Содержание

Введение

Современная криптография в значительной степени опирается на задачи, которые считаются трудноразрешимыми для классических компьютеров. Алгоритмы, такие как RSA и криптография на эллиптических кривых (ECC), основаны на сложности факторизации больших чисел и вычислении дискретного логарифма соответственно. Эти подходы обеспечивают высокую степень безопасности при разумных размерах ключей и на протяжении десятилетий оставались основой цифровой безопасности в интернете, финансовых системах, мобильных устройствах и других областях.

Однако в последние годы активное развитие квантовых вычислений поставило под угрозу криптографические методы, устойчивые лишь к классическим атакам. Алгоритмы Шора и Гровера, разработанные для квантовых компьютеров, в теории позволяют эффективно решать задачи факторизации и дискретного логарифмирования. Это делает такие схемы, как RSA и ECC, уязвимыми при наличии достаточно мощного квантового компьютера. Несмотря на то, что практическое применение квантовых машин всё ещё ограничено, интенсивные исследования в этой области создают необходимость разработки криптографических алгоритмов, стойких к атакам с использованием квантовых технологий.

Одним из направлений постквантовой криптографии являются криптосистемы, основанные на задачах из области алгебры и теории решёток. В частности, алгоритм NTRU (Nth-degree Truncated Polynomial Ring Units) представляет собой перспективную альтернативу классическим схемам благодаря высокой эффективности и предполагаемой стойкости как к классическим, так и к квантовым атакам. В отличие от RSA и ECC, он базируется на трудности поиска ближайшего вектора в решетке — задачи, для которой не известно эффективных квантовых алгоритмов.

Цель данной курсовой работы — исследование криптосистемы NTRU и ее сравнение с RSA, ECC с точки зрения вычислительной сложности, производительности и применимости в современных системах защиты информации. Также будет рассмотрена безопасность исходной системы NTRU и возможные модификации данной системы.

1 АНАЛИЗ ВНУТРЕННЕЙ СТРУКТУРЫ КРИПТОСИСТЕМЫ

В данном разделе будет представлено описание алгоритма вместе с математическими принципами, на основе которых он работает. Также будут кратко описаны известные на данный момент атаки и возможные модификации, которые впоследствие будут применены в программной реализации.

1.1 Кольцо усеченных полиномов

Алгоритм NTRUEncrypt использует операции над кольцом $\mathbb{Z}[X]/(X^N - 1)$ усеченных полиномов степени, не превосходящей $N - 1$ [4]:

$$\mathbf{a}(X) = a_0 + a_1X + a_2X^2 + \cdots + a_{N-2}X^{N-2} + a_{N-1}X^{N-1}$$

В рамках данного кольца справедливо, что $X^N = 1$. Таким образом умножение полинома на X циклически сдвигает коэффициенты: коэффициент при X^1 становится коэффициентом при X^2 и так далее. Также немаловажно, что операции проводятся над кольцом $\mathbb{Z}_q[X]/(X^N - 1)$, где $\mathbb{Z}_q$ - кольцо целых чисел по некоторому модулю q.

Для NTRU определяется три целых параметра (N, p, q). N - степень, определяющая размерность кольца, p - малый модуль и q - больший модуль. Обычно за N принимают простое число, а q всегда намного больше p, и q и p взаимнопросты (НОД$(p, q) = 1$). Открытый текст в данной криптосистеме - это полином по модулю p (обозначим его за $\mathbf{m}$), а шифротекст - полином по модулю q (обозначим его за $\mathbf{e}$).

Правильный подбор параметров кольца играет важную роль в работе алгоритма и обеспечении безопасности. Как правило степень N берут достаточно большую и при этом простую, малый модуль p обычно берут равным 3 или полиному $X + 2$, что используют в оптимизированных версиях NTRU [2], а большой модуль q равен некоторой степени двойки для быстроты вычислений. Большая степень N обеспечивает достаточную безопасность алгоритма и сложность подбора параметров, но при этом слишком больше значение будет замедлять работу алгоритма.

В публикации NTRU NIST 2020 [9] года были предложены параметры для обеспечения безопасной работы алгоритма. Параметры представлены в таблице 1.

	N	**p**	**q**
128 bit security margin (NTRU-HPS)	509	2048	3
192 bit security margin (NTRU-HPS)	677	2048	3
256 bit security margin (NTRU-HPS)	821	4096	3
256 bit security margin (NTRU-HPS)	701	8192	3

Таблица 1: Предложенные параметры NTRU

1.2 Работа алгоритма

Для работы алгоритма требуется пара: открытый и закрытый ключи. Для генерации ключей требуется два полинома $\mathbf{f}$ и $\mathbf{g}$ с коэффициентами, принадлежащими мно-

жеству $\{-1, 0, 1\}$. Полином $\mathbf{f}$ должен быть обратим по модулям p и q, то есть $\mathbf{f} \cdot \mathbf{f}_p = 1 \pmod{p}$ и $\mathbf{f} \cdot \mathbf{f}_q = 1 \pmod{q}$. Если подобранный полином не удовлетворяет данному условию, подбирается новый. Дополнительные условия к данным полиномам подразумевают, что $\mathbf{f}(1) = 1$ и $\mathbf{g} = 0$, то есть количество единиц в полиноме $\mathbf{g}$ равно количеству минус единиц, а количество единиц в полиноме $\mathbf{f}$ на один больше, чем количество минус единиц.

Закрытым ключем является пара полиномов $\mathbf{f}$ и $\mathbf{f}_p$, а открытый ключ $\mathbf{h}$ вычисляется следующим образом:

$$\mathbf{h} = p \cdot \mathbf{f}_q \cdot \mathbf{g} \pmod{q}$$

Имея открытый ключ $\mathbf{h}$ и открытый текст $\mathbf{m}$, а также случайный полином $\mathbf{r}$, информация шифруется следующим образом:

$$\mathbf{e} = \mathbf{m} + \mathbf{h} \cdot \mathbf{r} \pmod{q}$$

Расшифровка информации происходит в несколько этапов, при этом важно, что коэффициенты полиномов по модулю q находятся в пределах $[-q/2;\ q/2 - 1]$, а по модулю p - в пределах $[-(p-1)/2;\ (p-1)/2]$, иначе результат может быть некорректным. Во-первых, вычисляется полином $\mathbf{a}$ как произведение полиномов $\mathbf{e}$ и $\mathbf{f}$ по модулю q.

$$\mathbf{a} = \mathbf{f} \cdot \mathbf{e} \pmod{q}$$

После этого вычисляется полином $\mathbf{b} = \mathbf{a} \pmod{p}$. Наконец полином $\mathbf{b}$ умножается на $\mathbf{f}_p$. Полученный в результате полином с большой вероятностью будет являться открытым текстом.

$$\mathbf{m} = \mathbf{b} \cdot \mathbf{f}_p \pmod{p}$$

1.3 Корректность работы алгоритма

Корректность работы алгоритма можно доказать следующим образом. Умножение $\mathbf{e}$ и $\mathbf{a}$ можно расписать как:

$$\mathbf{a} = \mathbf{e} \cdot \mathbf{f} = (\mathbf{m} + \mathbf{h} \cdot \mathbf{r}) \cdot \mathbf{f} \equiv \mathbf{m} \cdot \mathbf{f} + p \cdot \mathbf{g} \pmod{q}$$

Поскольку открытый ключ получается из произведения $\mathbf{g}$ и p на $\mathbf{f}_q$, множитель $\mathbf{f}_q$ сокращается по модулю q. Далее при нахождении коэффициентов полинома по модулю p слогаемое $\mathbf{g} \cdot p$ так же сокращается. Наконец умножение $\mathbf{b}$ на $\mathbf{f}_p$ по модулю p приводит к тому, что множитель $\mathbf{f}$ так же сокращается, и по итогу мы получаем исходный текст.

$$\mathbf{b} \cdot \mathbf{f}_p = \mathbf{m} \cdot \mathbf{f} \cdot \mathbf{f}_p \equiv \mathbf{m} \pmod{p}$$

1.4 Безопасность криптосистемы

Безопасность данной криптосистемы обеспечивается за счет трудоемкости обратных преобразований, другими словами имея шифротекст и открытый ключ вычислить открытый текст будет трудно. Один из самых эффективных алгоритмов для нахождения векторов $\mathbf{f}$ и $\mathbf{g}$ является алгоритм редукции базиса решетки Ленстры-Ленстры-Ловаса, который за полиномиальное время может найти кратчайший вектор решетки.

