

## 1. Описание алгоритма.

Криптосистема NewHope представляет собой набор механизмов инкапсуляции ключей (КЕМ), обозначенных как NewHopeCRA-КЕМ и NewHope-ССА-КЕМ, которые основаны на предполагаемой квантовой стойкости проблемы RLWE. Обе схемы основаны на варианте ранее предложенной схемы NewHope-Simple, смоделированной как схема семантически безопасного шифрования с открытым ключом (РКЕ) в отношении адаптивных атак с выбранным открытым текстом (CRA), которую мы называем NewHope-CRA-РКЕ. Однако в этом представлении NewHope-CRAPKE используется только внутри NewHope-CRA-КЕМ и NewHope-ССА-КЕМ и не предназначен для использования в качестве независимой схемы РКЕ с защитой CRA, потому что он не принимает сообщения произвольной длины.

Для предлагаемого нами NewHope-CRA-КЕМ мы обеспечиваем преобразование NewHope-CRA-РКЕ в пассивно безопасный КЕМ. Для NewHope-ССА-КЕМ мы показываем, как реализовать семантически безопасную инкапсуляцию ключей в отношении атак с адаптивным выбранным зашифрованным текстом (ССА) на основе NewHope-CRA-РКЕ. Заметим, что некоторые алгоритмы, особенно связанные с кодированием и декодированием, оптимизированы для размерностей  $n = 512$  или  $n = 1024$ , которые мы поддерживаем в NewHope. При этом алгоритмы рассчитаны на  $q = 12289$  и  $k = 8$  (параметр распределения шума).

## 2. Параметры.

Для нашей криптосистемы NewHope мы указываем два набора параметров NewHope512 и NewHope1024 в таблице 1. Эти наборы параметров используются для реализации схемы NewHope-CPA-КЕМ или NewHope-CCA-КЕМ. В случае, если уровень безопасности должен быть указан вместе со схемой, мы используем примерное обозначение NewHope1024-CPA-КЕМ для ссылки на схему NewHope-CPA-КЕМ, созданную с набором параметров NewHope1024. В таблице 2 мы приводим размеры открытого ключа, секретного ключа и зашифрованного текста для наших двух КЕМ, которые поддерживают передачу 256-битного сообщения или ключа.

Набор параметров	NewHope512	NewHope1024
Величина $n$	512	1024
Модуль $q$	122289	12289
Параметр шума $k$	8	8
NTT параметр $\gamma$	10968	7
Вероятность ошибки дешифрования	$2^{-213}$	$2^{-216}$
Заявленная постквантовая битовая безопасность	101	233
Категория уровня безопасности NIST	1	5

Таблица 1. Параметры для NewHope512 и для NewHope1024.

Параметры в таблице 1 полностью определяют NewHope, а все остальные промежуточные параметры могут быть рассчитаны оттуда. Для удобства перечислим промежуточные параметры:

- NewHope512:

$$\gamma = \sqrt{\omega} = 10968; \omega = 3; \omega^{-1} \bmod(q) = 8193; \gamma^{-1} \bmod(q) = 3656; n^{-1} \bmod(q) = 12265$$

- NewHope1024:

$$\gamma = \sqrt{\omega} = 7; \omega = 49; \omega^{-1} \bmod(q) = 1254; \gamma^{-1} \bmod(q) = 8778; n^{-1} \bmod(q) = 12277$$

Набор параметров	Открытый ключ ( $ pk $ )	Закрытый ключ ( $ sk $ )	Зашифрованный текст ( $ ciphertext $ )
NewHope512-CPA-KEM	928	869	1088
NewHope1024-CPA-KEM	1824	1792	2176
NewHope512-CCA-KEM	928	1888	1120
NewHope1024-CCA-KEM	1824	3680	2208

Таблица 2. Размеры открытых ключей, секретных ключей и зашифрованных текстов наших экземпляров NewHope в байтах.

Параметры NewHope не могут быть выбраны свободно. Размерность  $n$  должна быть целой степенью двойки, чтобы поддерживать эффективные алгоритмы NTT и сохранять свойства безопасности RLWE. Степени, не являющиеся степенью двойки, также возможны, но сопряжены с некоторыми осложнениями, в частности, определяющий многочлен кольца больше не может иметь вид  $X^n + 1$ .

Кроме того,  $n$  должно быть больше или равно 256 из-за нашего выбора функции кодирования, которая должна внедрять 256-битное сообщение в  $n$ -мерный полином в NewHope-CPA-PKE. Модуль  $q$  должен быть выбран как целое простое число  $q$ , при этом  $q \equiv 1 \pmod{2n}$ , для поддержки эффективных алгоритмов NTT. Целочисленный параметр  $k$  распределения шума должен быть выбран таким, чтобы вероятность ошибок дешифрования была незначительной. На высоком уровне окончательная безопасность NewHope зависит от  $(q, n, k)$ , где большее  $n$  и большее  $\frac{k}{q}$  приводят к более высокому уровню безопасности. Выбор  $\gamma$  не влияет на безопасность, но необходим для корректности и представляет собой просто наименьшее возможное значение.

