

ДОДАТОК А  
(довідковий)  
Вихідний код програми

```
#include <stdlib.h>
#include <stdint.h>
#include "crypto_stream_salsa20.h"
#include "rng.h"
#include "shred.h"

#define RAND_LEN_BYTES (4096)

static int init = 0;
static unsigned char key[crypto_stream_salsa20_KEYBYTES];
static unsigned char nonce[crypto_stream_salsa20_NONCEBYTES] = {0};
static unsigned char randpool[RAND_LEN_BYTES];
static uint16_t randpos = RAND_LEN_BYTES;

void
fastrandbytes(unsigned char *r, unsigned long long rlen)
{
    unsigned long long n=0;
    uint8_t i;
    if(!init)
    {
        randombytes(key, crypto_stream_salsa20_KEYBYTES);
        init = 1;
    }
    crypto_stream_salsa20(r, rlen, nonce, key);

    // Increase 64-bit counter (nonce)
    for(i=0;i<8;i++)
        n ^= ((unsigned long long)nonce[i]) << 8*i;
    n++;
    for(i=0;i<8;i++)
```

```
        nonce[i] = (n >> 8*i) & 0xff;
    }

void rng_cleanup()
{
    if(init)
    {
        init = 0;
        shred(key, crypto_stream_salsa20_KEYBYTES);
        shred(nonce, crypto_stream_salsa20_KEYBYTES);
        shred(randpool, crypto_stream_salsa20_KEYBYTES);
    }
}

void
rng_init()
{
    fastrandombytes(randpool, RAND_LEN_BYTES);
    randpos = 0;
}

void
rng_uint16(uint16_t *r)
{
    if(randpos >= (RAND_LEN_BYTES - sizeof(uint16_t)))
    {
        fastrandombytes(randpool, RAND_LEN_BYTES);
        randpos = 0;
    }
    *r = (uint16_t)(randpool[randpos++] & 0xff) << 8;
    *r |= (uint16_t)(randpool[randpos++] & 0xff);

    return;
}
```

```

void
rng_uint64(uint64_t *r)
{
    if(randpos >= RAND_LEN_BYTES - sizeof(uint64_t))
    {
        fastrandombytes(randpool, RAND_LEN_BYTES);
        randpos = 0;
    }
    *r = ((uint64_t)(randpool[randpos++] & 0xff)) << 070;
    *r |= ((uint64_t)(randpool[randpos++] & 0xff)) << 060;
    *r |= ((uint64_t)(randpool[randpos++] & 0xff)) << 050;
    *r |= ((uint64_t)(randpool[randpos++] & 0xff)) << 040;
    *r |= ((uint64_t)(randpool[randpos++] & 0xff)) << 030;
    *r |= ((uint64_t)(randpool[randpos++] & 0xff)) << 020;
    *r |= ((uint64_t)(randpool[randpos++] & 0xff)) << 010;
    *r |= ((uint64_t)(randpool[randpos++] & 0xff));

    return;
}

```

### Фрагмент коду 1 – Випадкова генерація байтів

```

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "param.h"
#include "poly.h"
#include "../common/fastrandombytes.h"
#include "../common/crypto_hash_sha512.h"

/*
 * memory requirement: 6 ring elements
 */
void

```

```

keygen(
    uint16_t *F,      /* output secret key f */
    uint16_t *g,      /* optional output secret key g */
    uint16_t *h,      /* output public key h */
    uint16_t *buf,
    const PARAM_SET *param)
{
    int16_t i;
    uint16_t *f;
    uint16_t *f_inv;
    uint16_t *localbuf;

    f = buf;
    f_inv = f + param->padN;
    /* three ring elements for karatsuba */
    localbuf = f_inv + param->padN;

    do{
        /* generate f = pF+1 until f is invertible mod 2*/
        trinary_poly_gen(F, param->N, param->d);
        for (i=0;i<param->N;i++)
            f[i] = param->p*F[i];
        f[0]++;
    }while (ntru_ring_inv(f, param->N, localbuf, f_inv) == -1);

    /* compute f^-1 by lifting f_inv mod 2 to f_inv mod q*/
    ring_lift_inv_pow2(f_inv, f, param, localbuf);

    /* generate g*/
    trinary_poly_gen(g, param->N, param->d);

    for (i=0;i<param->N;i++)
    {
        f[i] = f[i] & 0x7FF;
        g[i] = g[i] & 0x7FF;
    }
}