Однако при достаточно больших параметрах вычисления становятся слишком трудоемкими.

Другой вектор атаки может быть полный перебор, поскольку вектор $\mathbf{f}$ содержит малые коэффициенты. Злоумышленник может воспользоваться этим и попытаться перебрать значения $\mathbf{f'}$, проверяя их путем вычисления $\mathbf{f'} \cdot \mathbf{h} \pmod q$. Если данное произведение дает малые коэффициенты, то злоумышленник может попробовать расшифровать полученным вектором собственное зашифрованное сообщение. Сократить время вычислений позволяет атака встречи посередине.

Более существенной проблемой алгоритма NTRU, примененного без каких-либо модификаций, является атака по подобранному шифротексту [3] и возможность определения связи между шифротекстом и исходным текстом [6]. Первая атака позволяет скомпрометировать закрытый ключ, используя некорректные параметры шифрования, а вторая - определять, какая информация была зашифрована, то есть при данных двух открытых текстах и шифротексте можно будет определить, какой открытый текст был зашифрован. Другими словами, не обеспечена семантическая безопасность алгоритма.

Атака по подобранному шифротексту может быть реализована через попытку дешифровать полином $\mathbf{h} \cdot c + c$, где c является целыми числом таким, что $c = 0 \pmod p$, $c < \frac{q}{2}$ и $2c > \frac{q}{2}$. Далее дешифровка данного сообщения предполагает умножение на полином $\mathbf{f}$.

$$\mathbf{a} = \mathbf{f} \cdot c \cdot \mathbf{h} + c \cdot \mathbf{f} \equiv c \cdot \mathbf{g} + c \cdot \mathbf{f} \pmod q$$

Таким образом, поскольку полиномы $\mathbf{g}$ и $\mathbf{f}$ состоят из малых коэффициентов $-1, 0, 1$, полученный полином $\mathbf{a}$ будет состоять из коэффициентов, равных $-2c$, c, 0, c или $2c$. Поскольку мы подобрали c таким образом, что $c < \frac{q}{2}$ и $2c > \frac{q}{2}$, нам остается редуцировать только коэффициенты $2c$ и $-2c$.

Если предположить, что существует единственный коэффициент $\mathbf{a}$, равный $\pm 2c$, например, $a_i = +2c$, то значение $\mathbf{a}$ можно представить как $c \cdot \mathbf{f} + c \cdot \mathbf{g} - qx^i$. Таким образом, после умножения $\mathbf{a}$ на $\mathbf{f}_p$ мы получим:

$$c \cdot \mathbf{g} \cdot \mathbf{f}_p + c - qx^i \cdot \mathbf{f}_p \pmod p$$

Так как $c = 0 \pmod p$ мы получаем

$$-qx^i \cdot \mathbf{f}_p \pmod p$$

Поскольку НОД$(p, q) = 1$, мы можем получить $x^i \cdot \mathbf{f}_p \equiv x^i / \mathbf{f} \pmod q$ и вычислить его обратное значение $\mathbf{f}/x^i \pmod q$. Если коэффициенты $\mathbf{f}$ лежат в пределах $-1, 0, 1$, то мы получили настоящее значение $\mathbf{f}$. Далее вычислить $\mathbf{g}$ можно с помощью следующей формулы:

$$\mathbf{g}/x^i = \mathbf{h} \cdot \mathbf{f}/x^i \pmod q$$

Поскольку умножение на x^i приводит только к циклическому сдвигу коэффициентов, мы получаем корректные ключи.

Данная атака однако подразумевает, что найдутся такие два полинома $\mathbf{f}$ и $\mathbf{g}$, что у них только один коэффициент будет совпадать. На практике вероятность такого не очень высока. Модифицировать данную атаку можно тем, чтобы заменить qx^i на полином, который будет уменьшать результат дешифрования. Коэффициенты полинома будет подобрать весьма просто, если совпадающих коэффициентов у $\mathbf{f}$ и $\mathbf{g}$ мало.

1.5 Модификации системы

Существует ряд возможных модификаций для данной криптосиситемы, от тех, что просто ускоряют работа алгоритма и облегчают подбор ключей, до тех, что увеличивают безопасность.

1.5.1 Оптимизация и ускорение

С целью оптимизации и ускорения работы алгоритма могут быть сделаны следующие модификации:

- Выбрать в качесвте параметра p полином вместо малого числа, взаимно простого с q (например, $\mathbf{p} = X + 2$). При этом важно, что идеал, порождаемый полиномом, должен быть взаимно прост с идеалом $\langle q \rangle$, порожденным q. Данная модификация позволяет более простую кодировку сообщений и более высокую вероятность корректной расшифровки.
- Модифицировать алгоритм подбора полинома $\mathbf{f}$ с целью облегчить поиск таких полиномов, которые будут мультипликативно обратны по модулям p и q. Вместо того, чтобы подбирать полином $\mathbf{f}$ степени $n-1$ подбирается полином $\mathbf{F}$ степени $n-2$, и при этом $\mathbf{F}(1) = 0$. Далее вычисляется $\mathbf{f} = 1 + 3 \cdot \mathbf{F}$. Данная модификация, однако, снижает безопасность алгоритма либо требует подбора параметров, которые увеличивают затраты ресурсов. Для данной схемы потребуется, чтобы $q > 12d + 1$, где d - количество единиц и количество минус единиц в полиноме $\mathbf{F}$.

1.5.2 Безопасность

Для повышения безопасности в систему NTRU могут быть добавлены padding. Было предложено множество форматов padding, однако в рамках этой работы я остановлюсь на рассмотрении padding, основанной на схеме OAEP, адаптированной под параметры $N = 821$, $q = 4096$ и $p = 3$. Для данной схемы потребуются три хэш-функции: $H : \{0,1\}^{\text{mLen}} \to \{0,1\}^{\text{rLen}}$, $F : \{0,1\}^{k_1} \to \{0,1\}^{k_2}$ и $G : \{0,1\}^{k_2} \to \{0,1\}^{k_1}$, где mLen - длина шифруемой информации, rLen - длина информации, составляющей ослепляющий полином, k_1 - длина сообщения M в битовом представлении и k_2 - длина рандомной строки R в битовом представлении. Далее для этой схемы вычисляются следующие строки: $s = M \oplus G(R)$ и $t = R \oplus F(s)$, а также хэш $H(M \,||\, R)$. Далее информация шифруется следующим образом: $E_{pk}(s \,||\, t; H(M \,||\, R))$. Данная схема padding обеспечивает семантическую безопасность, так как открытый текст маскируется и конкатенируется со случайной маскированной строкой, а хэш-функция H усиливает защищенность алгоритма от атак по подобранному шифротексту.

Другие модификации могут в себя включать защиту от утечек по сторонним каналам, однако NTRU не использует действительно трудоемкие операции вроде возведения в степень, так что алгоритм не будет выдавать себя за счет разного времени исполнения операций. Впрочем, в некоторых работах отмечается, что за счет утечек могут быть реализованы атаки с использованием подобранного шифротекста, используя информацию из каналов утечки как устройство для проверки открытого текста [8]. Но в рамках данного обзора я не буду фокусироваться на модификациях для защиты от утечек по сторонним каналам.

2 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

В данном разделе будет описана программная реализация NTRU. Будут описаны некоторые особенности реализации этой системы, а также примененные модификации. Для реализации был выбран язык C++ в силу своей высокой скорости и поддержки объектно-ориентированного программирования.

