,

де Kj - j-я літера ключа,

Pi - і-я літера вихідного повідомлення, а

Ci - i-я літера шифртексту.

**Pseudo-code:**

m <-- length of key

for index, character in plaintext:

ciphertext[index] <-- (character + key[index % m]) % 26

return ciphertext

Правильність роботи вашої версії шифру Віженера ви можете перевірити на тих же векторах (використовуваних для тестування вашої програми) за наступним посиланням <https://cryptii.com/pipes/vigenere-cipher>.

Шифр Віженера є більш просунутою версією шифру Цезаря (який використовує моноалфавітну підстановку). Ви можете подати шифр Цезаря замість шифру Віженера. Для отримання опису алгоритму та можливості порівняння вашого алгоритму з референсними значеннями ви можете використати наступне джерело: <https://www.dcode.fr/caesar-cipher>.

## 

## 2.2 Шифр AES

Алгоритм шифрування AES визначає численні перетворення, які виконуються з даними, що зберігаються у масиві. AES має на увазі оперування масивом 4×4 байт, що називається станом (state) (у найпростішому випадку).

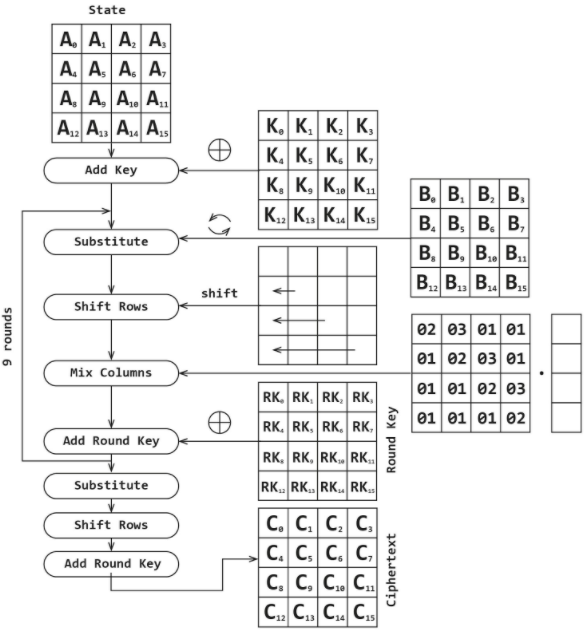


Рисунок 2.2 – Схема роботи AES

**Pseudo-code:**

function Encrypt(plaintext, key) {

blocks := DivideIntoBlocks(plaintext);

roundKeys = GetRoundKeys(key)

for (block in blocks) {

//first round

AddRoundKey(roundKeys[0], block);

//intermediate rounds

for (8, 10 or 12 rounds) {

SubBytes(block);

ShiftRows(block);

MixColumns(block);

AddRoundKey(roundKeys[..], block);

}

//last round

SubBytes(block);

ShiftRows(block);

AddRoundKey(roundKeys[numRounds - 1], block);

}

ciphertext := Reassemble(blocks);

return ciphertext;

}

1. Функція **DivideIntoBlocks** повинна виконувати поділ вхідного тексту на блоки 128 біт (16 байт). Кожен блок представляється як матриці 4x4 (приклад наступному рисунку).

| BA | AB | FF | 12 |
| --- | --- | --- | --- |
| 32 | 54 | 68 | F5 |
| 4E | AA | FF | 5F |
| 6D | EF | FE | 01 |

Рисунок 2.3 – Приклад блоку вихідного повідомлення, над яким виконуватиметься шифрування

1. Функція **GetRoundKeys** розгортає ключ. Оскільки AES має на увазі використання ключа довжиною 128 біт (у найпростішому випадку), це означає, що для даних більшої довжини повинна породжуватися ключова послідовність яка дорівнює довжині повідомлення, що шифрується. Для цього і використовується функція розгортання (окремий 128 бітний ключ шифрування генерується для кожного блоку, що шифрується).

