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-KEM или NewHope-CCA-KEM. В случае, если уровень безопасности должен быть указан вместе со схемой, мы используем примерное обозначение NewHope1024-CPA-KEM для ссылки на схему NewHope-CPA-KEM, созданную с набором параметров NewHope1024. В таблице 2 мы приводим размеры открытого ключа, секретного ключа и зашифрованного текста для наших двух KEM, 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 γ	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:

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

- NewHope1024:

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

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 $q \equiv 1 \bmod (2n)$, 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 γ 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 $q \equiv 1 \bmod (2n)$. 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 (NewHopeCPA-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, γ, 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 $v \leftarrow SHAKE256(64, seed)$, где мы хэшируем 32-байтовое случайное начальное число, обозначенное как начальное, и выводим массив байтов v с 64 элементами в диапазоне $\{0,...,255\}$.

Algorithm 1 NewHope-CPA-PKE Key Generation

function NewHope-CPA-PKE.Gen()
  $seed \xleftarrow{\$} \{0, \dots, 255\}^{32}$
  $z \leftarrow \mathsf{SHAKE256}(64, seed)$
  $publicseed \leftarrow z[0{:}31]$
  $noiseseed \leftarrow z[32{:}63]$
  $\hat{\mathbf{a}} \leftarrow \mathsf{GenA}(publicseed)$
  $\mathbf{s} \leftarrow \mathsf{PolyBitRev}(\mathsf{Sample}(noiseseed, 0))$
  $\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$
  $\mathbf{e} \leftarrow \mathsf{PolyBitRev}(\mathsf{Sample}(noiseseed, 1))$
  $\hat{\mathbf{e}} \leftarrow \mathsf{NTT}(\mathbf{e})$
  $\hat{\mathbf{b}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
  return $(pk = \mathsf{EncodePK}(\hat{\mathbf{b}}, publicseed), sk = \mathsf{EncodePolynomial}(\hat{\mathbf{s}}))$

Algorithm 2 NewHope-CPA-PKE Encryption

function NewHope-CPA-PKE.Encrypt($pk \in \{0, \dots, 255\}^{7 \cdot n/4+32}$, $\mu \in \{0, \dots, 255\}^{32}$, $coin \in \{0, \dots, 255\}^{32}$)
  $(\hat{\mathbf{b}}, publicseed) \leftarrow \mathsf{DecodePk}(pk)$
  $\hat{\mathbf{a}} \leftarrow \mathsf{GenA}(publicseed)$
  $\mathbf{s}' \leftarrow \mathsf{PolyBitRev}(\mathsf{Sample}(coin, 0))$
  $\mathbf{e}' \leftarrow \mathsf{PolyBitRev}(\mathsf{Sample}(coin, 1))$
  $\mathbf{e}'' \leftarrow \mathsf{Sample}(coin, 2)$
  $\hat{\mathbf{t}} \leftarrow \mathsf{NTT}(\mathbf{s}')$
  $\hat{\mathbf{u}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{t}} + \mathsf{NTT}(\mathbf{e}')$
  $\mathbf{v} \leftarrow \mathsf{Encode}(\mu)$
  $\mathbf{v}' \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{b}} \circ \hat{\mathbf{t}}) + \mathbf{e}'' + \mathbf{v}$
  $h \leftarrow \mathsf{Compress}(\mathbf{v}')$
  return $c = \mathsf{EncodeC}(\hat{\mathbf{u}}, h)$

Algorithm 3 NewHope-CPA-PKE Decryption

function NewHope-CPA-PKE.Decrypt($c \in \{0, \dots, 255\}^{7\frac{n}{4}+3\frac{n}{8}}$, $sk \in \{0, \dots, 255\}^{7 \cdot n/4}$)
  $(\hat{\mathbf{u}}, h) \leftarrow \mathsf{DecodeC}(c)$
  $\hat{\mathbf{s}} \leftarrow \mathsf{DecodePolynomial}(sk)$
  $\mathbf{v}' \leftarrow \mathsf{Decompress}(h)$
  $\mu \leftarrow \mathsf{Decode}(\mathbf{v}' - \mathsf{NTT}^{-1}(\hat{\mathbf{u}} \circ \hat{\mathbf{s}}))$
  return μ