2.1 Реализация кольца усеченных полиномов

Для реализации кольца усеченных полиномов был создан специальный класс Polynomial, в рамках которого были реализованы все базовые арифметические операции с полиномом. Параметрами объекта класса Polynomial является максимальная степень deg и массив с целочисленными коэффициентами полинома pol длиной в deg.

```
class Polynomial
{
public:
    int* pol;
    int deg;

    template <size_t len>
    Polynomial(const int (&arr)[len]);
    Polynomial(const int* = NULL, int = 0);
    Polynomial(int);
    Polynomial(const Polynomial& other);
    ~Polynomial();

    string pol_to_string(int = 0);

    friend ostream& operator<<(ostream& os, const Polynomial& pol);

    Polynomial& operator=(const Polynomial&);
    int& operator[](int);

    friend Polynomial operator+(const Polynomial&, const Polynomial&);
    friend Polynomial operator+(const Polynomial&, int);
    friend Polynomial operator-(const Polynomial&, const Polynomial&);
    friend Polynomial operator*(const Polynomial&, const Polynomial&);
    friend Polynomial operator*(const Polynomial&, int);
    friend Polynomial operator*(int, const Polynomial&);
    friend Polynomial operator%(const Polynomial&, int);
};
```

Листинг 1: Описание класса Polynomial и определение методов внутри класса

Нахождение мультипликативно обратных полиномов было реализовано с помощью расширенного алгоритма Евклида. Для его работы было реализовано деление одного полинома на другой по модулю целого числа в рамках функции mod_div.

```
Polynomial mod_div(
    const Polynomial& pol1,
    const Polynomial& pol2,
    int mod)
{
    int deg = pol1.deg;
    Polynomial res(deg);
    Polynomial div = pol1;

    int deg1 = deg - 1;
    int deg2 = deg - 1;

    while (deg2 >= 0 && pol2.pol[deg2] == 0) --deg2;    // find the degree
    of the greatest non-zero coefficient
    while (deg1 >= deg2 && pol1.pol[deg1] == 0) --deg1;

    while (deg1 >= deg2)
    {
        int div_res = inverse_integer(pol2.pol[deg2], mod); // find
   inverse coefficient
        if (!div_res)
            return Polynomial(0);
        int coef = div[deg1] * div_res % mod;
        int shift = deg1 - deg2;
        res[shift] = coef;

        Polynomial xn(deg);
        xn[shift] = 1;

        div = (div - xn * pol2 * coef) % mod;
        while (deg1 >= 0 && div[deg1] == 0) --deg1;
    }

    return res % mod;
}
```

Листинг 2: Функция деления одного полинома на другой по модулю целого числа

Нахождение мультипликативно обратных полиномов по модулю $p = 3$ не требовалось при условии модификации, описанной в разделе выше, однако все еще было необходимо нахождение мультипликативно обратного полинома по модулю $q = 4096$. Базовый расширенный алгоритм Евклида с этой задачей не всегда справлялся в силу того, что модуль q не является простым числом, и не все коэффициенты меньше q были взаимно просты с ним. Для этого я решил прибегнуть к использованию леммы Хенсена, которая описывает способ "подъема"простого модуля полинома. Если у нас имеется полином $\mathbf{p}_q$, взятый из кольца $\mathbb{Z}_q[X]/(X^N - 1)$, где q - это простой модуль, то можно найти приближенное значение данного полинома по степени простого модуля q^k. Нахождение приближения происходит по следующей формуле:

$$\mathbf{p}_{q^{i+1}} = \mathbf{p}_{q^i} \cdot (2 - \mathbf{p}_{q^i} \cdot \mathbf{f}) \pmod{q^{i+1}}$$

где $\mathbf{p}_q \cdot \mathbf{f} \equiv 1 \pmod{q}$. Поскольку модуль q является степенью числа 2, данная лемма применима к моей задаче. Ниже представлена реализация данного алгоритма.

```
int eucl_inverse_mod2k(Polynomial pol, int mod, Polynomial *inv)
{
  int deg = pol.deg;
  int md2k = mod;
  while(md2k > 2)
  {
    if (md2k % 2 != 0)
    {
      cout << "Modulus must be the power of 2" << endl;
      return 1;
    }
    md2k /= 2;
  }

  Polynomial res;
  int ret = eucl_inverse_mod(pol, 2, &res); // find inverse modulo 2 via
   EEA

  if (ret)
    return 1;

  Polynomial int_2(deg);
  int_2[0] = 2;

  while (md2k < mod)
  {
    md2k *= 2;
    res = res * (int_2 - pol * res) % md2k;
  }

  *inv = res;
  return 0;
}
```

Листинг 3: Функция нахождения мультипликативно обратного по модулю степени двойки

2.2 Реализация криптосистемы

Поскольку в классе Polynomial уже реализованы функции, которые будут ответственны за шифрование и дешифрование, а также было реализовано нахождение мультипликативно обратных, оставалось только реализовать функцию генерации ключей и реализовать кодировку сообщений в виде полиномов степени p. А также был реализован OAEP-подобный padding.

Функция генерации ключей подбирает варианты малых полиномов, пока не найдет тот, что может быть обратим по модулю p и q. Также генерируется полином $\mathbf{g}$ и рассчитываются мультипликативно обратные $\mathbf{f}_q$ и $\mathbf{f}_p$. Открытый ключ $\mathbf{h}$ в дальнейшем рассчитывается по формуле.

Кодировку полиномов я решил реализовать путем представления байтов полиномов в троичном виде, чтобы потом записать триты в качестве коэффициентов полинома. Таким образом один байт сообщения может быть представлен шестью тритами или шестью коэффициентами полинома. Еще для приведения коэффициентов полинома к классу абсолютно наименьших вычетов по целому модулю была написана функция miminize. Она важна в процессе дешифровки для получения корректного результата.

```
Polynomial minimize(Polynomial poly, int mod)
{
  int deg = poly.deg;
  Polynomial res(deg);

  for (int i = 0; i < deg; i++)
  {
    if (poly[i] > (mod - mod % 2) / 2 - (mod + 1) % 2)
      res[i] = poly[i] - mod;
    else if (poly[i] < -(mod - mod % 2) / 2)
      res[i] = poly[i] + mod;
    else
      res[i] = poly[i];
  }
  return res;
}
```

Листинг 4: Функция miminize

Набивка была реализована в соответствии с описанием, данным в разделе 1. В качестве хэш-функции H была выбрана SHA256, а в качестве хэш-функции F и G - функция генерации маски с хэш-функцией SHA256, взятая из стандарта OAEP для RSA. Функция, формирующая padding, также рассчитывает значение $H(M \,||\, R)$. Затем полученные байтовые строки преобразуются в полиномы. Строку $s \,||\, t$ было решено сделать длиной 128 байт, где 96 байт занимает замаскированное сообщение M, а 32 байта - замаскированные случайные байты R. Вывод хэш-функции H было решено распределить по всей длине ослепляющего полинома, поскольку 256-битный хэш занимал бы максимум 192 коэффициента. Такая плотность коэффициентов не больше степени 192 может плохо сказаться на безопасности криптосистемы. Ниже представлена функция, формирующая padding.

```
unsigned char* padding_encode(string data, unsigned char* hash)
{
  unsigned char* enc_data = new unsigned char[129];

  unsigned char* rand = new unsigned char[32];
  randombytes_buf(rand, 32);

  unsigned char* mask = mgf1(rand, 32, 96); \\ mask generating function

  for (int i = 0; i < 96; i++)
  {
    if (i < data.size())
      enc_data[i] = data[i] ^ mask[i];
    else if (i == data.size())
      enc_data[i] = 0x01 ^ mask[i];
    else
      enc_data[i] = mask[i];
  }
  delete[] mask;

  mask = mgf1(enc_data, 96, 32);
  for (int i = 0; i < 32; i++)
    enc_data[96 + i] = rand[i] ^ mask[i];
  delete[] mask;

  enc_data[128] = '\0';

  BYTE m_r_string[128];
  for (int i = 0; i < 96; i++)
  {
    if (i < data.size())
      m_r_string[i] = data[i];
    else if (i == data.size())
      m_r_string[i] = 0x01;
    else
      m_r_string[i] = 0x00;
  }

  for (int i = 96; i < 128; i++)
    m_r_string[i] = rand[i % 32];

  delete[] rand;
  sha256(m_r_string, 128, hash);

  return enc_data;
}
```

Листинг 5: Реализация OAEP-padding

3 ТЕСТИРОВАНИЕ И СРАВНЕНИЕ КРИПТОСИСТЕМ

В данном разделе будет производиться тестирование реализованного алгоритма и его сравнение с аналогами, а также сравнение с версией с имплементированным OAEP-padding.

3.1 Тестирование реализации NTRU

Для начала протестируем работу базовой версии NTRU без каких-либо модификаций. Для удобства чтения выводятся первые 12 коэффициентов всех полиномов.