В маловероятном случае, когда требуется более высокий уровень безопасности при сохранении уверенности в допущении RLWE, необходимо просто выбрать набор параметров NewHopeLudicrous с размерностью  $n=2048$  и  $k=8$ . Это фактически удвоит время выполнения и размер открытых ключей, зашифрованных текстов, секретных ключей (возможно). Также возможно небольшое повышение безопасности для NewHope-CPA-KEM. Поскольку на практике эту схему следует использовать только в эфемерных условиях, где ошибки дешифрования

менее критичны, можно немного увеличить  $k$  (например,  $k=16$ , как в NewHope-Usenix).

Мы не считаем, что больший модуль  $q$  приведет к повышению производительности или лучшему компромиссу между производительностью и безопасностью. Однако в случае увеличения  $q$  необходимо адаптировать и параметр  $k$ . Выбор  $n$  не как степени двойки сделал бы схему небезопасной. В общем, схемы на основе RLWE не требуют простого модуля  $q$  для обеспечения безопасности или производительности. Однако, поскольку NewHope напрямую использует свойства негациклического NTT, параметры необходимо выбирать так, чтобы  $q$  было простым числом и чтобы выполнялось  $q \equiv 1 \pmod{2n}$ . Схема без ограничений относительно модуля  $q$  выглядела бы совсем иначе с точки зрения разработчика, чем NewHope.

### 3. Описание работы алгоритма.

- *Схема шифрования с открытым ключом IND-CPA-secure.*

Для нашего промежуточного строительного блока пассивно защищенной схемы PKE NewHope-CPA-PKE с фиксированным пространством сообщений 256 бит мы определяем генерацию ключа в алгоритме 1 (NewHope-CPA-PKE.Gen), шифрование в алгоритме 2 (NewHope-CPA-PKE.Encrypt) и расшифровку в алгоритме 3 (NewHope-CPA-PKE.Decrypt). В этом разделе описаны все подфункции, используемые в NewHope-CPA-PKE. Мы предполагаем в каждой функции неявный доступ к глобальным параметрам  $n$ ,  $q$ ,  $\gamma$ , которые определяются выбранным набором параметров.

#### *Выборка, случайность, байтовые массивы и SHAKE.*

Помимо полиномов в  $R_q$  и векторов основной другой структурой данных, которую мы используем, являются массивы байтов. Например, вся случайность выбирается в виде массивов байтов. При генерации ключей  $seed \xleftarrow{\$} \{0, \dots, 255\}^{32}$  обозначает выборку массива байтов с 32 однородными целочисленными элементами в диапазоне от 0 до 255 из генератора случайных чисел. Этот генератор случайных чисел должен быть непредсказуемым и, следовательно, должен использовать физический источник энтропии или другие средства.

В качестве сильной хеш-функции мы используем SHAKE256. Функция SHAKE256( $l$ ,  $d$ ) принимает в качестве входных данных целое

число  $l$ , задающее количество выходных байтов, и массив байтов входных данных  $d$ . Количество данных, которые необходимо усвоить, равно длине  $d$ . Например, при генерации ключа мы используем SHAKE256 для вычисления  $v \leftarrow \text{SHAKE256}(64, \text{seed})$ , где мы хэшируем 32-байтовое случайное начальное число, обозначенное как начальное, и выводим массив байтов  $v$  с 64 элементами в диапазоне  $\{0, \dots, 255\}$ .

---

**Algorithm 1** NEWHOPE-CPA-PKE Key Generation

---

```

1: function NEWHOPE-CPA-PKE.GEN()
2:    $\text{seed} \xleftarrow{\$} \{0, \dots, 255\}^{32}$ 
3:    $z \leftarrow \text{SHAKE256}(64, \text{seed})$ 
4:    $\text{publicseed} \leftarrow z[0:31]$ 
5:    $\text{noiseseed} \leftarrow z[32:63]$ 
6:    $\hat{a} \leftarrow \text{GenA}(\text{publicseed})$ 
7:    $s \leftarrow \text{PolyBitRev}(\text{Sample}(\text{noiseseed}, 0))$ 
8:    $\hat{s} \leftarrow \text{NTT}(s)$ 
9:    $e \leftarrow \text{PolyBitRev}(\text{Sample}(\text{noiseseed}, 1))$ 
10:   $\hat{e} \leftarrow \text{NTT}(e)$ 
11:   $\hat{b} \leftarrow \hat{a} \circ \hat{s} + \hat{e}$ 
12:  return  $(pk = \text{EncodePK}(\hat{b}, \text{publicseed}), sk = \text{EncodePolynomial}(\hat{s}))$ 