```

```

/* compute  $h = f^{-1}g$  */
ntru_ring_mult_coefficients(f_inv, g, param, localbuf, h);

memset(buf, 0, sizeof(uint16_t)*param->padN*6);
return;
}

/*
 * memory requirement: 5 ring elements
 * checking if  $h = g/f$ 
 */
int check_keys(
    const uint16_t *F,
    const uint16_t *g,
    const uint16_t *h,
    uint16_t *buf,
    const PARAM_SET *param)
{
    int16_t i;
    uint16_t *f, *grec, *localbuf;

    memset(buf, 0, sizeof(uint16_t)*param->padN*5);
    f = buf;
    grec = f + param->padN;
    localbuf = grec + param->padN;

    for (i=0;i<param->N;i++)
        f[i] = F[i]*param->p;
    f[0]++;
    ntru_ring_mult_coefficients(f, h, param, localbuf, grec);

    for(i=0;i<param->N;i++)
    {
        if (grec[i]!=g[i])
        {
            printf("checking keys error for %dth coefficients: %d %d\n", i,
grec[i], g[i]);
            return -1;
        }
    }
}

```

```

    }
}

memset(buf, 0, sizeof(uint16_t)*param->padN*5);
return 0;
}

/* check if message is a valid trinary poly for kem */
int
check_m (
    const uint16_t *m,
    const uint16_t N)
{
    uint16_t i;
    for(i=0;i<N;i++)
    {
        if(m[i]!=1 && m[i]!=65535 && m[i]!=0)
        {
            printf("checking message error for %dth coefficients: %d \n", i,
m[i]);
            return -1;
        }
    }
    return 0;
}

/*
* memory requirement: 5 ring elements
*/

int encrypt_kem(
    const uint16_t *m,      /* input binary message */
    const uint16_t *h,      /* input public key */
    uint16_t *c,           /* output ciphertext */
    uint16_t *buf,
    const PARAM_SET *param)
{

```

```

if (check_m(m, param->N) == -1 )
{
    printf("error message\n");
    return -1;
}
uint16_t    i;
uint16_t    *r, *t, *localbuf;

r           = buf;
t           = r + param->padN;
/* three ring elements for karatsuba */
localbuf    = t + param->padN;

trinary_poly_gen(r, param->N, param->d);

ntru_ring_mult_coefficients(r, h, param, localbuf, t);

for (i=0;i<param->N;i++)
    c[i] = (t[i]*param->p + m[i]) & (param->q-1);

memset(buf, 0, sizeof(uint16_t)*param->padN*5);
return 0;
}

/*
 * lift the message back to a trinary polynomial
 * by mod q then mod p
 */
static void
lift_msg(
    uint16_t    *m,
    PARAM_SET   *param)
{
    uint16_t    i;
    int         tmp;
    for (i=0;i<param->N;i++)
    {

```

```

    tmp = m[i] % param->q;
    if (tmp > param->q/2)
        tmp -= param->q;

    tmp = tmp % param->p;
    if (tmp == 2)
        tmp = -1;
    if (tmp == -2)
        tmp = 1;
    m[i] = tmp;
}

return;
}

/*
 * memory requirement: 4 ring elements
 */
int decrypt_kem(
    uint16_t *m, /* output trinary message */
    uint16_t *F, /* input public key */
    uint16_t *c, /* input ciphertext */
    uint16_t *buf,
    PARAM_SET *param)
{
    uint16_t *f, *localbuf, i;

    f = buf;
    localbuf = f + param->padN;

    for (i=0; i<param->N; i++)
        f[i] = F[i]*param->p;
    f[0]++;

    /* compute e = c * f */

```

```

ntru_ring_mult_coefficients(c, f, param, localbuf, m);

/* recover  $m = e \bmod p$  */
lift_msg(m, param);

memset(buf, 0, sizeof(uint16_t)*param->padN*4);
return 0;
}

/*
 * check if a message length is valid for ntruencrypt-cca
 * then convert the message into a binary polynomial and
 * pad the message with a random binary string p
 */
int
pad_msg(
    uint16_t *m, /* output message */
    const char *msg, /* input message string */
    const size_t msg_len, /* input length of the message */
    const PARAM_SET *param)
{
    if (msg_len > param->max_msg_len)
    {
        printf("error: message too long");
        return -1;
    }
    uint16_t *pad;
    uint16_t i, j;
    char tmp;
    memset(m, 0, sizeof(uint16_t)*param->N);

    /* generate the pad of a degree 167 trinary polynomial*/
    pad = m + param->N - 167;
    trinary_poly_gen(pad, 167, 56);

    /* convert the message length into coefficients */