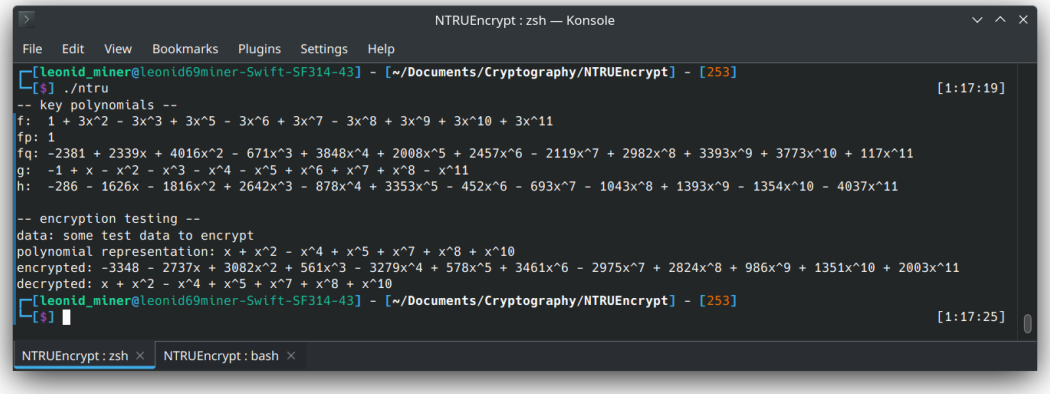

Рис. 1: Тестирование базовой версии NTRU

Теперь протестируем работу NTRU с padding на основе OAEP. Будут так же выводиться лишь первые 12 коэффициентов всех полиномов кроме ослепляющего, у которого будет выведены первые 48 коэффициентов. Также будет выведено 16-ричное представление строки $s \,||\, t$.

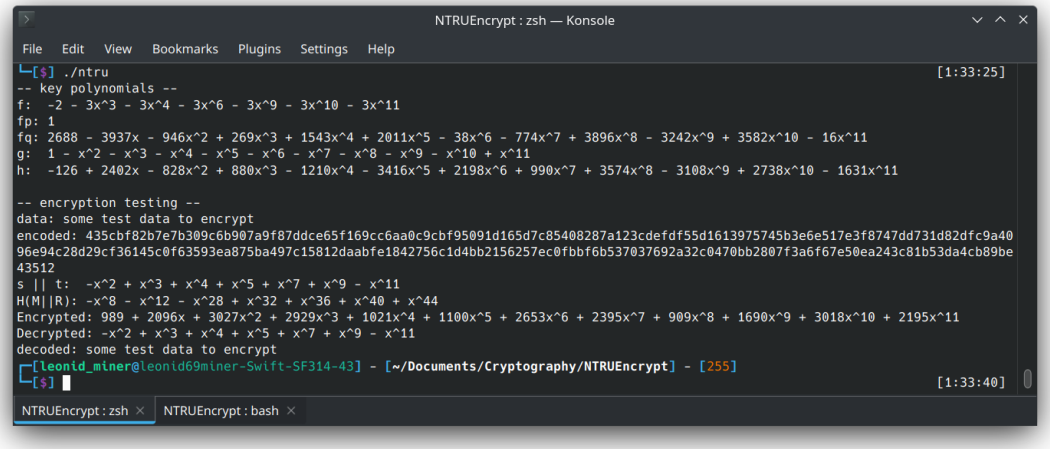

Рис. 2: Тестирование модифицированной версии NTRU

Как мы видим, алгоритм успешно справляется с задачей генерации ключей, шифрованием и дешифрованием данных. Также padding работает корректно.

3.2 Методика сравнения

Сравнение криптосистем будет производиться с точки зрения производительности, сложности вычислений, потребления ресурсов и безопасности. Для сравнения помимо криптосистемы NTRU была реализована криптосистема RSA. ECC будет тестироваться как "черный ящик"с использованием библиотеки libsodium. На практике будет тестироваться процесс генерации ключей, процесс шифрования и дешифрования и количество памяти, нужное для хранения ключей. Тажке будет сравниваться размер шифротекста относительно размера исходного сообщения. Наконец теоретически будет оценена безопасность алгоритмов.

Для цели сравнения были подобраны такие параметры криптосистем, чтобы они были максимально близки друг к другу по уровню безопасности. Уровень безопасности сравнивается с эквивалентом ключей AES (128 бит, 192 бита и 256 бит). Для NTRU были взяты параметры $N = 821$, $q = 4096$ и $p = 3$, что соответствует 256-битному уровню безопасности, следуя одной из последних публикаций NIST [9]. Для RSA такой же уровень безопасности обеспечивает размер модуля, равный 15360 битам[1], хотя на практике используются модули размера 2048 или 3072 бита. Сравниваться будет RSA с размером модулюя 2048 бита, как самый практичный и при этом обеспечивающий достаточный уровень безопасности на данный момент, и 15360 бит, как эквивалетный по уровню безопасности. Наконец в качестве алгоритма на основе эллиптических кривых будет использоваться алгоритм шифрования на основе Curve25519, который обеспечивает 128-битную защиту.

Помимо сравнения базовых версий криптосистем будет также произведено сравнение криптосистем с модификациями. Будет сравнение системы NTRU без модификаций и с padding, также будет сравнен NTRU с padding и RSA с padding так же на основе OAEP.

3.3 Практическое сравнение алгоритмов

Проведем практическое сравнение алгоритмов по времени генерации ключей, шифрования и дешифрования. Для этого использовалась следующая функция, измеряющая общее время исполнения всех итераций и среднее время выполнения одной итерации.

```cpp
void benchmark(const string& name, int iterations, void (*func)())
{
    using namespace chrono;

    auto start = high_resolution_clock::now();

    for (int i = 0; i < iterations; ++i) {
        func();
    }

    auto end = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(end - start).count();

    std::cout << name << ":\n";
    std::cout << "  Total time: " << duration << " microsecs\n";
    std::cout << "  Avg per op: " << (double)duration / iterations << "
   microsecs\n\n";
}
```

Листинг 6: Функция для измерения времени исполнения алгоритмов

3.3.1 NTRU

Функция	Среднее время выполнения, мс	Количество итераций
Генерация ключа	4273.1	100
Шифрование	2.89	1000
Дешифрование	5	1000

Таблица 2: Результаты измерений для NTRU

Можно заметить, что генерация ключей занимает значительно больше времени, чем процесс шифрования и дешифрования. Это связано с тем фактом, что в процессе перебора производится проверка на обратимость, что требует вычисления мультипликативно обратного полинома с помощью расширенного алгоритма Евклида вместе с поднятием модуля полинома. Однако возможным улучшением может быть параллельный подбор полиномов в разных потоках, чтобы кратно уменьшить время подбора нужного.

3.3.2 RSA2048

Функция	Среднее время выполнения, мс	Количество итераций
Генерация ключа	718.21	100
Шифрование	3.67	1000
Дешифрование	1.77	1000
Вычисление padding	0.019	1000

Таблица 3: Результаты измерений для RSA2048

Опять же генерация ключа занимает значительно больше времени, чем операции шифрования и дешифрования, однако меньше, чем NTRU. В моей реализации RSA2048 для генерации ключей используется генерация случайных простых чисел с вероятностной проверкой на простоту по методу Рабина-Миллера. Этот метод быстрее,

чем классическая проверка по малой теореме Ферма и некоторой мере надежнее. К тому же так же возможна параллельная генерация простых чисел, что кратно сократит процесс генерации потенциально. Генерация padding для RSA занимает минимальное время.

3.3.3 RSA15360

Функция	Среднее время выполнения, мс	Количество итераций
Генерация ключа	9448	10
Генерация простого числа	60336	1
Шифрование	557.12	100
Дешифрование	401.01	1000

Таблица 4: Результаты измерений для RSA15360

Результаты измерений красноречиво говорят, почему RSA с модулем длиной 15360 бит не используются на практике, поскольку они бы очень замедляли работу систем. Некоторые шаги можно было бы не повторять каждый раз, например, генерацию модулей, однако время выполнения остальных операций все еще сильно превосходит версию с 2048-битным модулем.

3.3.4 ECC Curve25519

Функция	Среднее время выполнения, мс	Количество итераций
Генерация ключа	0.065	1000
Шифрование	0.068	1000
Дешифрование	0.067	1000

Таблица 5: Результаты измерений для ECC

Готовая реализация алгоритма шифрования на основе эллиптических кривых показала превосходные результаты производительности. Стоит отметить, что самостоятельная реализация довольно часто оказывается куда менее оптимизированной, чем готовая эталонная реализация, но сами по себе вычисления на эллиптических кривых не такие трудоемкие, как, например, в RSA, поэтому результаты вышли предсказуемо лучше. Но стоит отметить, что эллиптическая кривая в данном тесте обеспечивает уровень безопасности лишь в 128 бит по сравнению с NTRU и RSA.

3.3.5 NTRU-OAEP

Функция	Среднее время выполнения, мс	Количество итераций
Шифрование	2.47	1000
Дешифрование	2.511	1000

Таблица 6: Результаты измерений для NTRU-OAEP

Модофикация NTRU не привела к значительным падениям в производительности, что идет только в плюс. Однако и на примере RSA можно убедиться, что подобные механизмы как правило не сильно затратны и трудоемки.

3.4 Теоретическое сравнение алгоритмов

На практике мы убедились, что при сопоставимых уровнях защиты NTRU выигрывает у RSA, однако алгоритмы на основе ECC могут составить конкуренкцию в плане производительности и экономности. Теперь рассмотрим другие параметры этих алгоритмов.

3.4.1 Затраты памяти

Рассмотрим длину ключей алгоритмов при сопоставимых уровнях защииты.

Алгоритм	**Длина ключа**	**Уровень защиты**
RSA	3072 бит	128
NTRU	$\approx$ 4800+ бит	256
ECC	512+ бит	256

Таблица 7: Сравнение длин ключей

Легко заметить, что длина ключей у алгоритмов RSA и алгоритмов на эллиптических кривых значительно меньше, чем у NTRU, что говорит об относительной экономности данных алгоритмов в плане хранения и передачи ключей. Это может быть критично особенно при передаче, так как в начале как правило происходит обмен ключами для налаживания защищенного канала. В этом NTRU значительно проигрывает своим аналогам.

Сравним теперь получаемую длину шифротекстов после преобразования информации данными алгоритмами.

Алгоритм	**Длина шифротекста**
RSA	384 байт (сопоставим с размером модуля)
NTRU	$\approx$ 1200+ байт (сопоставим с размером полинома)
ECC	64 байта (сопоставим с размером ключа)

Таблица 8: Сравнение длин шифротекстов

В данном сравнении видно, что и размер шифротекста у NTRU выходит немаленький относительно остальных алгоритмов. Таким образом при шифровании с помощью NTRU придется отправлять куда больше данных, чем с остальными шифрами. Это справедливо даже если этот алгоритм будет использоваться лишь для налаживания защищенного канала, так как нужно произвести обмен ключами.

3.4.2 Безопасность

Безопасность данных алгоритмов можно рассматривать с разных сторон.

Одна сторона вопроса - это трудноразрешимые задачи. Каждый из трех алгоритмов основывается на одной из таких задач: RSA основывается на сложности факторизации произведения двух больших простых чисел, ECC основывается на сложности задачи дискретного логарифмирования на эллиптической кривой, а NTRU основывается

на сложности нахождения кратчайшего и ближайшего векторов. В перспективе задачи дискретного логарифмирования и факторизации можно будет решать примерно так же быстро, как и задачи возведения в степень по модулю и умножения целых чисел благодаря алгоритму Шора. Оценка трудоемкости факторизации целого числа N с помощью алгоритма Шора примерно $O((\log N)^2(\log\log N)(\log\log\log N))$. Однако этот алгоритм разработан для работы на квантовых компьютерах, и в ближайшее время перспектива взлома классических ассиметричных криптосистем квантовыми вычислениями достаточно мала. Но у NTRU есть преимущество в этом плане, поскольку на данный момент неизвестно достаточно быстрых алгоритмов нахождения ближайших векторов, которые могли бы взломать криптосистему NTRU даже на квантовом компьютере. Единственный известный алгоритм решения задачи - это алгоритм Ленстры-Ленстры-Ловаса. При данной n-мерной решетке с целыми координатами и верхней границей в виде базиса $|B|$ трудоемкость алгоритма равна $O(n^6 \log^3 |B|)$. С некоторыми модификациями можно достичь трудоемкости вычислений в $O(n^5 \log^2 |B|)$[7].

С другой стороны, есть другие уязвимости, которым могут быть подверженны данные алгоритмы. У NTRU, как было сказано в первом разделе, проблема заключается в семантической незащищенности и возможности проверки связи открытого текста и шифротекста, однако эти проблемы возникают при использовании NTRU в чистом виде без каких-либо модификаций. С реализованным OAEP-padding атаки по подобранному шифротексту становятся крайне маловероятными благодаря $H(M \,||\, R)$, а также обеспечивается семантическая защита за счет маскирования сообщения и конкатенации его со строкой случайных байт, которая тоже замаскирована [6]. Подобное улучшение есть для алгоритма RSA, которое так же обеспечивает защиту от нахождения связи между открытым текстом и шифротекстом, особенно если один и тот же ключ применяется на одних и тех же данных. Но в целом данные системы не могут работать и без прочих модификаций, обеспечивающих аутентификацию, в противном случае существует риск атаки человека посередине. Поэтому без подписей или аутентификационных кодов такие алгоритмы лучше не использовать.

Наконец стоит отметить, что алгоритм NTRU относительно молодой по сравнению с RSA и алгоритмами на основе эллиптических кривых, и исследований о нем меньше, чем о последних двух алгоритмах, так что его применение в реальных условиях пока что ограниченное, и оно как правило не оправдано тем фактом, что старые криптосистемы еще могут работать без угрозы их компрометации.

4 Возможные применения криптосистемы

Криптосистема NTRU была финалистом в отборе алгоритмов для стандартизации постквантовой криптографии в рамках инициативы NIST PQC. Что говорит о том, что она будет в дальнейшем изучаться и применяться на практике. Также данная система была принята в стандарт IEEE P1363 в рамках спецификации для криптографии на решетках с открытым ключом.

В общем случае криптографические системы с открытым ключом могут применяться для сиситем обмена и согласования секретных ключей или для цифровых подписей. У NTRU есть спецификации для обоих задач, однако версия для цифровых подписей NTRUSign показала себя как недостаточно безопасная, так как существует возможность подделки закрытых ключей, собирая данные подписей [5]. Система шифрования при этом показывает хороший уровень безопасности, а также достаточно низкое потребление ресурсов, что может быть удобно для применения в мобильных устройствах и смарт-картах.

Реальные применения NTRU включают пока что лишь реализации в рамках библиотек с открытым исходным кодом и некоторых коммерческих решений. Например, алгоритм NTRU был добавлен в OpenSSH в версию 9.0 вкупе с X25519 ECDH, а также была создана эталонная реализация в рамках публичных лицензий GNU (GPL).

5 Заключение

В ходе выполнения курсовой работы был изучен алгоритм NTRU, его устройство, его достоинства и недостатки, а также возможные модификации. Также была сделана программная реализация алгоритма вместе с модификацией, и был проведен сравнительный анализс с аналогичными криптосистемами с открытым ключом. В ходе анализа было выявлено, что криптосистема NTRU не может быть использована в чистом виде без маскирования входных данных из-за угрозы различных атак с использованием открытого или закрытого текста. Также была отмечена достаточно высокая производительность данного алгоритма, но при этом большие затраты по использованию памяти, что может накладывать определенные ограничения на использование данного алгоритма.

Список литературы

[1] Elaine Barker. “Recommendation for Key Management”. В: NIST Special Publication 800-57 Pa
(2020).

[2] J. Hoffstein и J. H. Silverman. “Optimizations for NTRU.” В: Public-key Cryptography and Com
(2000).

[3] Eliane Jaulmes и Antoine Joux. “A Chosen-Ciphertext Attack against NTRU”. В: Proc. of Crypto ’00 (2000).

[4] Joseph H. Silverman Jeffrey Hoffstein Jill Pipher. “NTRU: A new high-speed public key cryptosystem”. В: CRYPTO 1996 (1996).

[5] P. Nguyen и O. Regev. “Learning a Parallelepiped: Cryptanalysis of GGH and NTRU Signatures”. В: Advances in Cryptology – EUROCRYPT 2006 (2006), с. 271—288.

[6] Phong Q. Nguyen и David Pointcheval. “Analysis and Improvements of NTRU Encryption Paddings”. В: Advances in Cryptology — CRYPTO 2002 (2002).