```

---



---

**Algorithm 2** NEWHOPE-CPA-PKE Encryption

---

```

1: function NEWHOPE-CPA-PKE.ENCRYPT( $pk \in \{0, \dots, 255\}^{7 \cdot n/4 + 32}$ ,  $\mu \in \{0, \dots, 255\}^{32}$ ,  $\text{coin} \in \{0, \dots, 255\}^{32}$ )
2:   $(\hat{b}, \text{publicseed}) \leftarrow \text{DecodePk}(pk)$ 
3:   $\hat{a} \leftarrow \text{GenA}(\text{publicseed})$ 
4:   $s' \leftarrow \text{PolyBitRev}(\text{Sample}(\text{coin}, 0))$ 
5:   $e' \leftarrow \text{PolyBitRev}(\text{Sample}(\text{coin}, 1))$ 
6:   $e'' \leftarrow \text{Sample}(\text{coin}, 2)$ 
7:   $\hat{t} \leftarrow \text{NTT}(s')$ 
8:   $\hat{u} \leftarrow \hat{a} \circ \hat{t} + \text{NTT}(e')$ 
9:   $v \leftarrow \text{Encode}(\mu)$ 
10:  $v' \leftarrow \text{NTT}^{-1}(\hat{b} \circ \hat{t}) + e'' + v$ 
11:  $h \leftarrow \text{Compress}(v')$ 
12: return  $c = \text{EncodeC}(\hat{u}, h)$ 

```

---



---

**Algorithm 3** NEWHOPE-CPA-PKE Decryption

---

```

1: function NEWHOPE-CPA-PKE.DECRYPT( $c \in \{0, \dots, 255\}^{7 \cdot \frac{n}{4} + 3 \cdot \frac{n}{8}}$ ,  $sk \in \{0, \dots, 255\}^{7 \cdot n/4}$ )
2:   $(\hat{u}, h) \leftarrow \text{DecodeC}(c)$ 
3:   $\hat{s} \leftarrow \text{DecodePolynomial}(sk)$ 
4:   $v' \leftarrow \text{Decompress}(h)$ 
5:   $\mu \leftarrow \text{Decode}(v' - \text{NTT}^{-1}(\hat{u} \circ \hat{s}))$ 
6:  return  $\mu$ 

```

---

Для доступа к байтовым массивам мы используем обозначение скобок, где  $v[i]$  для положительного целого числа  $i$  обозначает  $i$ -й байт в массиве  $v$ . Для доступа к диапазонам байтов мы используем обозначение  $x \leftarrow v[i:j]$  для положительных целых чисел  $i \leq j$ , где  $x$

соответствует байту с  $i$  по  $j$  из  $v$ . По  $r \leftarrow \{0, \dots, 255\}^x$  мы объявляем, что  $r$  является байтовым массивом длины  $x$ . Используя аналогичное обозначение, через  $g$  мы объявляем, что переменная  $g$  является полиномом в  $R_q$ , где все коэффициенты равны нулю. Для битовых операций мы используем операторы  $\oplus, \ominus, |$  и  $\&$ , как и в контексте языка программирования C. Таким образом,  $x \ll i$  для положительных целых чисел  $i$ ,  $x$  обозначает сдвиг вправо на  $i$ . Тот же оператор можно применить к байту массива байтов, так что  $y[j] \ll i$  для положительного целого числа  $i$ ,  $j$  представляет сдвиг вправо на  $i$   $j$ -го байта массива байтов  $y$ . Сдвиг влево обозначается как  $x \gg i$  для положительных целых чисел  $i$ ,  $y$  и предполагается неявная модульная редукция по модулю  $2^{32}$ . Когда оператор сдвига влево применяется к  $j$ -му байту массива байтов  $y$  как  $y[j] \gg i$  для положительного целого числа  $i$ ,  $j$  предполагается неявное сокращение по модулю  $2^8$ . Оператор  $a | b$  обозначает побитовое «или», а оператор  $a \& b$  обозначает побитовое «и» двух положительных целых чисел  $a, b$  или двух байтов в байтовом массиве. Чтобы преобразовать байт  $a[i]$  в массиве байтов  $a$  в положительное целое число  $z$ , мы используем  $z = b2^i(a[i])$ . Для обозначения положительных целых чисел в шестнадцатеричном представлении мы используем префикс  $0x$ , так что  $0x01010101 = 16843009$ . Чтобы вычислить вес Хэмминга, сумму всех битов, установленных в единицу в двоичной записи, байта или целого числа  $b$ , мы пишем  $HW(b)$ .

Обратите внимание, что NewHope-CPA-PKE.Encrypt не имеет прямого доступа к генератору случайных чисел, поскольку все псевдослучайные данные получаются путем расширения 32-х байтового начального числа, предоставленного пользователем,  $coin \in \{0, \dots, 255\}^{32}$ , которое должно быть получено от генератора истинных случайных значений. Это необходимо для прямого использования NewHope-CPA-PKE.Encrypt в стандартных преобразованиях ССА. Расшифровка является детерминированной и не требует случайных значений. Для распределения секрета и ошибки RLWE мы используем центрированное биномиальное распределение  $\psi_k$  параметра  $k=8$ . В общем случае можно произвести выборку из  $\psi_k$  для целого числа  $k > 0$ , вычислив  $\sum_{i=0}^{k-1} b_i - b'_i$ , где  $b_i, b'_i \in \{0, 1\}$  являются равномерными независимыми битами. Распределение  $\psi_k$  центрировано (среднее значение равно 0), имеет дисперсию  $k/2$ , и мы устанавливаем  $k = 8$  во всех экземплярах. Это дает стандартное отклонение