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 $i \le j$, где x соответствует байту с i по j из v. По $r \leftarrow \{0,...,255\}^x$ мы объявляем, что r является байтовым массивом длины x. Используя аналогичное обозначение, через мы объявляем, что переменная r является полиномом в R_q, где все коэффициенты равны нулю. Для битовых операций мы используем операторы $\gg, \ll$, | и &, как и в контексте языка программирования C. Таким образом, $x \gg i$ для положительных целых чисел i, x обозначает сдвиг вправо на i. Тот же оператор можно применить к байту массива байтов, так что $y[j] \gg i$ для положительного целого числа i, j представляет сдвиг вправо на i j-го байта массива байтов y. Сдвиг влево обозначается как $x \ll i$ 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 $y[j] \ll i$ для положительного целого числа i, j предполагается неявное сокращение по модулю 2^8.Оператор $a \mid b$ 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 $z = b2i(a[i])$. 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, $coin \in \{0,...,255\}^{32}$, которое должно быть получено от генератора истинных случайных значений. Это необходимо для прямого использования NewHope-CPA-PKE.Encrypt в стандартных преобразованиях CCA. Расшифровка является детерминированной и не требует случайных значений. Для распределения секрета и ошибки RLWE мы используем центрированное биномиальное распределение ψ_k параметра k=8. В общем случае можно произвести выборку из ψ_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 ψ_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 ψ_8 в алгоритме 4 как функцию Sample, которая принимает в качестве входных данных 32-байтовое начальное число и целочисленный параметр $0 \le nonce < 2^8$ 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 ψ_8.

Algorithm 4 Deterministic sampling of polynomials in $\mathcal{R}_q$ from ψ_8^n

```
function SAMPLE(seed ∈ {0,...,255}^32, positive integer nonce)
    r ← R_q
    extseed ← {0,...,255}^34
    extseed[0:31] ← seed[0:31]
    extseed[32] ← nonce
    for i from 0 to (n/64) − 1 do
        extseed[33] ← i
        buf ← SHAKE256(128, extseed)
        for j from 0 to 63 do
            a ← buf[2 * j]
            b ← buf[2 * j + 1]
            r_{64*i+j} = HW(a) + q − HW(b) mod q
    return r ∈ R_q
```

Многочлены и НТТ.

Основными математическими объектами, которыми оперируют в NewHope, 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 $i \in \{0, ..., n-1\}$.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 $R_q = \mathbf{Z}_q[X]/(X^n+1)$ 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 корень n-й степени из единицы ω и его квадратный корень $\gamma = \sqrt{\omega} \bmod(q)$. Путем умножения по коэффициентам на степени γ 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 ω – 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 $state \leftarrow SHAKE128Absorb(d)$ принимают в качестве входных данных массив байтов d. Он выводит массив байтов длиной 200, который представляет состояние после поглощения d. Для получения псевдослучайных значений buf используется $state \leftarrow SHAKE128Squeeze(j, state)$. 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 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($\hat{b}, seed$), 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 c кодируется как массив из 1088 байт (n = 512) или 2176 байт (n = 1024) с помощью 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 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 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 μ, 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 $\mu \in \{0,...,255\}^{32}$ кодируется в коэффициенты $\lfloor n/256 \rfloor$ с помощью Encode (см. Алгоритм 10). Функция декодирования 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 $\{0,...,q-1\}$, вычтите $\lfloor q/2 \rfloor$ 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 $\hat{\mathbf{a}}$ by expansion of a seed

```
function GenA(seed ∈ {0,...,255}^32)
    â ← R_q
    extseed ← {0,...,255}^33
    extseed[0 : 31] ← seed[0:31]
    for i from 0 to (n/64) − 1 do
        ctr ← 0
        extseed[32] ← i
        state ← SHAKE128Absorb(extseed)
        while ctr < 64 do
            buf, state ← SHAKE128Squeeze(1, state)
            j ← 0
            for j < 168 and ctr < 64 do
                val ← b2i(buf[j]) |(b2i(buf[j + 1]) ≪ 8)
                if val < 5 · q then
                    â_{i*64+ctr} ← val
                    ctr ← ctr + 1
                j ← j + 2
    return â ∈ R_q
```