[7] Nick Howgrave-Graham Nicolas Gama и Phong Q. Nguyen. “Symplectic Lattice Reduction and NTRU”. В: Advances in Cryptology – EUROCRYPT 2006 (2006).

[8] Prasanna Ravi и др. “Will You Cross the Threshold for Me? - Generic Side-Channel Assisted Chosen-Ciphertext Attacks on NTRU-based KEMs”. В: Cryptology ePrint Archive, Pap
(2021), с. 271—288.

[9] John M. Schanck. “NTRU”. В: NIST submission (2020).

Приложение А

Листинг кода NTRU

```
class Polynomial
{
public:
  int* pol;
  int deg;

  template <size_t len>
  Polynomial(const int (&arr)[len]);
  Polynomial(const int* = NULL, int = 0);
  Polynomial(int);
  Polynomial(const Polynomial& other);
  ~Polynomial();

  string pol_to_string(int = 0);

  friend ostream& operator<<(ostream& os, const Polynomial& pol);

  Polynomial& operator=(const Polynomial&);
  int& operator[](int);

  friend Polynomial operator+(const Polynomial&, const Polynomial&);
  friend Polynomial operator-(const Polynomial&, const Polynomial&);
  friend Polynomial operator*(const Polynomial&, const Polynomial&);
  friend Polynomial operator*(const Polynomial&, int);
  friend Polynomial operator*(int, const Polynomial&);
  friend Polynomial operator%(const Polynomial&, int);
};

template <size_t len>
Polynomial::Polynomial(const int (&arr)[len]) :deg(len)
{
  if (len == 0 || !arr)
  {
    pol = NULL;
    return;
  }

  pol = new int[deg];
  if (!pol)
  {
    deg = 0;
    pol = NULL;
    return;
  }

  for (int i = 0; i < deg; i++)
    pol[i] = arr[i];
}

Polynomial::Polynomial(const int* arr, int len) :deg(len)
{
  if (deg == 0)
  {
    pol = NULL;
    return;
  }
```

```
57
58    pol = new int[deg];
59    if (!pol)
60    {
61      pol = NULL;
62      deg = 0;
63      return;
64    }
65
66    for (int i = 0; i < deg; i++)
67      pol[i] = arr[i];
68  }
69
70  Polynomial::Polynomial(int len) :deg(len)
71  {
72    if (len == 0)
73    {
74      pol = NULL;
75      return;
76    }
77
78    pol = new int[deg];
79    if (!pol)
80    {
81      deg = 0;
82      pol = NULL;
83      return;
84    }
85
86    for (int i = 0; i < deg; i++)
87      pol[i] = 0;
88  }
89
90  Polynomial::Polynomial(const Polynomial& other)
91  {
92    deg = other.deg;
93    pol = new int[deg];
94    for (int i = 0; i < deg; i++)
95      pol[i] = other.pol[i];
96  }
97
98  Polynomial::~Polynomial()
99  {
100   if (pol)
101     delete[] pol;
102 }
103
104 string Polynomial::pol_to_string(int max_deg)
105 {
106   int out_deg;
107   if (max_deg <= 0)
108     out_deg = deg;
109   else if (max_deg < deg)
110     out_deg = max_deg;
111   else
112   {
113     cerr << "invalid max_deg value" << endl;
114     return "";
115   }
116
```

```
117   string st = "";
118   bool first = true;
119
120   for (int i = 0; i < out_deg; i++)
121   {
122     int c = pol[i];
123     if (c == 0) continue;
124
125     if (!first)
126     {
127       st = st + (c > 0 ? " + " : " - ");
128     }
129     else
130     {
131       if (c < 0)
132         st = st + "-";
133     }
134
135     int abs_c = abs(c);
136     if (abs_c != 1 || i == 0)
137       st = st + to_string(abs_c);
138     if (i >= 1)
139       st = st + "x";
140     if (i >= 2)
141       st = st + "^" + to_string(i);
142
143     first = false;
144   }
145
146   if (first) return "0";
147   return st;
148 }
149
150 ostream& operator<<(ostream& os, const Polynomial& poly)
151 {
152   string st = "";
153   bool first = true;
154
155   for (int i = 0; i < poly.deg; i++)
156   {
157     int c = poly.pol[i];
158     if (c == 0) continue;
159
160     if (!first)
161     {
162       st = st + (c > 0 ? " + " : " - ");
163     }
164     else
165     {
166       if (c < 0)
167         st = st + "-";
168     }
169
170     int abs_c = abs(c);
171     if (abs_c != 1 || i == 0)
172       st = st + to_string(abs_c);
173     if (i >= 1)
174       st = st + "x";
175     if (i >= 2)
176       st = st + "^" + to_string(i);
```

```
177
178     first = false;
179   }
180
181   if (first) os << "0";
182   else
183     os << st;
184
185   return os;
186 }
187
188 Polynomial& Polynomial::operator=(const Polynomial& oth)
189 {
190   if (&oth != this)
191   {
192     if (pol)
193       delete[] pol;
194
195     deg = oth.deg;
196     pol = new int[deg];
197
198     for (int i = 0; i < deg; i++)
199       pol[i] = oth.pol[i];
200   }
201   return *this;
202 }
203
204 int& Polynomial::operator[](int ind)
205 {
206   if (ind >= 0 && ind < deg)
207     return pol[ind];
208   else
209     throw out_of_range("Index is out of bounds");
210 }
211
212 Polynomial operator*(const Polynomial& pol1, const Polynomial& pol2)
213 {
214   if (pol1.deg != pol2.deg)
215     return pol1;
216   else
217   {
218     int deg = pol1.deg;
219     Polynomial res(deg);
220
221     for (int i = 0; i < deg; i++)
222       for (int j = 0; j < deg; j++)
223         res.pol[(i + j) % deg] += pol1.pol[i] * pol2.pol[j];
224
225     return res;
226   }
227 }
228
229 Polynomial operator*(const Polynomial& pol, int ml)
230 {
231   int deg = pol.deg;
232   Polynomial res(deg);
233
234   for (int i = 0; i < deg; i++)
235     res.pol[i] = pol.pol[i] * ml;
236
```

```
237   return res;
238 }
239
240 Polynomial operator*(int ml, const Polynomial& pol)
241 {
242   int deg = pol.deg;
243   Polynomial res(deg);
244
245   for (int i = 0; i < deg; i++)
246     res.pol[i] = pol.pol[i] * ml;
247
248   return res;
249 }
250
251 Polynomial operator+(const Polynomial& pol1, const Polynomial& pol2)
252 {
253   if (pol1.deg != pol2.deg)
254     return pol1;
255   else
256   {
257     int deg = pol1.deg;
258     Polynomial res(deg);
259
260     for (int i = 0; i < deg; i++)
261       res.pol[i] = pol1.pol[i] + pol2.pol[i];
262
263     return res;
264   }
265 }
266
267 Polynomial operator-(const Polynomial& pol1, const Polynomial& pol2)
268 {
269   if (pol1.deg != pol2.deg)
270     return pol1;
271   else
272   {
273     int deg = pol1.deg;
274     Polynomial res(deg);
275
276     for (int i = 0; i < deg; i++)
277       res.pol[i] = pol1.pol[i] - pol2.pol[i];
278
279     return res;
280   }
281 }
282
283 Polynomial operator%(const Polynomial& pol, int mod)
284 {
285   if (mod <= 0)
286     return pol;
287
288   int deg = pol.deg;
289   Polynomial res(deg);
290
291   for (int i = 0; i < deg; i++)
292     res.pol[i] = pol.pol[i] % mod;
293
294   return res;
295 }
```