```

```

    pad -= 8;
    tmp = msg_len;
    for(j=0;j<8;j++)
    {
        pad[j] = tmp & 1;
        tmp >>= 1;
    }
    /* form the message binary polynomial */
    for (i=0;i<msg_len;i++)
    {
        tmp = msg[i];
        for(j=0;j<8;j++)
        {
            m[i*8+j] = tmp & 1;
            tmp >>= 1;
        }
    }
    return 0;
}

/*
 * converting a binary polynomial into a char string
 * return the length of the message string
 */
int
recover_msg(
    char      *msg,      /* output message string */
    const uint16_t *m,    /* input binary message */
    const PARAM_SET *param)
{
    char      tmp;
    int      msg_len;
    uint16_t i,j;
    msg_len = 0;

    for (j=0;j<8;j++)
    {

```

```

        msg_len += (m[param->N - 167 - 8 + j]<<j);
    }

    if (msg_len > param->max_msg_len)
    {
        printf("error: message too long");
        return -1;
    }

    for (i=0;i<msg_len;i++)
    {
        tmp = 0;
        for (j=0;j<8;j++)
        {
            tmp += (m[i*8+j]<<j);
        }
        msg[i] = tmp;
    }
    return msg_len;
}

/*
 * generate a balanced trinary r from msg and h
 * memory requirement: 2 * LENGTH_OF_HASH
 */
int
generate_r(
    uint16_t *r,      /* output r */
    const uint16_t *msg, /* input binary message */
    const uint16_t *h,  /* input public key */
    uint16_t *buf,
    const PARAM_SET *param)
{
    uint16_t i;
    for (i=0;i<param->N;i++)
    {

```

```

    if (msg[i]!=0 && msg[i]!=1 && (msg[i]%param->q)!=param->q-1)
    {
        printf("invalid messages\n");
        return -1;
    }
}
unsigned char *seed = (unsigned char*) buf;
memset(seed, 0, sizeof(unsigned char)* LENGTH_OF_HASH*2);

/* hash message/public key into a string 'seed'*/
crypto_hash_sha512(seed, (unsigned char*)msg, param->N*2);
crypto_hash_sha512(seed+LENGTH_OF_HASH, (unsigned char*)h, param->N*2);

/* use the seed to generate r */
trinary_poly_gen_w_seed(r, param->N, param->d, seed, LENGTH_OF_HASH*2);

memset(seed, 0, sizeof(unsigned char)* LENGTH_OF_HASH*2);

return 0;
}

/*
 * input a message msg, output msg \trixor hash(rh)
 * memory requirements: LENGTH_OF_HASH + 1 ring element
 */
int
mask_m(
    uint16_t *msg, /* in/output binary message */
    const uint16_t *rh,
    uint16_t *buf,
    const PARAM_SET *param)
{
    unsigned char *seed;
    uint16_t *mask;
    uint16_t i;

```

```

    memset(buf,          0,          sizeof(uint16_t)*param->padN          +
sizeof(char)*LENGTH_OF_HASH);
    seed = (unsigned char*) buf;
    mask = (uint16_t *) (seed + LENGTH_OF_HASH);

    crypto_hash_sha512(seed,      (unsigned      char*)      rh,      param-
>N*sizeof(uint16_t)/sizeof(unsigned char));

    rand_tri_poly_from_seed(mask, param->N, seed, LENGTH_OF_HASH);

    for (i=0;i<param->N;i++)
    {
        if (mask[i] == 65535)
            msg[i] --;
        else if (mask[i] == 1)
            msg[i] ++;

        if (msg[i] == 65534)
            msg[i] = 1;
        if (msg[i] == 2)
            msg[i] = -1;
    }
    memset(buf,          0,          sizeof(uint16_t)*param->padN          +
sizeof(char)*LENGTH_OF_HASH);
    return 0;
}

/*
 * input a message msg, output msg \trixor hash(rh)
 * memory requirements: LENGTH_OF_HASH + 1 ring element
 */
static int
unmask_m(
    uint16_t *msg, /* in/output binary message */
    const uint16_t *rh,