$\zeta = \sqrt{8/2}$ . Мы описываем выборку из  $\psi_8$  в алгоритме 4 как функцию **Sample**, которая принимает в качестве входных данных 32-байтовое начальное число и целочисленный параметр  $0 \leq \text{nonce} < 2^8$  для разделения доменов. Таким образом, одно семя можно использовать для выборки нескольких полиномов. На выходе получается многочлен  $r \in R_q$ , где все  $n$  коэффициентов независимо распределены в соответствии с  $\psi_8$ .

---

**Algorithm 4** Deterministic sampling of polynomials in  $R_q$  from  $\psi_8^n$

---

```

1: function SAMPLE( $\text{seed} \in \{0, \dots, 255\}^{32}$ , positive integer  $\text{nonce}$ )
2:    $\mathbf{r} \leftarrow R_q$ 
3:    $\text{extseed} \leftarrow \{0, \dots, 255\}^{34}$ 
4:    $\text{extseed}[0:31] \leftarrow \text{seed}[0:31]$ 
5:    $\text{extseed}[32] \leftarrow \text{nonce}$ 
6:   for  $i$  from 0 to  $(n/64) - 1$  do
7:      $\text{extseed}[33] \leftarrow i$ 
8:      $\text{buf} \leftarrow \text{SHAKE256}(128, \text{extseed})$ 
9:     for  $j$  from 0 to 63 do
10:       $a \leftarrow \text{buf}[2 * j]$ 
11:       $b \leftarrow \text{buf}[2 * j + 1]$ 
12:       $r_{64*i+j} = \text{HW}(a) + q - \text{HW}(b) \bmod q$ 
13:   return  $\mathbf{r} \in R_q$ 

```

---

### Многочлены и НТТ.

Основными математическими объектами, которыми оперируют в NewHore, являются многочлены от  $R_q = \mathbf{Z}_q[X]/(X^n + 1)$ , такие как  $s, e, \hat{s}, \hat{a}, \hat{b}, s', e', e'', \hat{t}, \hat{u}, \hat{e}, v, v'$ . Для

многочлена  $c \in R_q$ , где  $c = \sum_{i=0}^{n-1} c_i X^i$ , обозначим через  $c_i$   $i$ -й

коэффициент при  $c$  при целом  $i \in \{0, \dots, n-1\}$ . Мы используем те же обозначения для доступа к элементам векторов, которые не нужны в  $R_q$ . Сложение или вычитание

полиномов из  $R_q$  (обозначаемых как  $+$  или  $-$  соответственно) представляет собой обычное сложение или вычитание по коэффициентам, такое что для

$a = \sum_{i=0}^{n-1} a_i X^i \in R_q$  и  $b = \sum_{i=0}^{n-1} b_i X^i \in R_q$  получаем  $a + b = \sum_{i=0}^{n-1} (a_i + b_i \bmod(q)) X^i$  и

$a - b = \sum_{i=0}^{n-1} (a_i - b_i \bmod(q)) X^i$ . В общем, для полиномиального

умножения существуют быстрые квазилогарифмические алгоритмы. Мы явно указываем, как использовать теоретико-числовое преобразование (NTT); некоторые полиномы также передаются в преобразованном представлении. Однако разработчик может выбрать другой алгоритм полиномиального умножения, такой как умножение Карацубы или школьного учебника, а затем преобразовать результат в домен NTT, чтобы он соответствовал этой спецификации.

С помощью NTT полиномиальное умножение элементов из  $R_q = \mathbb{Z}_q[X]/(X^n + 1)$  может быть выполнено путем вычисления  $c = NTT^{-1}(NTT(a) \circ NTT(b))$  для  $a, b, c \in R_q$ . Оператор  $\circ$  обозначает покоэффициентное умножение двух многочленов  $a, b \in R_q$  таких, что  $a \circ b = \sum_{i=0}^{n-1} (a_i b_i \bmod(q)) X^i$ . NTT определенный в  $R_q$ , может быть реализован очень эффективно, если  $n$  — степень двойки, а  $q$  — простое число, для которого выполняется  $q \equiv 1 \bmod(2n)$ . Таким образом, существуют примитивный  $n$ -й степени из единицы  $\omega$  и его квадратный корень  $\gamma = \sqrt{\omega} \bmod(q)$ . Путем умножения по коэффициентам на степени  $\gamma$  перед вычислением NTT и после обратного преобразования на  $\gamma^{-1} \bmod(q)$  заполнение нулями не требуется, и NTT с  $n$  точками можно использовать для преобразования многочлена с  $n$  коэффициентами.

Для полинома  $g = \sum_{i=0}^{n-1} g_i X^i \in R_q$  определим

$$NTT(g) = \hat{g} = \sum_{i=0}^{n-1} \hat{g}_i X^i, \text{ с}$$

$$\hat{g}_i = \sum_{j=0}^{n-1} \gamma^j g_j \omega^{ij} \bmod(q),$$

где  $\omega$  —  $n$ -й первообразный корень из единицы, а  $\gamma = \sqrt{\omega} \bmod(q)$ .



Обратите внимание, что в большинстве реализаций будет использоваться встроенный алгоритм NTT, который обычно требует операций реверсирования битов, которые не включены в приведенное ранее простое описание NTT. В качестве оптимизации мы позволяем реализациям пропускать эти обращения битов для прямого преобразования, поскольку все входные данные представляют собой только случайный шум. Таким образом, и это немного нелогично, мы определяем бит-реверсирование и выполняем его на полиномах, которые входят в NTT. С реверсированием битов и нашим простым определением NTT разработчикам не нужно применять реверсирование при использовании NTT на месте. Обратите внимание, что при генерации ключа эта оптимизация прозрачна для схемы, но из-за повторного шифрования разработчики должны следовать нашим инструкциям. Для положительного целого числа  $v$  и степени двойки  $n$  мы формально определяем обращение битов как  $BitRev(v) = \sum_{i=0}^{\log_2(n)-1} (((v \gg i) \& 1) \ll (\log_2(n) - 1 - i))$ .

Для многочленов  $s, z \in R_q$  обращение битов многочлена  $s$  равно

$$z = PolyBitRev(s) = \sum_{i=0}^{n-1} s_i X^{BitRev(i)}$$

Функция  $NTT^{-1}$  является обратной функции NTT. Вычисление  $NTT^{-1}$  по существу такое же, как вычисление NTT, за исключением того, что оно использует  $\omega^{-1} \bmod(q)$ , умножает на степени  $\gamma^{-1} \bmod(q)$  после суммирования, а также умножает каждый коэффициент на скаляр  $n^{-1} \bmod(q)$  так что

$$NTT^{-1}(\hat{g}) = g = \sum_{i=0}^{n-1} g_i X^i, \text{ с}$$

$$g_i = (n^{-1} \gamma^{-1} \sum_{j=0}^{n-1} \hat{g}_j \omega^{-ij}) \bmod(q)$$

Обратите внимание, что мы определяем операцию  $x \bmod(q)$  для целых чисел  $x$  и  $q$  так, чтобы результат всегда находился в диапазоне  $[0; q-1]$ . Если не указано иное, когда мы обращаемся к элементу  $a_i$  полинома  $a \in R_q$ , мы всегда предполагаем, что  $a_i$  приведено по модулю  $q$  и в диапазоне  $[0; q-1]$ .

### **Определение GenA.**

Общедоступный параметр  $a$  генерируется GenA, который принимает в качестве входных данных 32-байтовое начальное значение массива. Функция описана в Алгоритме 5. Результирующий многочлен  $a$  (обозначаемый как  $\hat{a}$ ) считается находящимся в области NTT. Это возможно потому, что NTT преобразует однородные многочлены в однородные многочлены. Внутри GenA мы используем хэш-функцию SHAKE128 для расширения псевдослучайного начального числа и определяем функцию, которая преобразует массив байтов во внутреннее состояние SHAKE128, а затем мы используем другую функцию для получения псевдослучайных данных путем сжатия внутреннего состояния. Функции  $state \leftarrow \text{SHAKE128Absorb}(d)$  принимают в качестве входных данных массив байтов  $d$ . Он выводит массив байтов длиной 200, который представляет состояние после поглощения  $d$ . Для получения псевдослучайных значений  $\text{buf}$  используется  $state \leftarrow \text{SHAKE128Squeeze}(j, state)$ . В качестве входных данных функция принимает положительное целое число  $j$ , определяющее количество выходных блоков SHAKE128, которые должны быть созданы, и состояние в 200 байт. Он выводит  $\text{buf}$  байтового массива длиной  $168j$  и состояние байтового массива длиной 200.

### **Кодирование и декодирование секретного и открытого ключей.**

Обратите внимание, что многочлены передаются в домене NTT, и, следовательно, для обеспечения совместимости необходимо использовать наше определение и параметризацию NTT. Чтобы закодировать многочлен принадлежащий  $R_q$  в массив байтов, мы используем EncodePolynomial, как описано в Алгоритме 6. Функция DecodePolynomial, как описано в Алгоритме 7, преобразует массив байтов в элемент, принадлежащий  $R_q$ . Секретный ключ состоит только из одного многочлена  $s \in R_q$ , и, таким образом, мы можем напрямую применить EncodePolynomial( $\hat{s}$ ). Затем секретный ключ кодируется либо в массив из 869 байт ( $n = 512$ ), либо в 1792 байта ( $n = 1024$ ). Открытый ключ кодируется в виде массива из 928 байт ( $n = 512$ ) или 1824 байт ( $n = 1024$ ) с помощью EncodePK( $\hat{b}, \text{seed}$ ), описанного в Алгоритме 8. Он принимает в качестве входных данных многочлен  $\hat{B} \in R_q$  и начальный массив байтов с 32 элементами. Функция DecodePk(pk) декодирует открытый ключ и предусмотрена в Алгоритме 9.

### **Кодирование и декодирование зашифрованного текста.**

Кодирование зашифрованного текста описано в Алгоритме 13. Зашифрованный текст с кодируется как массив из 1088 байт ( $n = 512$ ) или 2176 байт ( $n = 1024$ ) с помощью  $\text{EncodeC}(\hat{u}, h)$ , который принимает в качестве входных данных многочлен в  $\hat{u} \in R_q$  и массив  $h$  из  $3n/8$  байт, который был сгенерирован  $\text{Compress}(v')$ , как указано в Алгоритме 12. Функции сжатия и декомпрессии просто выполняют переключение модуля по коэффициенту между модулем  $q$  и модулем 8 путем умножения на новый модуль, а затем выполнения округления по старому модулю. Для декодирования зашифрованного текста используется функция  $\text{DecodeC}$ , которая выводит  $\hat{u} \in R_q$  и массив байтов  $h$ , который затем передается для распаковки, чтобы получить  $v' \in R_q$ .

В NewHope-CPA-PKE 256-битное сообщение  $\mu$ , представленное в виде массива из 32 байт, должно быть закодировано в элемент в  $R_q$  во время шифрования и декодировано из элемента в  $R_q$  в массив байтов во время дешифрования. Для обеспечения устойчивости к ошибкам каждый бит 256-битного сообщения  $\mu \in \{0, \dots, 255\}^{32}$  кодируется в коэффициенты  $\lfloor n/256 \rfloor$  с помощью  $\text{Encode}$  (см. Алгоритм 10). Функция декодирования  $\text{Decode}$  (см. Алгоритм 11) отображает коэффициенты  $\lfloor n/256 \rfloor$  обратно в исходный бит ключа. Например, для  $n = 1024$  возьмите коэффициенты  $4 = \lfloor 1024/256 \rfloor$  каждый в диапазоне  $\{0, \dots, q-1\}$ , вычитите  $\lfloor q/2 \rfloor$  из каждого из них, накопите их абсолютные значения и установите бит ключа равным 0, если сумма больше  $q$  или к 1 в противном случае.

---

**Algorithm 5** Deterministic generation of  $\hat{a}$  by expansion of a seed

---

```

1: function GENA( $seed \in \{0, \dots, 255\}^{32}$ )
2:    $\hat{a} \leftarrow \mathcal{R}_q$ 
3:    $extseed \leftarrow \{0, \dots, 255\}^{33}$ 
4:    $extseed[0 : 31] \leftarrow seed[0:31]$ 
5:   for  $i$  from 0 to  $(n/64) - 1$  do
6:      $ctr \leftarrow 0$ 
7:      $extseed[32] \leftarrow i$ 
8:      $state \leftarrow \text{SHAKE128Absorb}(extseed)$ 
9:     while  $ctr < 64$  do
10:       $buf, state \leftarrow \text{SHAKE128Squeeze}(1, state)$ 
11:       $j \leftarrow 0$ 
12:      for  $j < 168$  and  $ctr < 64$  do
13:         $val \leftarrow \text{b2i}(buf[j]) \mid (\text{b2i}(buf[j+1]) \ll 8)$ 
14:        if  $val < 5 \cdot q$  then
15:           $\hat{a}_{i*64+ctr} \leftarrow val$ 
16:           $ctr \leftarrow ctr + 1$ 
17:           $j \leftarrow j + 2$ 
18:   return  $\hat{a} \in \mathcal{R}_q$ 

```

---

---

**Algorithm 6** Encoding of a polynomial in  $\mathcal{R}_q$  to a byte array

---

```
1: function ENCODEPOLYNOMIAL( $\hat{s}$ )
2:    $r \leftarrow \{0, \dots, 255\}^{7 \cdot n/4}$ 
3:   for  $i$  from 0 to  $n/4 - 1$  do
4:      $t0 \leftarrow \hat{s}_{4*i+0} \bmod q$ 
5:      $t1 \leftarrow \hat{s}_{4*i+1} \bmod q$ 
6:      $t2 \leftarrow \hat{s}_{4*i+2} \bmod q$ 
7:      $t3 \leftarrow \hat{s}_{4*i+3} \bmod q$ 
8:      $r[7*i+0] \leftarrow t0 \& 0xff$ 
9:      $r[7*i+1] \leftarrow (t0 \gg 8) | (t1 \ll 6) \& 0xff$ 
10:     $r[7*i+2] \leftarrow (t1 \gg 2) \& 0xff$ 
11:     $r[7*i+3] \leftarrow (t1 \gg 10) | (t2 \ll 4) \& 0xff$ 
12:     $r[7*i+4] \leftarrow (t2 \gg 4) \& 0xff$ 
13:     $r[7*i+5] \leftarrow (t2 \gg 12) | (t3 \ll 2) \& 0xff$ 
14:     $r[7*i+6] \leftarrow (t3 \gg 6) \& 0xff$ 
15:   return  $r \in \{0, \dots, 255\}^{7 \cdot n/4}$ 
```

---

---

**Algorithm 7** Decoding of a polynomial represented as a byte array into an element in  $\mathcal{R}_q$ 

---

```
1: function DECODEPOLYNOMIAL( $v \in \{0, \dots, 255\}^{7 \cdot n/4}$ )
2:   for  $i$  from 0 to  $n/4 - 1$  do
3:      $r \leftarrow \mathcal{R}_q$ 
4:      $r_{4*i+0} \leftarrow \text{b2i}(v[7*i+0]) | ((\text{b2i}(v[7*i+1]) \& 0x3f) \ll 8)$ 
5:      $r_{4*i+1} \leftarrow (\text{b2i}(v[7*i+1]) \gg 6) | (\text{b2i}(v[7*i+2]) \ll 2) | ((\text{b2i}(v[7*i+3]) \& 0x0f) \ll 10)$ 
6:      $r_{4*i+2} \leftarrow (\text{b2i}(v[7*i+3]) \gg 4) | (\text{b2i}(v[7*i+4]) \ll 4) | ((\text{b2i}(v[7*i+5]) \& 0x03) \ll 12)$ 
7:      $r_{4*i+3} \leftarrow (\text{b2i}(v[7*i+5]) \gg 2) | (\text{b2i}(v[7*i+6]) \ll 6)$ 
8:   return  $r \in \mathcal{R}_q$ 
```

---

---

**Algorithm 8** Encoding of the public key

---

```
1: function ENCODEPK( $\hat{\mathbf{b}} \in \mathcal{R}_q, \text{publicseed} \in \{0, \dots, 255\}^{32}$ )
2:    $r \leftarrow \{0, \dots, 255\}^{7 \cdot n/4 + 32}$ 
3:    $r[0 : 7 \cdot n/4 - 1] \leftarrow \text{EncodePolynomial}(\hat{\mathbf{b}})$ 
4:    $r[7 \cdot n/4 : 7 \cdot n/4 + 31] \leftarrow \text{publicseed}[0 : 31]$ 
5:   return  $r \in \{0, \dots, 255\}^{7 \cdot n/4 + 32}$ 
```

---

---

**Algorithm 9** Decoding of the public key

---

```
1: function DECODEPK( $pk \in \{0, \dots, 255\}^{7 \cdot n/4 + 32}$ )
2:    $\hat{\mathbf{b}} \leftarrow \text{DecodePolynomial}(pk[0 : 7 \cdot n/4 - 1])$ 
3:    $\text{seed} \leftarrow pk[7 \cdot n/4 : 7 \cdot n/4 + 31]$ 
4:   return  $(\hat{\mathbf{b}} \in \mathcal{R}_q, \text{seed} \in \{0, \dots, 255\}^{32})$ 
```

---

---

**Algorithm 10** Message encoding

---

```
1: function ENCODE( $\mu \in \{0, \dots, 255\}^{32}$ )
2:    $\mathbf{v} \leftarrow \mathcal{R}_q$ 
3:   for  $i$  from 0 to 31 do
4:     for  $j$  from 0 to 7 do
5:        $mask \leftarrow -((msg[i] \gg j) \& 1)$ 
6:        $v_{8*i+j+0} \leftarrow mask \& (q/2)$ 
7:        $v_{8*i+j+256} \leftarrow mask \& (q/2)$ 
8:       if  $n$  equals 1024 then
9:          $v_{8*i+j+512} \leftarrow mask \& (q/2)$ 
10:         $v_{8*i+j+768} \leftarrow mask \& (q/2)$ 
11:   return  $\mathbf{v} \in \mathcal{R}_q$ 
```

---

---

**Algorithm 11** Message decoding

---

```
1: function DECODE( $\mathbf{v} \in \mathcal{R}_q$ )
2:    $\mu \leftarrow \{0, \dots, 255\}^{32}$ 
3:   for  $i$  from 0 to 255 do
4:      $t \leftarrow |(v_{i+0} \bmod q) - (q-1)/2|$ 
5:      $t \leftarrow t + |(v_{i+256} \bmod q) - (q-1)/2|$ 
6:     if  $n$  equals 1024 then
7:        $t \leftarrow t + |(v_{i+512} \bmod q) - (q-1)/2|$ 
8:        $t \leftarrow t + |(v_{i+768} \bmod q) - (q-1)/2|$ 
9:        $t \leftarrow ((t - q))$ 
10:    else
11:       $t \leftarrow ((t - q/2))$ 
12:     $t \leftarrow t \gg 15$ 
13:     $\mu[i \gg 3] \leftarrow \mu[i \gg 3] \vee (t \ll (i \& 7))$ 
14:   return  $\mu \in \{0, \dots, 255\}^{32}$ 
```

---

## 4. Ожидаемый уровень безопасности.

Мы оцениваем следующие уровни безопасности для двух версий нашей схемы по шкале от 1 до 5:

- ***NewHope512:***

Уровень 1 (эквивалент AES128, то есть  $2^{170}/\text{MAXDEPTH}$  квантовых вентилей или  $2^{143}$  классических вентиля) с заявленной постквантовой битовой безопасностью 101 бит.

- ***NewHope1024:***

Уровень 5 (эквивалент AES256, то есть  $2^{298}/\text{MAXDEPTH}$  квантовых вентилей или  $2^{272}$  классических вентилей) с заявленной постквантовой битовой безопасностью 233 бита.

Приведенные выше утверждения относятся к любому значению  $\text{MAXDEPTH} \geq 2^{40}$ . Действительно, этого уровня легче достичь по мере увеличения MAXDEPTH (поскольку безопасность AES уменьшается с увеличением MAXDEPTH, а MAXDEPTH не влияет на анализ безопасности нашей схемы). В отличие от атак на AES, самые быстрые атаки на нашу схему задействуют очень большие объемы памяти (не менее  $2^{79}$  бит для NewHope512 и не менее  $2^{182}$  бит для NewHope1024), что делает прямое сравнение гейткоунтов менее актуальным.

## 5. Преимущества и недостатки.

NewHope — это быстрая, эффективная и простая схема, которая является подходящей заменой RSA и ECC. Основные преимущества NewHope:

- ***Высокая производительность.***

NewHope был реализован на широком спектре платформ и показал очень хорошую производительность и имеет разумный размер ключа и зашифрованных текстов.

- ***Простота и легкость реализации.***

Базовая реализация NewHope очень проста и может быть выполнена всего несколькими строками кода в таком инструменте, как SageMath. Сложность окончательной эталонной реализации в основном связана с функциями кодирования и декодирования, а также с конкретной реализацией NTT. Кроме того, разница между наборами параметров остается минимальной, поскольку между NewHope512 и NewHope1024 изменяются только  $n$  и  $\gamma$ . Более того, код NewHope-Usenix уже портирован на различные языки программирования. Успешная интеграция NewHope-Usenix в Google Chrome и OpenSSL/Apache показывает пригодность для использования в гибридной конфигурации.

- ***Эффективность памяти.***

Неявное использование NTT позволяет эффективно использовать память на месте вычислений. Никаких больших временных структур данных не требуется.

- ***Консервативный дизайн.***

NewHope1024 обладает значительным запасом безопасности и основан на консервативном анализе безопасности, который оставляет место для улучшений в криптоанализе. Более того, схема разработана так, чтобы быть несколько устойчивой к неправомерному использованию: например, утечка информации из системного генератора случайных чисел затруднена, потому что мы всегда хешируем случайные монеты перед их использованием.

- ***Безопасность реализации.***

Хотя необходимы дополнительные усилия по безопасности реализации, уже существуют некоторые работы, связанные с криптографией на основе решетки и схемами, подобными NewHope. Некоторые из вариантов дизайна приводят к определенным компромиссам, и мы пришли к выводу, что принимаем некоторые недостатки нашего дизайна из-за преимуществ, которые мы получаем в других областях (например, скорость, производительность, простота).

Тем не менее, алгоритм NewHope обладает также рядом недостатков. Далее мы перечислим основные:

- ***Малое распространение шума.***

Выбор  $k = 8$  был сделан в качестве компромисса для обоих наборов параметров, чтобы добиться незначительной частоты ошибок дешифрования и упростить выборку, поскольку мы можем получить доступ к случайности побайтно. Однако некоторую безопасность можно получить, оптимизировав  $k$  для  $n = 512$  и  $n = 1024$ , а также для большей безопасности в эфемерном варианте Диффи-Хеллмана, где правильность менее важна (например, оригинальный NewHope-Usenix).

- ***Кольцо-LWE.***

Использование проблемы Ring-LWE является основой для хорошей производительности и простоты NewHope, и в настоящее время неизвестны атаки, которые могут использовать структуру добавления. Однако, стандартное предположение LWE можно считать более консервативным и, следовательно, лучшим выбором на случай, если следующие годы приведут к прогрессу в криптоанализе RLWE.

- ***Ограниченная параметризация.***

С текущей структурой NewHope сложно построить схему, которая достигает категории безопасности NIST 2, 3 или 4, поскольку необходимо использовать размер кольца  $n = 512$  или  $n = 1024$ .

- ***Ограничения из-за использования NTT.***

Мы используем NTT в нашей базовой схеме CPA-безопасности из соображений эффективности и выводим элементы в домене NTT. Предыдущие исследования показывают, что NTT является очень удобным способом реализации полиномиального умножения на различных платформах, особенно для больших размерностей  $n$ . Тем не менее, этот выбор дизайна также несколько ограничивает разработчика от выбора полиномиального алгоритма умножения по своему выбору, такого как умножение Нуссбаумера, Карацубы или, по крайней мере, приводит к влиянию на производительность. Если используется другой алгоритм полиномиального умножения, все равно требуется преобразовать элементы в домен NTT с нашими точными параметрами.

## **6. Список литературы:**