Листинг А.1: Описание класса Polynomial и методов класса

```
int inverse_integer(int num, int mod)
{
  int r0 = mod;
  int r1;
  if (num < 0)
    r1 = num - mod * (num / mod) + mod;
  else
    r1 = num;

  int s0 = 0;
  int s1 = 1;

  int q;
  int tmp;

  while (r1 != 0)
  {
    q = r0 / r1;

    tmp = r1;
    r1 = r0 - r1 * q;
    r0 = tmp;

    tmp = s1;
    s1 = s0 - s1 * q;
    s0 = tmp;
  }

  if (r0 > 1) return 0;
  if (s0 < 0) s0 += mod;
  return s0;
}

Polynomial mod_div(const Polynomial& pol1, const Polynomial& pol2, int mod
    )
{
  int deg = pol1.deg;
  Polynomial res(deg);
  Polynomial div = pol1;

  int deg1 = deg - 1;
  int deg2 = deg - 1;

  while (deg2 >= 0 && pol2.pol[deg2] == 0) --deg2;
  while (deg1 >= deg2 && pol1.pol[deg1] == 0) --deg1;

  while (deg1 >= deg2)
  {
    int div_res = inverse_integer(pol2.pol[deg2], mod);
    if (!div_res)
      return Polynomial(0);
    int coef = div[deg1] * div_res % mod;
    int shift = deg1 - deg2;
    res[shift] = coef;

    Polynomial xn(deg);
    xn[shift] = 1;

    div = (div - xn * pol2 * coef) % mod;
    while (deg1 >= 0 && div[deg1] == 0) --deg1;
```

```
  }

  return res % mod;
}

bool check_zero(const Polynomial poly)
{
  for (int i = 0; i < poly.deg; i++)
    if (poly.pol[i] != 0)
      return true;

  return false;
}

bool check_one(const Polynomial poly)
{
  if (poly.pol[0] != 1)
    return false;

  for (int i = 1; i < poly.deg; i++)
    if (poly.pol[i] != 0)
      return false;

  return true;
}

int eucl_inverse_mod(Polynomial pol, int mod, Polynomial *inv)
{
  Polynomial q;
  Polynomial tmp;
  int deg = pol.deg;

  Polynomial r0(deg + 1);
  r0[0] = -1;
  r0[deg] = 1;

  Polynomial r1(deg + 1);
  for (int i = 0; i < deg + 1; i++)
  {
    if (i < deg)
      r1[i] = pol[i];
  }

  Polynomial t0(deg + 1);
  Polynomial t1(deg + 1);
  t1[0] = 1;

  while(check_zero(r1))
  {
    q = mod_div(r0, r1, mod);
    if (!q.deg)
      return 1;
    q = q % mod;

    // cout << "q: " << q << endl;

    tmp = r1;
    r1 = (r0 - r1 * q) % mod;
    r0 = tmp;

```

```
120     // cout << "r: " << r1 << endl;
121
122     tmp = t1;
123     t1 = (t0 - t1 * q) % mod;
124     t0 = tmp;
125   }
126
127   if (r0[0] < 0)
128     r0[0] = r0[0] + mod;
129
130   if (!check_one(r0)) return 1;
131
132   *inv = Polynomial(deg);
133
134   for (int i = 0; i < deg; i++)
135     inv->pol[i] = t0[i];
136
137   return 0;
138 }
139
140 int eucl_inverse_mod2k(Polynomial pol, int mod, Polynomial *inv)
141 {
142   int deg = pol.deg;
143   int md2k = mod;
144   while(md2k > 2)
145   {
146     if (md2k % 2 != 0)
147     {
148       cout << "Modulus must be the power of 2" << endl;
149       return 1;
150     }
151     md2k /= 2;
152   }
153
154   Polynomial res;
155   int ret = eucl_inverse_mod(pol, 2, &res);
156
157   if (ret)
158     return 1;
159
160   Polynomial int_2(deg);
161   int_2[0] = 2;
162
163   while (md2k < mod)
164   {
165     md2k *= 2;
166     res = res * (int_2 - pol * res) % md2k;
167   }
168
169   *inv = res;
170   return 0;
171 }
172
173 Polynomial generate_ternary_poly(int deg, int d)
174 {
175   Polynomial res(deg);
176
177   vector<int> inds(deg);
178   iota(inds.begin(), inds.end(), 0);
179
```

```
  for (int i = deg - 1; i > 0; --i)
  {
        uint32_t j;
        randombytes_buf(&j, sizeof(j));
        j = j % (1 + i);
        swap(inds[i], inds[j]);
    }

  for (int i = 0; i < d; i++)
    res[inds[i]] = 1;
  for (int i = 0; i < d; i++)
    res[inds[i + d]] = -1;

  return res;
}

vector<Polynomial> generate_ntru_key(int deg, int p, int q, int d)
{
  Polynomial f(deg);
  f[0] = 1;

  Polynomial gen = generate_ternary_poly(deg - 1, d);

  Polynomial g(deg);
  for (int i = 0; i < deg - 1; i++)
    g[i] = gen[i];

  Polynomial fq;
  Polynomial fp;

  while(true)
  {
    gen = generate_ternary_poly(deg - 1, d);
    for (int i = 0; i < deg - 1; i++)
      f[i] = f[i] + 3 * gen[i];

    int ret = eucl_inverse_mod2k(f % q, q, &fq);
    if (ret)
      continue;

    ret = eucl_inverse_mod(f % p, p, &fp);
    if (ret)
      continue;

    break;
  }
  vector<Polynomial> key = {f, g, fp, fq};
  return key;
}
```

Листинг А.2: Вспомогательные функции для вычислений и генерации ключей

```
vector<Polynomial> str_to_polyv(const string& msg, int deg)
{
  int poly_chs = deg / 6;
  int poly_len;
  if (msg.length() % poly_chs == 0)
    poly_len = msg.length() / poly_chs;
  else
    poly_len = msg.length() / poly_chs + 1;

```

```
  vector<Polynomial> poly_msg(poly_len);

  for (int i = 0; i < poly_len; i++)
  {
    Polynomial ch_poly(deg);

    for (int j = 0; j < poly_chs; j++)
    {
      unsigned char c;
      if (i * poly_chs + j < msg.length())
        c = msg[i * poly_chs + j];
      else if (i * poly_chs + j == msg.length())
        c = 1;
      else
        break;

      int k = 5;
      while (c > 0)
      {
        int coef = c % 3;
        coef = (coef == 2) ? -1 : coef;

        ch_poly[j * 6 + k] = coef;
        k--;
        c /= 3;
      }
    }

    poly_msg[i] = ch_poly;
  }
  return poly_msg;
}

Polynomial data_to_poly(const unsigned char* msg, int len, int deg)
{
  int poly_chs = deg / 6;

  Polynomial poly_data(deg);

  for (int j = 0; j < poly_chs; j++)
  {
    unsigned char c;
    if (j < len)
      c = msg[j];
    else if (j == len)
      c = 1;
    else
      break;

    int k = 5;
    while (c > 0)
    {
      int coef = c % 3;
      coef = (coef == 2) ? -1 : coef;

      poly_data[j * 6 + k] = coef;
      k--;
      c /= 3;
    }
  }
```

```
70
71    return poly_data;
72  }
73
74  Polynomial hash_to_poly(const unsigned char* msg, int len, int deg)
75  {
76    int poly_chs = deg / 6;
77
78    Polynomial poly_data(deg);
79
80    for (int j = 0; j < poly_chs; j++)
81    {
82      unsigned char c;
83      if (j < len)
84        c = msg[j];
85      else if (j == len)
86        c = 1;
87      else
88        break;
89
90      int k = 5;
91      while (c > 0)
92      {
93        int coef = c % 3;
94        coef = (coef == 2) ? -1 : coef;
95
96        poly_data[j * 24 + k * 4] = coef;
97        k--;
98        c /= 3;
99      }
100   }
101
102   return poly_data;
103 }
104
105 unsigned char* poly_to_data(Polynomial poly, int len, int deg)
106 {
107   unsigned char* data = new unsigned char[len];
108   int poly_chs = deg / 6;
109
110   for (int i = 0; i < poly_chs; i++)
111   {
112     unsigned char c = 0;
113
114     for (int j = 0; j < 6; j++)
115     {
116       c *= 3;
117       int coef = poly[i * 6 + j];
118       coef = (coef == -1) ? 2 : coef;
119
120       c += coef;
121     }
122
123     if (i < len)
124       data[i] = c;
125   }
126   return data;
127 }
128
129 string polyv_to_str(const vector<Polynomial> poly_msg, int deg)
```

```
{
  int poly_chs = deg / 6;
  int str_len = poly_msg.size() * poly_chs * 6;

  string data = "";

  for (int i = 0; i < poly_msg.size(); i++)
  {
    Polynomial ch_poly = poly_msg[i];

    for (int j = 0; j < poly_chs; j++)
    {
      unsigned char c = 0;

      for (int k = 0; k < 6; k++)
      {
        c *= 3;
        int coef = ch_poly[j * 6 + k];
        coef = (coef == -1) ? 2 : coef;

        c += coef;
      }

      if (c && c != 1)
        data = data + (char)c;
    }
  }

  return data;
}
```