```

```

        uint16_t *buf,
const PARAM_SET *param)
{
    unsigned char *seed;
    uint16_t *mask;
    uint16_t i;

    memset(buf, 0, sizeof(uint16_t)*param->padN +
sizeof(char)*LENGTH_OF_HASH);
    seed = (unsigned char*) buf;
    mask = (uint16_t *) (seed + LENGTH_OF_HASH);

    crypto_hash_sha512(seed, (unsigned char*) rh, param-
>N*sizeof(uint16_t)/sizeof(unsigned char));

    rand_tri_poly_from_seed(mask, param->N, seed, LENGTH_OF_HASH);

    for (i=0;i<param->N;i++)
    {
        if (mask[i] == 65535)
            msg[i] ++;
        else if (mask[i] == 1)
            msg[i] --;

        if (msg[i] == 65534)
            msg[i] = 1;
        if (msg[i] == 2)
            msg[i] = -1;
    }
    memset(buf, 0, sizeof(uint16_t)*param->padN +
sizeof(char)*LENGTH_OF_HASH);
    return 0;
}
/*
 * CCA-2 secure encryption algorithm using NAEP
 * memory requirement: 6 ring elements
 */

```

```

void
encrypt_cca(
    uint16_t *c,      /* output ciphertext */
    const char *msg,  /* input message: a string of chars */
    const size_t msg_len, /* input the length of the message */
    const uint16_t *h, /* input public key */
    uint16_t *buf,
    const PARAM_SET *param)
{
    uint16_t i;
    uint16_t *r, *t, *m, *localbuf;

    m = buf;
    r = buf + param->padN;
    t = r + param->padN;
    localbuf = t + param->padN;

    /* pad the message */
    if (pad_msg( m, msg, msg_len, param) == -1)
        return;

    /* generate r from the message */
    if (generate_r(r, m, h, localbuf, param) == -1)
        return;

    /* compute r*h */
    ntru_ring_mult_coefficients(r, h, param, localbuf, t);
    for (i=0; i<param->N; i++)
    {
        t[i] *= param->p;
        t[i] &= (param->q-1);
    }

    /* mask the message with hash(r*h) */
    mask_m (m, t, localbuf, param);

    for (i=0; i<param->N; i++)

```

```

        c[i] = (t[i] + m[i]) & (param->q-1);

memset(buf,0, sizeof(uint16_t)*param->padN*6);

return ;
}

/*
 * CCA-2 secure encryption algorithm using NAEP
 * return the length of the message
 * memory requirement: 7 ring elements
 */
int decrypt_cca(
    char          *msg, /* output message: a string of chars */
    const uint16_t *F,  /* input public key */
    const uint16_t *h,  /* input public key */
    const uint16_t *c,  /* input ciphertext */
    uint16_t      *buf,
    const PARAM_SET *param)
{
    uint16_t i, msg_len;
    uint16_t *f, *m, *t, *r, *t_rec, *localbuf;

    memset(buf, 0, sizeof(int16_t)*param->padN*8);

    f          = buf;
    m          = f      + param->padN;
    t          = m      + param->padN;
    r          = t      + param->padN;
    t_rec      = r      + param->padN;
    localbuf   = t_rec  + param->padN;

    for (i=0;i<param->N;i++)
        f[i] = F[i]*param->p;
    f[0]++;

```

```

/* compute e = c * f */
ntru_ring_mult_coefficients(c, f, param, localbuf, m);

/* recover m = e mod p */
lift_msg(m, param);

/* recover r*h */
for (i=0;i<param->padN;i++)
    t[i] = (c[i] - m[i]) & (param->q-1);

/* unmask m with hash(r*h) */
unmask_m (m, t, localbuf, param);

/* recover r from hash(m) */
if (generate_r(r, m, h, localbuf,param) == -1)
{
    memset(buf,0, sizeof(uint16_t)*param->padN*7);
    return -1;
}

/* check if recovered r is correct */
ntru_ring_mult_coefficients(r, h, param, localbuf, t_rec);

for(i=0;i<param->N;i++)
{
    if (((param->p*t_rec[i] - t[i]) & (param->q-1)) !=0)
    {
        printf("error: \n");
        printf("r: \n");
        for (i=0;i<param->padN;i++)
            printf("%d, ", r[i]);
        printf("\n");
        printf("h: \n");
        for (i=0;i<param->padN;i++)
            printf("%d, ", h[i]);
        printf("\n");
        printf("t_rec: \n");
    }
}

```

```

    for (i=0;i<param->padN;i++)
        printf("%d, ", t_rec[i]);
    printf("\n");
    printf("t: \n");
    for (i=0;i<param->padN;i++)
        printf("%d, ", t[i]);
    printf("\n");
    printf("c: \n");
    for (i=0;i<param->padN;i++)
        printf("%d, ", c[i]);
    printf("\n");

    memset(buf,0, sizeof(uint16_t)*param->padN*8);
    return -1;
}

}

/* convert the message polynomial into char string */
msg_len = recover_msg(msg, m, param);
memset(buf,0, sizeof(uint16_t)*param->padN*8);
return msg_len;
}

```

**Фрагмент коду 2 – Алгоритм несиметричного шифрування NTRU-KEM**

ДОДАТОК Б  
(обов'язковий)  
СЛАЙДИ ПРЕЗЕНТАЦІЇ



Рисунок Б.2 – Слайд 1

## 2 МЕТА РОБОТИ

---

- Об'єкт дослідження є пост-квантумні криптографічні алгоритми та їх особливості для операції шифрування.
- Предметом дослідження є спосіб захисту від алгоритму обчислення дискретного логарифму за допомогою квантових комп'ютерів
- Метою роботи є дослідження методів та алгоритмів захисту від квантових процесорів.
- Методи рішення базуються на дослідженні пост-квантумної криптосистем NTRUEncrypt.

Рисунок Б.2 – Слайд 2

## 3 ПОСТАНОВКА ЗАДАЧІ

---

- Дослідити алгоритми пост-квантового шифрування;
- Провести огляд алгоритмів та вибрати алгоритм для поглибленого вивчення;
- Визначити технологій програмування, що будуть покладені в основу розробки системи;
- Моделювання алгоритму NTRU;

Рисунок Б.2 – Слайд 3

## 4 ПОСТАНОВКА ЗАДАЧІ

---

Результатом дослідження має бути:

- Реалізація генерації ключів;
- Досліджено криптосистему NTRU;
- Програмна реалізація алгоритму;
- Виведення нових методів шифрування;

Рисунок Б.2 – Слайд 4

## КЛАСИФІКАЦІЯ АЛГОРИТМІВ ШИФРУВАННЯ

### 5

---

- Шифрування – кодування інформації, після якого її неможливо буде прочитати без спец ключа, – зможе захистити ваші данні від атак.
- Симетричне шифрування – це метод, за якого ключі шифрування і розшифрування або однакові, або легко вивольяться один з одного, забезпечуючи таким чином спільний ключ, який є таємним.
- Несиметричне або асиметричне шифрування – набір методів криптографічного шифрування, в яких використовують два ключі - таємний(приватний) і відкритий; жоден із ключів не може бути обчислений з іншого за прийнятий час. Таке шифрування ще називають шифруванням з відкритим ключем

Рисунок Б.2 – Слайд 5

## РОЗГЛЯД АСИМЕТРИЧНИХ МЕТОДІВ ШИФРУВАННЯ

6

Асиметричні методи дозволяють:

- Розповсюджувати ключі по відкритим каналам зв'язку;
- Генерувати ключі для симетричного шифрування(інкапсуляція ключа).

Рисунок Б.2 – Слайд 6

## 7 АЛГОРИТМИ СТВОРЕННЯ СУЧАСНИХ АСИМЕТРИЧНИХ АЛГОРИТМІВ

Параметри:

- Наприклад,  $a$ ,  $q$  – великі числа або поліноми;
- Особистий ключ(ключ розшифрування  $d$ ) велике число, може бути, простим;
- Відкритий ключ обчислюється по формулі  $e = a^d \bmod q$ .

Рисунок Б.2 – Слайд 7

## 8 АЛГОРИТМИ СТВОРЕННЯ СУЧАСНИХ АСИМЕТРИЧНИХ АЛГОРИТМІВ

---

Атака:

- Для визначення особистого ключа по відкритому:
  - $d = \log_a e \text{ mod } q$ .

Тобто треба визначити логарифм у цілих числах по модулю.

Для стандартного комп'ютера ці обчислення потребують дуже багато часу, але для квантового це займе декілька годин.

Рисунок Б.2 – Слайд 8

## КВАНТОВІ ОБЧИСЛЕННЯ

---

9

- Кубіт – одиниця квантової інформації, квантовий аналог біта. Дворівнева квантовомеханічна система, наприклад, поляризація окремого фотона, яка може бути вертикальною або горизонтальною. В класичній системі біт завжди прийматиме одне з двох значень, але квантова механіка дозволяє кубітові перебувати в стані суперпозиції. Ця властивість кубіта є базисом для всієї теорії квантових обчислень.

Рисунок Б.2 – Слайд 9

## 10 КВАНТОВІ ОБЧИСЛЕННЯ

---

Якщо в класичному комп'ютері найменша одиниця інформації представляється бітом, який може набувати значення або 0, або 1 в одно час, то в квантовому цю роль виконують кубіти. Їх особливість полягає в тому, що кубіт може знаходитися і в змозі 0, і в змозі 1 одночасно. Це і дає квантовим комп'ютерам їх неймовірну обчислювальну потужність. Наприклад, якщо ми розглядаємо чотири біти інформації, то зі всіляких 16 станів ми можемо вибрати лише одно водночасу. 4 кубіта ж можуть знаходитися в 16 станах одночасно, тобто в суперпозиції, і ця залежність росте експоненційно з кожним новим кубітом.

Рисунок Б.2 – Слайд 10

## 11 КВАНТОВІ ОБЧИСЛЕННЯ

---

Якщо в класичному комп'ютері логічні елементи отримують на вхід біти інформації, а на виході видають однозначно певний результат, то в квантовому комп'ютері в якості логічного елемента береться так званий квантовий гейт (quantum gate), який маніпулює значенням цілої суперпозиції. Важливе явище, властиве кубітам, – це заплутаність. Наприклад, маємо два заплутаних кубіта. Вимір стану одного з них допоможе дізнатися інформацію про стан його пари без необхідності якої-небудь перевірки.

Рисунок Б.2 – Слайд 11

## 12 ГРАТКИ

---

Серед методів шифрування, захищених від квантових атак, є також популярні алгоритми на ґратках. Задачі теорії ґраток — це клас задач оптимізації на ґратках (тобто дискретних адитивних підгрупах, заданих на  $R^n$ ). Труднощі при розв'язуванні таких задач є центральним місцем для побудови стійких криптосистем на ґратках. Для додатків в таких криптосистемах зазвичай розглядаються ґратки на векторних просторах (часто  $Q^n$ ) або вільних модулях (часто  $Z^n$ ).

Для всіх задач нижче припустимо, що нам дано (крім інших більш конкретних вхідних даних) базис для векторного простору  $V$  і норма  $N$ . Для норм зазвичай розглядається  $L^2$ . Однак інші норми, такі як  $L^p$ , також розглядаються і з'являються в різних результатах. Нижче в статті  $\lambda L$  означає довжину найкоротшого вектора в ґратці  $L$ , тобто  $\lambda L = \min_{v \in L \setminus \{0\}} \|v\|_N$ .

Рисунок Б.2 – Слайд 12

## 13 ГРАТКИ

---

Задачі знаходження найкоротшого вектору ґратки. Для ґратки  $L$  дані базис векторного простору  $V$  і норма  $N$ , потрібно знайти ненульовий вектор мінімальної довжини в  $V$  за нормою  $N$  в ґратці  $L$ . Іншими словами, виходом алгоритму повинен бути ненульовий вектор  $v$ , такий, що  $N(v) = \lambda(L)$ .

В  $\gamma$ -наближеній версії знаходження найкоротшого вектору потрібно знайти ненульовий вектор ґратки довжини, що не перевищує  $\gamma\lambda(L)$ .

Рисунок Б.2 – Слайд 13

## 14 ДЛЯ ПРАКТИЧНОГО ДОСЛІДЖЕННЯ ОБРАНО АЛГОРИТМ NTRU

---

Генерація ключів:

Пара ключів повинна бути сформована із використанням наступного або математично еквівалентного набору кроків. Звертаю увагу, що наведений нижче алгоритм виводить тільки значення  $f$  і  $H$ . У деяких додатках може бути бажано зберегти ці значення  $f$ - $l$  і  $g$  теж. Цей стандарт не визначає формат виведення ключа до тих пір, поки він є однозначним.

Компоненти: параметри  $N, q, p, d, F, D, g$ .

Вхідні дані: рядок  $s$  як джерело випадало.

Вихід: пару ключів, що складається із закритого ключа  $i$  і відкритого ключа  $h$ .

Рисунок Б.2 – Слайд 14

## 15 ДЛЯ ПРАКТИЧНОГО ДОСЛІДЖЕННЯ ОБРАНО АЛГОРИТМ NTRU

---

Генерація ключів:

Операція: пара ключів повинна бути розрахована за допомогою наступної або еквівалентної послідовності кроків:

Встановіть многочлен  $F := 0$ ;

Вмикаємо генератор поліномів фіксованого навантаження;

Обчислити многочлен  $f := 1 + p \times F$ ;

Обчислити поліном  $f$ - $l$  в кільці. Якщо  $f$ - $l$  не існує, перейдіть до першого кроку

Викликаємо генератор поліномів фіксованого навантаження із вхідними даними  $N, D, g, s$  для отримання  $a$  многочлену  $G$ ;

Обчислити многочлен  $h := f \cdot l \times g \times r$  в кільці;

Вихід  $F, h$ .

Рисунок Б.2 – Слайд 15

## 16 ДЛЯ ПРАКТИЧНОГО ДОСЛІДЖЕННЯ ОБРАНО АЛГОРИТМ NTRU

Операція інкапсуляції:

Опис операції інкапсуляції. Операція інкапсуляції інкапсулює кільцевий елемент. Він не перевіряє достовірність цього елемента. Виконавець повинен забезпечити достатню ентропію в цьому елементі на рівні протоколу.

Компоненти:

Довжина буфера кодування,  $\text{bufferLenBits}$ ;

Повідомлення  $m$ , яке являє собою тринарний поліном;

Відкритий ключ  $h$ ;

Рядок  $s$ , як джерело випадковості.

Результатом виконання алгоритму є зашифрований текст  $e$ , який формується як поліном (кільцевий елемент).

Операція така, що шифротекст  $e$  повинен бути розрахований за наступною або еквівалентною послідовністю кроків:

Використовуйте функцію генерації поліномів фіксованим навантаженням з  $s$  і параметрами  $N, dr$  для отримання  $g$ ;

Обчислити  $R = g \times h \bmod q$ ;

Обчисліть шифротекст, як  $e = R + M \bmod q$ ;

Вихід  $e$ .

Рисунок Б.2 – Слайд 16

## ДЛЯ ПРАКТИЧНОГО ДОСЛІДЖЕННЯ ОБРАНО АЛГОРИТМ NTRU

17

Операція декапсуляції:

Компоненти відсутні.

Вхідні дані:

Шифротекст  $e$ , що представляє собою многочлен ступеня  $N-1$ ;

Закритий ключ  $F$ ;

Відкритий ключ  $H$ .

Вихід: елемент із кільця поліномів.

Операція: повідомлення  $m$  має бути розраховане за наступною або еквівалентною послідовністю кроків:

Обчислити  $f = PF + I$ ;

Обчислити  $cm = f \times e \bmod p$ ;

Вхідний  $CM$ , як розшифровка повідомлення  $M$ .

Рисунок Б.2 – Слайд 17

## 18 ВИСНОВКИ

---

- У ході виконання атестаційної роботи було досліджено проблему захисту від квантових комп'ютерів. Були розглянуті алгоритми захисту такі як: NTRU-KEM, NTRU-CCA. Також були розглянуті особливості квантових алгоритмів, та проблеми квантових обчислень на яких базується постквантовий захист.
- Були проведені детальні аналізи криптосистеми NTRUЕncгурт, її переваги та недоліки перед конкурентними алгоритмами. у ході свого дослідження було зрозуміло, що дана криптосистема знаходиться попереду перед іншими системами та алгоритмами шифрування, і є сама по собі достатньо новою і оновлюється із кожним роком.
- Що до захищеності даної системи: NTRU, пристосована для захисту ключа від майже усіх нині можливих атак. Стійкість алгоритму забезпечена складністю пошуку найкоротшого вектору ґратки, яка більш стійка до атак, здійснюваних на квантових комп'ютерах, була створена для заміни своїх попередників RSA та еліптичних кривих.

Рисунок Б.2 – Слайд 18

---

ДЯКУЮ ЗА УВАГУ!

Рисунок Б.2 – Слайд 19