**Pseudo-code:**

function GetRoundKeys(byte key[4\*Nk], word w[Nb\*(Nr+1)], Nk) {

begin

word temp

i = 0

while (i < Nk)

w[i] = word(key[4\*i], key[4\*i+1], key[4\*i+2], key[4\*i+3])

i = i+1

end while

i = Nk

while (i < Nb \* (Nr+1)]

temp = w[i-1]

if (i mod Nk = 0)

temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]

else if (Nk > 6 and i mod Nk = 4)

temp = SubWord(temp)

end if

w[i] = w[i-Nk] xor temp

i = i + 1

end while

end

}

1. Функція **AddRoundKey** виконує складання блоку з ключем RoundKey (операція XOR (⊕)).
2. Функція **SubBytes** необхідна для виконання підстановки: кожен байт у блоці замінюється відповідним елементом із фіксованої таблиці (S-box).

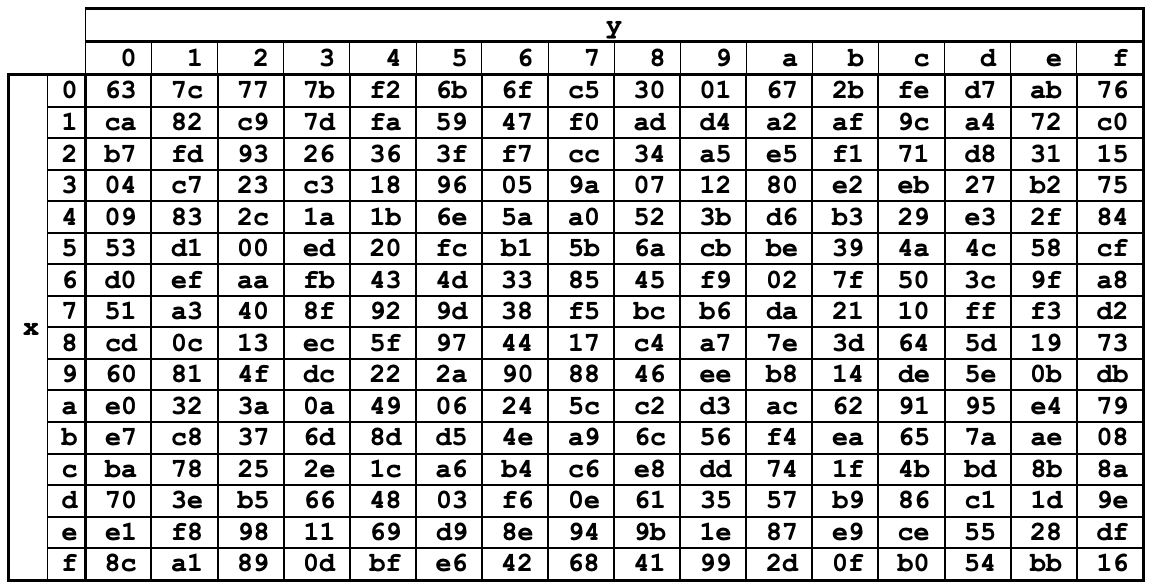


Рисунок 2.4 – Таблиця підстановки

Для зручності імплементації можна скопіювати значення для таблиці з масиву нижче:

Sbox = array{

0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,

0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,

0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,

0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,

0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,

0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,

0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,

0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,

0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,

0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,

0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,

0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,

0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,

0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,

0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,

0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16

};

1. Функція **ShiftRows** циклічно зсуває байти у кожному рядку блоку на r байт вліво в залежності від номера рядка.

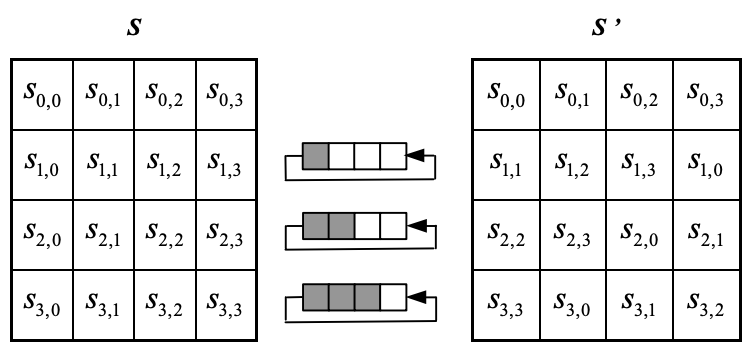


Рисунок 2.5 – Схема зсуву рядків блоку

1. Функція **MixColumns** полягає у перемноженні кожної колонки блоку з константною матрицею наступним чином:

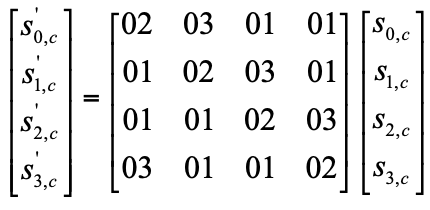


Рисунок 2.6 – Операція MixColumns

Розшифрування відбувається згідно з наступним алгоритмом:

**Pseudo-code:**

function InvCipher(byte in[4\*Nb], byte out[4\*Nb], word w[Nb\*(Nr+1)])

begin

byte state[4,Nb]

state = in

AddRoundKey(state, w[Nr\*Nb, (Nr+1)\*Nb-1])

for round = Nr-1 step -1 downto 1

InvShiftRows(state)

InvSubBytes(state)

AddRoundKey(state, w[round\*Nb, (round+1)\*Nb-1])

InvMixColumns(state) // See Sec. 5.3.3

end for

InvShiftRows(state)

InvSubBytes(state)

AddRoundKey(state, w[0, Nb-1])

out = state

end

Функції **InvShiftRows**, **InvSubBytes** і **InvMixColumns** є інверсними – тобто виконують зворотні операції (4-6).

Сам стандарт та тестові вектори ви можете знайти за наступним посиланням: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf>

## 2.3 Алгоритм RSA

Алгоритм шифрування RSA є асиметричним алгоритмом шифрування. Сьогодні дуже часто цей алгоритм використовується для організації захищеного каналу між користувачами (формування загального секрету та шифрування його за допомогою RSA).

Послідовність кроків алгоритму RSA (генерація ключів):

1. Обираються два великі прості числа p и q;
2. Обчислюється: n = p ⋅ q, m = (p – 1) ⋅ (q – 1);
3. Обирається випадкове число d, взаємно просте с m;
4. Визначається таке число e, для якого є істинним вираз: (e ⋅ d) mod (m) = 1;
5. Числа e та n – це відкритий ключ, а числа d и n – це закритий ключ.

Відкритим ключем шифрують повідомлення, а закритим – розшифровують. Пара чисел закритого ключа тримається у секреті.

Шифрування та дешифрування інформації за допомогою алгоритму

RSA:

1. Вихідний текст розбивається на блоки, кожен з яких може бути представлений у вигляді числа M(i);
2. Шифрування тексту виконується так: C(i) = (M(i)e) mod n;
3. Розшифрування повідомлення виконується так: M(i) = (C(i)d) mod n.

**Pseudo-code:**

KeyGen(){

P and Q <= primary big numbers.

n = P\*Q.

e <= integer value, not be a factor of n, 1<e<Φ(n), Φ(n) = (P-1)(Q-1)

d = (k\*Φ(n) + 1) / e , for some integer k

PrivateKey (d, n)

PublicKey (e, n)

}

Encrypt(message, PublicKey){

return powmod(message, e, n)

}

Decrypt(ciphertext, PrivateKey){

return powmod(ciphertext, d, n)

}

Сам текст стандарту та тестові вектори ви можете знайти за наступним посиланням:

<https://datatracker.ietf.org/doc/html/rfc3447>

## 2.4 Алгоритм ECDSA

Алгоритм цифрового підпису на основі кривих еліптичних (ECDSA) пропонує варіант алгоритму цифрового підпису (DSA), в якому використовується криптографія на основі еліптичних кривих.

**Pseudo-code:**

KeyGen(){

d <= random ∈ [1, n - 1].

Q = d x P.

// additional verifications

Checking that Q ≠ 0.

Checking that xQ and yQ are properly represented elements of Fq.

Checking that Q is on the elliptic curve defined by a and b.

Checking that nQ = Q.

If any of these checks fail the public key Q is invalid, otherwise Q is valid.

Sign(m, d){

k <= random ∈ [1, n - 1].

k x P = (x1, y1)

r = x1 mod n.

if r = 0 then go to start.

s = pow(k, -1)(H(m) + d • r) mod n.

if s = 0 go to start.

Signature (r, s).

}

Verify(r, s, Q){

Verify that r and s are integers in the interval [1,n-1].

c = s-1 mod n and H(m)

u1 = H(m) • c mod n

u2 = r • c mod n

u1 x P + u2 x Q = (x0, y0)

v = x0 mod n

if v = r -> return true

}

Сам текст стандарту та тестові вектори ви можете знайти за наступним посиланням:

<https://datatracker.ietf.org/doc/html/rfc6979>

## 

## 2.5 Підписи Шнорра

Підписи Шнорра – це вид цифрових підписів, які підтримують агрегацію публічних ключів та значень підписів, таким чином, щоб зробити мультипідпис компактним та анонімним (одиничний підпис жодним чином не відрізняється від підпису, згенерованого потенційно нескінченною кількістю учасників).

**Pseudo-code:**

KeyGen(){

d <= random ∈ [1, n - 1].

Q = d x P.

// additional verifications

Checking that Q ≠ 0.

Checking that xQ and yQ are properly represented elements of Fq.

Checking that Q is on the elliptic curve defined by a and b.

Checking that nQ = Q.

If any of these checks fail the public key Q is invalid, otherwise Q is valid.

Sign(m, d){

r <= random

R := scalarMult(P, \*r)

r = r + H(X,R,m)\*x

signature.R = R

signature.S = \*r

return

}

Verify(r, s, Q){

sP ?= R + H(X,R,m)Q

}

Алгоритм мультипідпису Шнорра має такий вигляд:

**Pseudo-code:**

Sign(){

L = H(X1,X2,…)

X = sum(H(L,Xi)\*Xi)

ri - rand

Ri = ri\*G

R = sum(Ri)

Each signer computes si = ri + H(X,R,m)\*H(L,Xi)\*xi

s is the sum of the si values

Verify(){

sG ?= R + H(X,R,m)\*X

}

<https://tlu.tarilabs.com/cryptography/introduction-schnorr-signatures>

## 2.6 Ring traceable signatures

Один із стандартів CryptoNote описує алгоритм одноразового кільцевого підпису. Такий підхід дозволяє користувачеві підписати транзакцію і зберегти при цьому анонімність, тому що верифікатор може переконатися, що підпис був вироблений одним із членів групи, не маючи можливості дізнатися, ким саме. Для запобігання атакі подвійної витрати було вирішено використати механізм кільцевого підпису, який використовує private key image.

**Pseudo-code:**

Signature input:

M - message

A[0...n-1] - public keys of ring participants,

а[i] - private signer key,

А[i] - public signer key.

Procedure generate\_signature (M, A[1], A[2], ..., A[n], i, a[i]):

I <- a[i]\*H(A[i]) // Private key image

c[j], r[j] [j=0..n-1, j!=i] <- random

k <- random

For j <- 0..n-1, j!=i

4.1. X[j] <- c[j]\*A[j]+r[j]\*G

4.2. Y[j] <- c[j]\*I+r[j]\*H(A[j])

X[i] <- k\*G

Y[i] <- k\*H(A[i])

c[i] <- H(H(M) || X[0] || Y[0] || X[1] || Y[1] || ... || X[n-1] || Y[n-1]) - Sum[j=0..n-1, j!=i](c[j])

r[i] <- k-a[i]\*c[i]

Return (I, c[0] || r[0] || c[1] || r[1] || ... || c[n-1] || r[n-1])

Procedure verify\_signature(M, A[0], A[1], ..., A[n-1], I, c[0], r[0], c[1], r[1], ..., c[n-1], r[n-1]):

For i <- 0..n-1

1.1. X[i] <- c[i]\*A[i]+r[i]\*G

1.2. Y[i] <- c[i]\*I+r[i]\*H(A[i])

If H(H(M) || X[0] || Y[0] || X[1] || Y[1] || ... || X[n-1] || Y[n-1]) == Sum[i=0..n-1](c[i])

Return "Correct"

Else

Return "Incorrect"

}

CryptoNote Standards:

<https://cryptonote.org/standards/>

***------------------------------------------------------------------------------------------------***

***Примітка\*. Додатково можна подивитись:***

<https://www.youtube.com/watch?v=gP4PqVGudtg&ab_channel=Computerphile>