Листинг А.3: Функции конвертации данных в полиномы и обратно

```
Polynomial random_blind(int deg)
{
  Polynomial res(deg);

  for (int i = 0; i < deg; i++)
  {
    uint8_t r = randombytes_uniform(3);
    res[i] = static_cast<int>(r) - 1;
  }

  return res;
}

vector<Polynomial> ntru_encrypt(vector<Polynomial> data, Polynomial
    pub_key, int q, int dr)
{
  int deg = pub_key.deg;
  Polynomial blind;
  vector<Polynomial> enc_data(data.size());

  for (int i = 0; i < data.size(); i++)
  {
    blind = generate_ternary_poly(deg, dr);
    enc_data[i] = (blind * pub_key + data[i]) % q;
  }

  return enc_data;
}
```

```
28
29 Polynomial minimize(Polynomial poly, int mod)
30 {
31   int deg = poly.deg;
32   Polynomial res(deg);
33
34   for (int i = 0; i < deg; i++)
35   {
36     if (poly[i] > (mod - mod % 2) / 2 - (mod + 1) % 2)
37       res[i] = poly[i] - mod;
38     else if (poly[i] < -(mod - mod % 2) / 2)
39       res[i] = poly[i] + mod;
40     else
41       res[i] = poly[i];
42   }
43   return res;
44 }
45
46 vector<Polynomial> ntru_decrypt(vector<Polynomial> enc_data, Polynomial
     priv_f, Polynomial priv_fp, int q, int p)
47 {
48   vector<Polynomial> data(enc_data.size());
49
50   for (int i = 0; i < enc_data.size(); i++)
51   {
52     Polynomial block = priv_f * enc_data[i] % q;
53     block = minimize(block, q);
54
55     block = block % p;
56     block = priv_fp * block % p;
57
58     block = minimize(block, p);
59     data[i] = block;
60   }
61   return data;
62 }
```

Листинг А.4: Функции шифрования и дешифрования

```
1  unsigned char* I2OSP (unsigned int x, int len)
2  {
3    unsigned char* res = new unsigned char[len];
4
5    for (int i = 0; i < len; i++)
6    {
7      res[3 - i] = x & 0xff;
8      x >>= 8;
9    }
10   return res;
11 }
12
13 // mask generating function
14 unsigned char* mgf1(unsigned char* seed, int seedLen, int maskLen)
15 {
16   unsigned char* mask = new unsigned char[maskLen];
17
18   unsigned char* seed_ctr = new unsigned char[seedLen + 4];
19   for (int i = 0; i < seedLen; i++)
20     seed_ctr[i] = seed[i];
21
22   int iter;
```

```
  if (maskLen % SHA256_BLOCK_SIZE == 0)
    iter = maskLen / SHA256_BLOCK_SIZE;
  else
    iter = maskLen / SHA256_BLOCK_SIZE + 1;

  for (unsigned int i = 0; i < iter; i++)
  {
    unsigned char* ctrStr = I2OSP(i, 4);

    BYTE hash[SHA256_BLOCK_SIZE];
    for (int j = 0; j < 4; j++)
      seed_ctr[seedLen + j] = ctrStr[j];
    sha256((const BYTE*)seed_ctr, seedLen + 4, hash);

    for (int j = 0; j < SHA256_BLOCK_SIZE; j++)
    {
      if (i * SHA256_BLOCK_SIZE + j < maskLen)
        mask[i * SHA256_BLOCK_SIZE + j] = hash[j];
      else
        break;
    }
    delete[] ctrStr;
  }
  delete[] seed_ctr;

  return mask;
}

unsigned char* padding_encode(string data, unsigned char* hash)
{
  unsigned char* enc_data = new unsigned char[129];

  unsigned char* rand = new unsigned char[32];
  randombytes_buf(rand, 32);

  unsigned char* mask = mgf1(rand, 32, 96);

  for (int i = 0; i < 96; i++)
  {
    if (i < data.size())
      enc_data[i] = data[i] ^ mask[i];
    else if (i == data.size())
      enc_data[i] = 0x01 ^ mask[i];
    else
      enc_data[i] = mask[i];
  }
  delete[] mask;

  mask = mgf1(enc_data, 96, 32);
  for (int i = 0; i < 32; i++)
    enc_data[96 + i] = rand[i] ^ mask[i];
  delete[] mask;

  enc_data[128] = '\0';

  BYTE m_r_string[128];
  for (int i = 0; i < 96; i++)
  {
    if (i < data.size())
      m_r_string[i] = data[i];
```

```
    else if (i == data.size())
      m_r_string[i] = 0x01;
    else
      m_r_string[i] = 0x00;
  }

  for (int i = 96; i < 128; i++)
    m_r_string[i] = rand[i % 32];

  delete[] rand;
  sha256(m_r_string, 128, hash);

  return enc_data;
}

string padding_decode(unsigned char* enc_data)
{
  unsigned char* mask = mgf1(enc_data, 96, 32);
  for (int i = 0; i < 32; i++)
    enc_data[96 + i] = enc_data[96 + i] ^ mask[i];
  delete[] mask;

  mask = mgf1(enc_data + 96, 32, 96);

  for (int i = 0; i < 96; i++)
    enc_data[i] = enc_data[i] ^ mask[i];
  delete[] mask;

  int len = 0;
  while(enc_data[len] && len < 96) len++;

  string data = "";
  for (int i = 0; i < len; i++)
    data += enc_data[i];
  data += '\0';

  return data;
}
```

Листинг А.5: Имплементация padding

```
int main()
{
  vector<Polynomial> key = generate_ntru_key(deg, p, q, d);

  Polynomial f = key[0], g = key[1], fp = key[2], fq = key[3];
  Polynomial h = p * fq * g % q;

  cout << "-- key polynomials --" << endl;

  cout << "f:  " << f.pol_to_string(12) << endl;
  cout << "fp: " << fp.pol_to_string(12) << endl;
  cout << "fq: " << fq.pol_to_string(12) << endl;
  cout << "g:  " << g.pol_to_string(12) << endl;
  cout << "h:  " << h.pol_to_string(12) << endl << endl;

  cout << "-- encryption testing --" << endl;

  string msg = "some test data to encrypt";
  cout << "data: " << msg << endl;

```

```
21
22    vector<Polynomial> poly_msg = str_to_polyv(msg, deg);
23    cout << "polynomial representation: " << poly_msg[0].pol_to_string(12)
       << endl;
24
25    vector<Polynomial> enc = ntru_encrypt(poly_msg, h, q, d);
26
27    cout << "encrypted: " << enc[0].pol_to_string(12) << endl;
28
29    enc = ntru_decrypt(enc, f, fp, q, p);
30
31    msg = polyv_to_str(enc, deg);
32    cout << "decrypted: " << enc[0].pol_to_string(12) << endl;
33
34    BYTE hash[SHA256_BLOCK_SIZE];
35
36    unsigned char* encoded = padding_encode(msg, hash);
37
38    cout << "encoded: ";
39      for (int i = 0; i < 128; i++)
40      cout << hex << setw(2) << setfill('0') << (int)encoded[i];
41    cout << endl;
42
43    Polynomial encoded_poly = data_to_poly(encoded, 128, deg);
44    Polynomial blind_poly = hash_to_poly(hash, 32, deg);
45    delete[] encoded;
46
47    cout << "s || t:  " << encoded_poly.pol_to_string(12) << endl;
48    cout << "H(M||R): " << blind_poly.pol_to_string(48) << endl;
49
50    Polynomial encrypted = (encoded_poly + blind_poly * h) % q;
51    for (int i = 0; i < deg; i++)
52      encrypted[i] += q;
53    encrypted = encrypted % q;
54    cout << "Encrypted: " << encrypted.pol_to_string(12) << endl;
55
56    Polynomial decrypted = (encrypted * f) % q;
57    decrypted = minimize(decrypted, q);
58    decrypted = decrypted % p;
59    decrypted = decrypted * fp % p;
60    decrypted = minimize(decrypted, p);
61    cout << "Decrypted: " << decrypted.pol_to_string(12) << endl;
62
63    encoded = poly_to_data(decrypted, 128, deg);
64
65    string data = padding_decode(encoded);
66    cout << "decoded: " << data << endl;
67    delete[] encoded;
68 }
```

Листинг А.6: Код для тестирования NTRU