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

function EncodePolynomial($\hat{\mathbf{s}}$)
  $r \leftarrow \{0, \ldots, 255\}^{7 \cdot n/4}$
  for i **from** 0 **to** $n/4 - 1$ **do**
    $t0 \leftarrow \hat{s}_{4*i+0} \mod q$
    $t1 \leftarrow \hat{s}_{4*i+1} \mod q$
    $t2 \leftarrow \hat{s}_{4*i+2} \mod q$
    $t3 \leftarrow \hat{s}_{4*i+3} \mod q$
    $r[7*i+0] \leftarrow t0\&\texttt{0xff}$
    $r[7*i+1] \leftarrow (t0 \gg 8)\,|(t1 \ll 6)\&\texttt{0xff}$
    $r[7*i+2] \leftarrow (t1 \gg 2)\&\texttt{0xff}$
    $r[7*i+3] \leftarrow (t1 \gg 10)\,|(t2 \ll 4)\&\texttt{0xff}$
    $r[7*i+4] \leftarrow (t2 \gg 4)\&\texttt{0xff}$
    $r[7*i+5] \leftarrow (t2 \gg 12)\,|(t3 \ll 2)\&\texttt{0xff}$
    $r[7*i+6] \leftarrow (t3 \gg 6)\&\texttt{0xff}$
  return $r \in \{0, \ldots, 255\}^{7 \cdot n/4}$

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

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

Algorithm 8 Encoding of the public key

function EncodePK($\hat{\mathbf{b}} \in \mathcal{R}_q, \mathit{publicseed} \in \{0, \ldots, 255\}^{32}$)
  $r \leftarrow \{0, \ldots, 255\}^{7 \cdot n/4+32}$
  $r[0 : 7 \cdot n/4 - 1] \leftarrow \mathsf{EncodePolynomial}(\hat{\mathbf{b}})$
  $r[7 \cdot n/4 : 7 \cdot n/4 + 31] \leftarrow \mathit{publicseed}[0 : 31]$
  return $r \in \{0, \ldots, 255\}^{7 \cdot n/4+32}$

Algorithm 9 Decoding of the public key

function DecodePK($pk \in \{0, \ldots, 255\}^{7 \cdot n/4+32}$)
  $\hat{\mathbf{b}} \leftarrow \mathsf{DecodePolynomial}(pk[0 : 7 \cdot n/4 - 1])$
  $\mathit{seed} \leftarrow pk[7 \cdot n/4 : 7 \cdot n/4 + 31])$
  return $(\hat{\mathbf{b}} \in \mathcal{R}_q, \mathit{seed} \in \{0, \ldots, 255\}^{32})$

Algorithm 10 Message encoding

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

Algorithm 11 Message decoding

function DECODE($\mathbf{v} \in \mathcal{R}_q$)
  $\mu \leftarrow \{0, \ldots, 255\}^{32}$
  for i **from** 0 **to** 255 **do**
    $t \leftarrow |(v_{i+0} \mod q) - (q-1)/2|$
    $t \leftarrow t + |(v_{i+256} \mod q) - (q-1)/2|$
    if n **equals** 1024 **then**
      $t \leftarrow t + |(v_{i+512} \mod q) - (q-1)/2|$
      $t \leftarrow t + |(v_{i+768} \mod q) - (q-1)/2|$
      $t \leftarrow ((t - q))$
    else
      $t \leftarrow ((t - q/2))$
    $t \leftarrow t \gg 15$
    $\mu[i \gg 3] \leftarrow \mu[i \gg 3]\,|(t \ll (i\,\&\,7))$
  return $\mu \in \{0, \ldots, 255\}^{32}$

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}/MAXDEPTH квантовых вентилей или 2^{143} классических вентиля) с заявленной постквантовой битовой безопасностью 101 бит.

- ***NewHope1024:***

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

Приведенные выше утверждения относятся к любому значению 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 γ. 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 удобный способ 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: