Contents

1 Written specification ........................................ 4
  1.1 Overview ............................................. 4
  1.2 Mathematical definitions ................................ 4
  1.3 Parameters .......................................... 5
    1.3.1 n ............................................. 5
    1.3.2 k ............................................. 5
    1.3.3 seed_bits ..................................... 5
    1.3.4 coin_bits ..................................... 5
    1.3.5 shared_key_bits ................................. 5
  1.4 Derived constants ..................................... 5
    1.4.1 logq .......................................... 5
    1.4.2 q ............................................. 5
    1.4.3 s3_packed_bits ................................ 6
    1.4.4 owcpa_public_key_bits .......................... 6
    1.4.5 owcpa_private_key_bits ......................... 6
    1.4.6 owcpa_ciphertext_bits .......................... 6
    1.4.7 cca_public_key_bits ............................. 6
    1.4.8 cca_private_key_bits ............................ 6
    1.4.9 cca_ciphertext_bits ............................. 6
  1.5 Summary of recommended parameters and derived constants ..... 7
  1.6 Externally defined algorithms .......................... 7
    1.6.1 SHAKE128 ...................................... 7
  1.7 Arithmetic Algorithms .................................. 7
    1.7.1 S3_to_R ....................................... 7
    1.7.2 S3_to_Zx ...................................... 8
    1.7.3 Sq_to_Zx ...................................... 8
    1.7.4 Rq_to_Zx ...................................... 9
    1.7.5 S2_inverse .................................... 9
    1.7.6 S3_inverse .................................... 9
    1.7.7 Sq_inverse .................................... 9
  1.8 Sampling Algorithms .................................. 10
    1.8.1 Sample_T ...................................... 10
    1.8.2 Sample_Tplus .................................. 10
  1.9 Encoding Algorithms .................................. 11
    1.9.1 Rq_to_bits .................................... 11
    1.9.2 Rq_from_bits .................................. 11
    1.9.3 S3_to_bits .................................... 12
    1.9.4 S3_from_bits .................................. 12
  1.10 Key Encapsulation Mechanism .......................... 13
    1.10.1 Generate_Key .................................. 13
    1.10.2 Generate_Private_Key .......................... 13
    1.10.3 Generate_Public_Key ........................... 13
    1.10.4 Encapsulate ................................... 14
    1.10.5 Decapsulate ................................... 14
    1.10.6 NTRU_OWF_Public ................................ 15
    1.10.7 NTRU_OWF_Private ................................ 16
2 Performance analysis
   2.1 Description of platform ............................................................... 16
   2.2 Time ......................................................................................... 16
   2.3 Space ....................................................................................... 17
   2.4 How parameters affect performance ............................................. 17
   2.5 Optimizations ........................................................................... 17

3 Known Answer Test values .............................................................. 17

4 Expected security strength .............................................................. 17
   4.1 Security definitions .................................................................... 17
   4.2 Rationale ................................................................................... 17

5 Analysis with respect to known attacks ........................................... 18
   5.1 Lattice attacks .......................................................................... 18
   5.2 Cost of SVP Algorithms. ............................................................. 19
   5.3 Core-SVP Cost Estimates. ............................................................ 20
     5.3.1 Primal Attack. ..................................................................... 20
   5.4 Hybrid attack ........................................................................... 21
   5.5 Attacks on symmetric primitives ................................................ 23

6 Advantages and limitations .............................................................. 23
   6.1 Compared with Standard NTRU. .................................................. 23
   6.2 Compared with Streamlined NTRUPrime. .................................... 24
     6.2.1 Advantages. ....................................................................... 24
     6.2.2 Disadvantages. ................................................................. 24
   6.3 Compared with LWE systems. ..................................................... 24
1 Written specification

1.1 Overview

NTRU-HRSS is a OWCPA-secure public key encryption scheme that was introduced in [15]. It is a direct parameterization of NTRUEncrypt as described in [12]. Its constructional novelty lies entirely in the choice of sample spaces for messages, blinding polynomials, and private keys. These spaces were chosen so that 1) NTRU-HRSS is correct (decryption never fails), 2) it admits a simple and efficient constant time implementation, and 3) it avoids the extraneous parameters common to other instantiations of NTRU.

NTRU-HRSS-KEM is a CCA2-secure KEM that was also introduced in [15]. The construction uses a generic transformation from a OWCPA-secure public key encryption scheme. As a direct KEM transform, it avoids the NAEP padding mechanism used in standard NTRUEncrypt [16]. A similar NTRU-based KEM was proposed by Martijn Stam in 2005 [21]; the main differences with that work are the underlying choice of parameters for NTRUEncrypt, and the inclusion of an additional hash that is appended to the ciphertext. The additional hash allows for a proof of security in the quantum-accessible random oracle model.

For further justification of design decisions see [15].

1.2 Mathematical definitions

1. \((\mathbb{Z}/n)^x\) is the multiplicative group of integers modulo \(n\).

2. \(\Phi_n\) is the polynomial \((x^n - 1)/(x - 1) = x^{n-1} + x^{n-2} + \cdots + 1\).

3. \(R\) is the quotient ring \(\mathbb{Z}[x]/(x^n - 1)\).

4. \(S\) is the quotient ring \(\mathbb{Z}[x]/(\Phi_n)\).

5. \(R/q\) is the quotient ring \(\mathbb{Z}[x]/(q, x^n - 1)\).

6. \(S/2\) is the quotient ring \(\mathbb{Z}[x]/(2, \Phi_n)\).

7. \(S/3\) is the quotient ring \(\mathbb{Z}[x]/(3, \Phi_n)\).

8. \(S/q\) is the quotient ring \(\mathbb{Z}[x]/(q, \Phi_n)\).

9. \(\text{Mod } R/q(a)\) is the canonical \(R/q\)-representative of the polynomial \(a\). The canonical \(R/q\)-representative of \(a\) is the unique polynomial \(b\) of degree at most \(n-1\) with coefficients in \(-q/2, -q/2+1, \ldots q/2-1\) such that \(a\) and \(b\) are equivalent as elements of \(R/q\).

10. \(\text{Mod } S/2(a)\) is the canonical \(S/2\)-representative of the polynomial \(a\). The canonical \(S/2\)-representative of \(a\) is the unique polynomial \(b\) of degree at most \(n-2\) with coefficients in \(\{0, 1\}\) such that \(a\) and \(b\) are equivalent as elements of \(S/2\).

11. \(\text{Mod } S/3(a)\) is the canonical \(S/3\)-representative of the polynomial \(a\). The canonical \(S/3\)-representative of \(a\) is the unique polynomial \(b\) of degree at most \(n-2\) with coefficients in \(-1, 0, 1\) such that \(a\) and \(b\) are equivalent as elements of \(S/3\).

12. \(\text{Mod } S/q(a)\) is the canonical \(S/q\)-representative of the polynomial \(a\). The canonical \(S/q\)-representative of \(a\) is the unique polynomial \(b\) of degree at most \(n-2\) with coefficients in \(-q/2, -q/2+1, \ldots q/2-1\) such that \(a\) and \(b\) are equivalent as elements of \(S/q\).

Implementation notes:

1. It may be more natural, or more efficient, to represent elements of \(S/3\) with coefficients in \(\{0, 1, 2\}\) and/or elements of \(R/q\) with coefficients in \(\{0, 1, \ldots, q-1\}\). The algorithms below are written to be independent of the coefficient range for canonical representatives. However, the degree requirements are strict.
1.3 Parameters

1.3.1 n
An odd prime integer that satisfies the following two conditions:
- the order of 2 in \((\mathbb{Z}/n)^\times\) is \(n - 1\),
- the order of 3 in \((\mathbb{Z}/n)^\times\) is \(n - 1\).

Recommended value: 701

1.3.2 k
The Sample_T uniformity parameter. Algorithm 1.8.1 uses \(2 \cdot k\) pseudorandom bits to sample a single value in \{-1, 0, 1\}.

Recommended value: 2

1.3.3 seed_bits
The length of the seed used by the pseudorandom generator in Algorithm 1.8.1.

Recommended value: 256

1.3.4 coin_bits
The length of random bitstring used in encapsulation.

Recommended value: 256

1.3.5 shared_key_bits
The number of bits of key material produced by the KEM.

Recommended value: 256

1.4 Derived constants
The constants in this section are all functions of \(n\). Descriptions of algorithms in later sections use these constants without explicit reference to \(n\). Recommended values assume \(n = 701\).

1.4.1 logq
The smallest positive integer such that the key encapsulation mechanism is correct with \(q = 2^{\log q}\). See [15] for the proof of correctness.

Formula: \(\lceil 7/2 + \log_2(n) \rceil\)

Recommended value: 13

1.4.2 q
The smallest power of two guaranteeing correctness of the key encapsulation mechanism.

Formula: \(2^{\log q}\)

Recommended value: 8192
1.4.3 s3_packed_bits
Bit-length of the output of S3_to_bits (Section 1.9.3).
Formula: $8 \cdot \lceil (n - 1)/5 \rceil$
Recommended value: 1120

1.4.4 owcpa_public_key_bits
Bit-length of the output of Rq_to_bits (Section 1.9.1).
Formula: $(n - 1) \cdot \log q$
Recommended value: 9100

1.4.5 owcpa_private_key_bits
Bit-length of two elements of $S/3$ encoded using S3_to_bits (Section 1.9.3).
Formula: $2 \cdot s3_packed_bits$
Recommended value: 2240

1.4.6 owcpa_ciphertext_bits
Bit-length of one elements of $R/q$ encoded using Rq_to_bits (Section 1.9.1).
Formula: $(n - 1) \cdot \log q$
Recommended value: 9100

1.4.7 cca_public_key_bits
Formula: owcpa_public_key_bits
Recommended value: 9100

1.4.8 cca_private_key_bits
Formula: owcpa_private_key_bits + owcpa_public_key_bits
Recommended value: 10220

1.4.9 cca_ciphertext_bits
Formula: owcpa_ciphertext_bits + s3_packed_bits
Recommended value: 10220
1.5 Summary of recommended parameters and derived constants

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>701</td>
</tr>
<tr>
<td>k</td>
<td>2</td>
</tr>
<tr>
<td>seed_bits</td>
<td>256</td>
</tr>
<tr>
<td>coin_bits</td>
<td>256</td>
</tr>
<tr>
<td>shared_key_bits</td>
<td>256</td>
</tr>
<tr>
<td>logq</td>
<td>13</td>
</tr>
<tr>
<td>q</td>
<td>8192</td>
</tr>
<tr>
<td>s3_packed_bits</td>
<td>1120</td>
</tr>
<tr>
<td>owcpa_public_key_bits</td>
<td>9100</td>
</tr>
<tr>
<td>owcpa_private_key_bits</td>
<td>2240</td>
</tr>
<tr>
<td>owcpa_ciphertext_bits</td>
<td>9100</td>
</tr>
<tr>
<td>cca_public_key_bits</td>
<td>9100</td>
</tr>
<tr>
<td>cca_private_key_bits</td>
<td>10220</td>
</tr>
<tr>
<td>cca_ciphertext_bits</td>
<td>10220</td>
</tr>
</tbody>
</table>

Table 1: Recommended parameters and derived constants

1.6 Externally defined algorithms

1.6.1 SHAKE128

Input:
- A bitstring $M$ of arbitrary length.
- The output length parameter $d$.

Output:
- A bitsring of length $d$.

Description:
1. Output KECCAK[256]($M || 1111, d$), as defined in [20].

1.7 Arithmetic Algorithms

Algorithms for integer addition, integer multiplication, polynomial addition, polynomial multiplication, and modular reduction ($\text{Mod}_R$, $\text{Mod}_S2$, $\text{Mod}_S3$, and $\text{Mod}_Sq$) are omitted.

1.7.1 $S3_to_R$

Input:
- A polynomial $a$.

Output:
- A polynomial $b$ of degree at most $n - 1$ that satisfies:
  - $b \equiv 0 \mod (x - 1)$,
  - $b \equiv a \mod (p, \Phi_n)$.
Description:
1. Set $v_0 = S3\_inverse(x - 1)$ \[1.7.6\]
2. Set $v_1 = v_0 \cdot a$
3. Set $v_2 = S3\_to\_Zx(v_1)$ \[1.7.2\]
4. Set $b = (x - 1) \cdot v_2$
5. Output $b$

Implementation notes:
1. The result $b$ is of degree at most $n - 1$ and has coefficients in $\{-1, 0, 1\}$.
2. The explicit value of $S3\_inverse(x - 1)$ is

$$v_0 = \text{Mod}_{S3}\left((n - 1) + \sum_{i=1}^{n-1} n \cdot (1 + i - n) \cdot x^i\right).$$

3. Pseudocode for a fast implementation is given in [15].

1.7.2 $S3\_to\_Zx$

Input:
− A polynomial $a$.

Output:
− The unique polynomial $b$ of degree at most $n - 2$ with coefficients in $\{-1, 0, 1\}$ such that $a$ and $b$ are equivalent as elements of $S/3$.

Description:
1. Output $\text{Mod}_S3(a)$.

Implementation notes:
1. The set of canonical $S/3$-representatives is defined so that $S3\_to\_Zx$ is trivial. An implementation that uses a different set of $S/3$-representatives may replace a call to $S3\_to\_Zx$ with a contextually equivalent routine. For example, when a call to $S3\_to\_Zx$ is followed by an operation in $R/q$ the implementation may combine the lift to $\mathbb{Z}[x]$ and the reduction modulo $q$ into a single operation.

1.7.3 $Sq\_to\_Zx$

Input:
− A polynomial $a$.

Output:
− The unique polynomial $b$ of degree at most $n - 2$ with coefficients in $\{-q/2, -q/2 + 1, \ldots, q/2 - 1\}$ such that $a$ and $b$ are equivalent as elements of $S/q$.

Description:
1. Output $\text{Mod}_{Sq}(a)$. 
1.7.4 \( R_{q \rightarrow Z} x \)

**Input:**
- A polynomial \( a \).

**Output:**
- The unique polynomial \( b \) of degree at most \( n - 1 \) with coefficients in \( \{-q/2, -q/2 + 1, \ldots, q/2 - 1\} \) such that \( a \) and \( b \) are equivalent as elements of \( R/q \).

**Description:**
1. Output \( \text{Mod}_{R_{q}}(a) \).

1.7.5 \( S_{2}^{-1} \)

**Input:**
- A polynomial \( a \).

**Output:**
- An \( S/2 \)-representative \( b \) that satisfies \( \text{Mod}_{S_{2}}(a \cdot b) = 1 \).

**Implementation notes:**
1. Inverting an element of \( S/2 \) in constant time is nontrivial. Pseudocode for one method is provided in [15].

1.7.6 \( S_{3}^{-1} \)

**Input:**
- A polynomial \( a \).

**Output:**
- An \( S/3 \)-representative \( b \) that satisfies \( \text{Mod}_{S_{3}}(a \cdot b) = 1 \).

**Implementation notes:**
1. Inverting an element of \( S/3 \) in constant time is nontrivial. Pseudocode for one method is provided in [15].

1.7.7 \( S_{q}^{-1} \)

**Input:**
- A polynomial \( a \).

**Output:**
- An \( S/q \)-representative \( b \) that satisfies \( \text{Mod}_{S_{q}}(a \cdot b) = 1 \).

**Description:**
1. Set \( v_0 = S_{2}^{-1}(a) \) [1.7.5]
2. Set \( t = 2 \)
3. While \( t < q \)
4. Set \( v_0 = \text{Mod}_{S_{q}}(v_0 \cdot (2 - a \cdot v_0)) \)
5. Set \( t = t \cdot t \)
6. Endwhile
7. Output $\text{Mod} \_\text{Sq}(v_0)$

Implementation notes:
1. When $\text{Sq}^{-1}$ is called from $\text{Generate \_Public \_Key}$ [1.10.3] it is safe to replace the calls to $\text{Mod} \_\text{Sq}$ in Lines 4 and 7 with calls to $\text{Mod} \_\text{Rq}$.

1.8 Sampling Algorithms

1.8.1 Sample_T

Input:
- A bitstring $\text{seed}$ of length $\text{seed\_bits}$.
- A bitstring $\text{domain}$ of length 64.

Output:
- An $S/3$-representative.

Description:
1. Set $v = 0$ (The zero polynomial)
2. Set $i = 0$
3. Set $\ell = 2 \cdot k \cdot (n - 1)$
4. Set $b_1b_2 \ldots b_\ell = \text{SHAKE128}(\text{domain} \parallel \text{seed}, \ell)$
5. While $i < n - 1$
6. Set $v_i = \sum_{j=1}^{k} b_{2ki+j} - b_{2ki+k+j}$
7. Set $i = i + 1$
8. Endwhile
9. Output $\text{Mod} \_S3(v)$

Implementation notes:
1. In this document we use four domain strings $\text{domm}, \text{domr}, \text{domf}, \text{domg}$. In our implementation $\text{domm}$ and $\text{domr}$ are the zero string, $\text{domf} = \text{0x0100000000000000}$ and $\text{domg} = \text{0x0200000000000000}$. This will likely be changed in a future version.

1.8.2 Sample_Tplus

Input:
- A bitstring $\text{seed}$ of length $\text{seed\_bits}$.
- A bitstring $\text{domain}$ of length 64.

Output:
- The canonical $S/3$-representative of a polynomial $v = \sum_{i=0}^{n-2} v_i x^i$ where $v_i \in \{-1, 0, 1\}$ for all $i$ and $\sum_{i=1}^{n-2} v_i v_{i-1} \geq 0$.

Description:
1. Set $v = \text{Sample\_T}(\text{seed}, \text{domain})$ [1.8.1]
2. Set $t = \sum_{i=1}^{n-2} v_i \cdot v_{i-1}$
3. Set $s = -1$ if $t < 0$, otherwise set $s = 1$
4. Set $i = 0$
5. While $i < n - 2$
   6. Set $v_i = s \cdot v_i$
   7. Set $i = i + 2$
8. Endwhile
9. Output $\text{Mod}_S(v)$

**Implementation notes:**
1. The value $t$ in Line 2 satisfies $-n + 1 < t < n - 1$.

## 1.9 Encoding Algorithms

The algorithms in this section are specified at the bit level. When converting to octets a bitstring is padded with zeros until it is of length $8 \cdot \ell$ for some $\ell$. The encoding is then order preserving for indices at distance at least 8, but order reversing within octets. Hence $b_1, b_2, b_3, b_4, \ldots, b_{15}, b_{16}$ is encoded as $b_8 b_7 \ldots b_1, b_{16} b_{15} \ldots b_9$.

### 1.9.1 $R_q$ to bits

**Input:**
- A polynomial $a$ that satisfies $a(1) \equiv 0 \pmod{q}$.

**Output:**
- A bitstring $b_1 b_2 \ldots b_\ell$ of length $\ell = (n - 1) \cdot \log q$.

**Description:**
1. $v = \text{Mod}_R(a)$ (Ensure $v = \sum_{i=0}^{n-1} v_i x^i$)
2. $i = 0$
3. while $i \leq n - 2$
   4. Set $b_{\log q+1}, b_{\log q+2} \ldots b_{\log q+8}$ so that $\sum_{j=1}^{\log q} b_{\log q+j} 2^{\log q-j} \equiv v_i \pmod{q}$
   5. $i = i + 1$
6. endwhile
7. Output $b_1 b_2 \ldots b_\ell$.

**Implementation notes:**
1. The coefficient $v_{n-1}$ is not encoded. The condition $a(1) \equiv 0 \pmod{q}$ ensures that $v_{n-1}$ can be recovered from the first $n - 1$ coefficients.

### 1.9.2 $R_q$ from bits

**Input:**
- A bitstring $b_1 b_2 \ldots b_\ell$ of length $\ell = (n - 1) \cdot \log q$.

**Output:**
- An $R/q$-representative.

**Description:**
1. $v = 0$ (The zero polynomial)
2. $i = 0$
3. while $i \leq n - 2$
4. Set $c = \sum_{j=1}^{\log q} b_{j \log q + j} \cdot 2^{\log q - j}$.
5. Set $v_i = c$
6. Set $v_{n-1} = v_{n-1} - c$
7. Set $i = i + 1$
8. endwhile
9. Output Mod $Rq(v)$

1.9.3 $S_3$ to bits

Input:
- A polynomial $a$.

Output:
- A bitstring $b_1b_2\ldots b_\ell$ of length $\ell = s_3$ packed bits. [1.4.3]

Description:
1. $v = \text{Mod}_S(a)$ (Ensure $v = \sum_{i=0}^{n-2} v_ix^i$)
2. $i = 0$
3. while $i < \lfloor (n-1)/5 \rfloor$
4. Set $c_1, c_2, \ldots, c_5 \in \{0, 1, 2\}$ with $c_j \equiv v_{5,i+j} \pmod{3}$ for $1 \leq j \leq 5$.
5. Set $b_{8,i+1}b_{8,i+2}\ldots b_{8,i+8}$ so that $\sum_{j=1}^{8} b_{8,i+j}2^{8-j} = \sum_{j=1}^{5} c_j 3^{5-j}$.
6. $i = i + 1$
7. endwhile
8. Output $b_1b_2\ldots b_\ell$.

Implementation notes:
1. In Line 4 we have $c_j = 0$ if $j > n - 2$

1.9.4 $S_3$ from bits

Input:
- A bitstring $a_1a_2\ldots a_\ell$ of length $\ell = s_3$ packed bits. [1.4.3]

Output:
- An $S/3$-representative.

Description:
1. $b = 0$ (The zero polynomial)
2. $i = 0$
3. while $i < \lfloor (n-1)/5 \rfloor$
4. Set $c_1, c_2, \ldots, c_5 \in \{0, 1, 2\}$ so that $\sum_{j=1}^{8} b_{8,i+j}2^{8-j} = \sum_{j=1}^{5} c_j 3^{5-j}$.
5. Set $b_{5,i+1} = c_1$
6. Set $b_{5,i+2} = c_2$
7. Set $b_{5,i+3} = c_3$
8. Set $b_{5,i+4} = c_4$
9. Set $b_{5,i+5} = c_5$
10. $i = i + 1$
11. endwhile
12. Output $\text{Mod}_S(b)$.
1.10 Key Encapsulation Mechanism

The algorithms Generate Key [1.10.1], Encapsulate [1.10.4], and Decapsulate [1.10.5] are a key encapsulation mechanism that is CCA-secure in the quantum random oracle model. The other functions in this section are used by these algorithms but are not part of the public API.

1.10.1 Generate Key

**Input:**
- The system parameters.

**Output:**
- A bitstring \texttt{packed\_public\_key} of length \texttt{cca\_public\_key\_bits} . \[1.4.7\]
- A bitstring \texttt{packed\_private\_key} of length \texttt{cca\_private\_key\_bits} . \[1.4.8\]

**Description:**
1. Let \texttt{seed} be a string of \texttt{seed\_bits} uniform random bits.
2. Set \texttt{f,fp} = \texttt{Generate\_Private\_Key}(\texttt{seed}) \[1.10.2\]
3. Set \texttt{h} = \texttt{Generate\_Public\_Key}(\texttt{seed}, \texttt{f}) \[1.10.3\]
4. Set \texttt{packed\_public\_key} = \texttt{Rq\_to\_bits}(\texttt{h}) \[1.9.1\]
5. Set \texttt{packed\_private\_key} = \texttt{S3\_to\_bits(} \texttt{f} \texttt{) || S3\_to\_bits(} \texttt{fp} \texttt{)} \[1.9.3\]

1.10.2 Generate Private Key

**Input:**
- The system parameters.
- A bitstring \texttt{seed} of length \texttt{seed\_bits}.

**Output:**
- \texttt{S/3\_representatives} \texttt{f} and \texttt{fp} that satisfy Mod \texttt{S3(} \texttt{f} \texttt{· \texttt{fp}} \texttt{)} = 1.

**Description:**
1. Set \texttt{f} = \texttt{Sample\_Tplus(} \texttt{seed}, \texttt{domf} \texttt{)} \[1.8.2\]
2. Set \texttt{fp} = \texttt{S3\_inverse(} \texttt{f} \texttt{)} \[1.7.6\]
3. Output \texttt{f} and \texttt{fp}

1.10.3 Generate Public Key

**Input:**
- The system parameters.
- A bitstring \texttt{seed} of length \texttt{seed\_bits}.
- An \texttt{S/3\_representative} \texttt{f}.

**Output:**
- An \texttt{R/q\_representative} \texttt{h} that satisfies the following three conditions:
  1. \texttt{h(1) \equiv 0 (mod q)},
  2. Mod \texttt{Rq(} \texttt{h} \texttt{· \texttt{f}} \texttt{)} = 3 \cdot (x - 1) \cdot \texttt{g} for some \texttt{g} with coefficients in \{-1, 0, 1\},
  3. Mod \texttt{S3(} \texttt{h} \texttt{· \texttt{f}} \texttt{)} = 0.
Description:
1. Set $v_0 = \text{Sample}_\text{Tplus}(\text{seed}, \text{domg})$ \[1.8.2\]
2. Set $g = \text{S3}_\text{to}_{\text{Zx}}(v_0)$ \[1.7.2\]
3. Set $v_1 = \text{Sq}_{\text{inverse}}(f)$ \[1.7.7\]
4. Set $f_q = \text{Sq}_{\text{to}_{\text{Zx}}}(v_1)$ \[1.7.3\]
5. Set $v_2 = 3 \cdot (x - 1) \cdot g \cdot f_q$
6. Set $h = \text{Mod}_{\text{Rq}}(v_2)$
7. Output $h$

Implementation notes:
1. The lifts to $\mathbb{Z}[x]$ are trivial when canonical representatives are used. Implementations that use different sets of representatives may map $g$ and $f_q$ directly to the appropriate $R/q$-representatives without going through $\mathbb{Z}[x]$.

1.10.4 Encapsulate

Input:
- The system parameters.
- A bitstring $\text{packed}_{\text{public key}}$ of length $\text{cca}_{\text{public key}}_{\text{bits}}$. \[1.4.7\]

Output:
- A bitstring $\text{shared key}$ of length $\text{shared key}_{\text{bits}}$.
- A bitstring $\text{packed}_{\text{cca ct}}$ of length $\text{cca ciphertext}_{\text{bits}}$. \[1.4.9\]

Description:
1. Let $\text{seed}$ be a string of $\text{seed}_{\text{bits}}$ uniform random bits.
2. Set $m = \text{Sample}_\text{T}(\text{seed}, \text{domm})$ \[1.8.1\]
3. Set $\text{packed}_m = \text{S3}_\text{to}_{\text{bits}}(m)$ \[1.9.3\]
4. Set $\text{hashes} = \text{SHAKE128}(\text{packed}_m, \text{coin}_{\text{bits}} + \text{shared key}_{\text{bits}} + s3_{\text{packed}_{\text{bits}}})$ \[1.6.1\]
5. Parse $\text{hashes}$ as $\text{coins} \parallel \text{shared key} \parallel qrom_{\text{hash}}$ with
   - $\text{coins}$ of length $\text{coin}_{\text{bits}}$,
   - $\text{shared key}$ of length $\text{shared key}_{\text{bits}}$, and
   - $qrom_{\text{hash}}$ of length $s3_{\text{packed}_{\text{bits}}}$. \[1.4.3\]
6. Set $\text{packed}_\text{owcpa ct} = \text{NTRU}_{\text{OWF Public}}(\text{packed}_m, \text{packed}_{\text{public key}}, \text{coins})$. \[1.10.6\]
7. Set $\text{packed}_\text{cca ct} = \text{packed}_\text{owcpa ct} \parallel qrom_{\text{hash}}$

1.10.5 Decapsulate

Input:
- The system parameters.
- A bitstring $\text{packed}_{\text{key pair}}$ of length $\text{cca}_{\text{private key}}_{\text{bits}}$. \[1.4.8\]
- A bitstring $\text{packed}_{\text{cca ct}}$ of length $\text{cca ciphertext}_{\text{bits}}$. \[1.4.9\]

Output:
- A bitstring $\text{shared key}$ of length $\text{shared key}_{\text{bits}}$. \[1.3.5\]

Description:
1. Parse packed_key_pair as packed_private_key || packed_public_key with
   - packed_private_key of length owcpa_private_key_bits and
   - packed_public_key of length owcpa_public_key_bits.

2. Parse packed_cca_ct as packed_owcpa_ct || qrom_hash with
   - packed_owcpa_ct of length owcpa_ciphertext_bits and
   - qrom_hash of length s3_packed_bits.

3. Set packed_m = NTRU_OWF_Private(packed_private_key, packed_owcpa_ct)

4. Set hashes = SHAKE128(packed_m, coin_bits + shared_key_bits + s3_packed_bits)

5. Parse hashes as coins || shared_key || re_qrom_hash with
   - coins of length coin_bits,
   - shared_key of length shared_key_bits, and
   - re_qrom_hash of length s3_packed_bits.

6. Let re_packed_owcpa_ct = NTRU_OWF_Public(n, packed_m, packed_public_key, coins)

7. If re_packed_owcpa_ct || re_qrom_hash is bitwise equal to packed_owcpa_ct || qrom_hash

8. Output shared_key.

9. Else

10. Output the zero string of length shared_key_bits.

11. Endif

1.10.6 NTRU_OWF_Public

Input:
- The system parameters.
- A bitstring packed_m of length s3_packed_bits.
- A bitstring packed_public_key of length owcpa_public_key_bits.
- A bitstring coins of length coin_bits.

Output:
- A bitstring packed_owcpa_ct of length owcpa_ciphertext_bits.

Description:
1. Set h = Rq_from_bits(packed_public_key)
2. Set v0 = Sample_T(coins, domr)
3. Set r = S3_to_Zx(v0)
4. Set v1 = S3_from_bits(packed_m)
5. Set m = S3_to_R(v1)
6. Set e = Mod_Rq(r · h + m)
7. Set packed_owcpa_ct = Rq_to_bits(e)
1.10.7 NTRU_OWF_Private

Input:
- A bitstring `packed_private_key` of length `owcpa_private_key_bits`.
- A bitstring `packed_owcpa_ct` of length `owcpa_ciphertext_bits`.

Output:
- A bitstring `packed_m` of length `s3packed_bits`.

Description:
1. Parse `packed_private_key` as `packed_f || packed_fp` with
   - `packed_f` of length `s3packed_bits`, and
   - `packed_fp` of length `s3packed_bits`. [1.4.3]
2. Parse `packed_owcpa_ct` as `packed_owcpa_ct || qrom_hash` with
   - `packed_owcpa_ct` of length `owcpa_ciphertext_bits`, and
   - `qrom_hash` of length `s3packed_bits`. [1.4.3]
3. Set $e = Rq_{\text{from bits}}(packed_owcpa_ct)$ [1.9.2]
4. Set $v_0 = S3_{\text{from bits}}(packed_f)$ [1.9.4]
5. Set $f = S3_{\text{from Zx}}(v_0)$ [1.7.2]
6. Set $fp = S3_{\text{from bits}}(packed_fp)$ [1.9.4]
7. Set $v_1 = \text{Mod}_Rq(e \cdot f)$
8. Set $v_2 = \text{Mod}_S3(v_1 \cdot fp)$
9. Set `packed_m` = `S3_to_bits(v2)` [1.9.3]
10. Output `packed_m`.

2 Performance analysis

The results in this section are with respect to the parameters listed in Table 1.5.

2.1 Description of platform

In order to obtain benchmarks, we evaluate our reference implementation on a machine using the Intel x64-86 instruction set. In particular, we use a single core of a 3.5 GHz Intel Core i7-4770K CPU. We follow the standard practice of disabling TurboBoost and hyper-threading. The system has 32KiB L1 instruction cache, 32KiB L1 data cache, 256KiB L2 cache and 8192KiB L3 cache. Furthermore, it has 32GiB of RAM, running at 1333MHz. When performing the benchmarks, the system ran on Linux kernel 4.9.0-4-amd64, Debian 9 (Stretch). We compiled the code using GCC version 6.3.0-18, with the compiler optimization flag -O3.

We used the same platform described above to evaluate our AVX2 implementation. For the AVX2 implementation, we included the additional compiler flags ‘-march=native’ and ‘-mtune=native’.

2.2 Time

The median resulting cycle counts are listed in the table below.

<table>
<thead>
<tr>
<th></th>
<th>key generation</th>
<th>encapsulation</th>
<th>decapsulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference C</td>
<td>18151998</td>
<td>1208946</td>
<td>3578538</td>
</tr>
<tr>
<td>optimized AVX2</td>
<td>294874</td>
<td>38456</td>
<td>68458</td>
</tr>
</tbody>
</table>
2.3 Space
The public key consists of 1138 bytes, and the secret key takes 1418 bytes. The transmitted ciphertext consists of 1278 bytes. Of the ciphertext size, 140 bytes are a direct result of the transformation from the underlying OW-CPA secure scheme to the CCA2 secure KEM.

Our reference implementation uses almost 11 KiB of stack space and our AVX2 software uses just over 43 KiB, but this was not a target of optimization and should not be considered to be a lower bound.

2.4 How parameters affect performance
As the main arithmetic operations are (sub-)quadratic, we would expect that doubling $n$ would lead to at most a factor of 4 overhead in time. Indeed, preliminary tests with our reference implementation suggest that $n = 1373$ would be less than a factor of 4 times slower. Likewise we would expect memory (and communication cost) to roughly double. Given that these parameters span an large range of relevant security levels (See Table 5), it is fair to say that parameters have only a modest impact on performance.

2.5 Optimizations
We refer to [15] for a detailed discussion of the optimizations used in our AVX2 implementation.

3 Known Answer Test values
All KAT values are included in subdirectories of the directory KAT/ntruhrss701 of the submission package. The KAT values were generated by the PQCgenKAT_kem program provided by NIST. The complete list of KAT files is:

- KAT/ntruhrss701/PQCkemKAT_1418.req
- KAT/ntruhrss701/PQCkemKAT_1418.rsp

4 Expected security strength
4.1 Security definitions
NTRU-HRSS-KEM meets the standard IND-CCA2 security definition for a key encapsulation mechanism. Parameters have been chosen so that decryption failure is impossible, and a key can be reused at least $2^{64}$ times without compromising security. This follows from Dent’s proof of security for the transform in [8, Table 5.], and from the presumed one-wayness of the underlying encryption scheme, NTRU-HRSS.

4.2 Rationale
Based on the analysis in Section 5.1, we expect that violating the one-wayness of NTRU-HRSS would require computational resources greater than those required to perform a key search on AES-128.

Our security claim is based primarily on Table 4, which costs the best known classical attack at $2^{136}$ operations and $2^{136}$ space. These operations mask large factors that put the true cost of the attack well above the $2^{145}$ bit operations required to attack AES-128, even in a RAM model.

Our security claim is also based on Table 5, which costs the best known quantum attack at $2^{123}$ Grover iterations. This attack is in the quantum RAM model and requires quantum-accessible classical memory of size $2^{123}$. Again, large factors are ignored and it is unlikely that this attack maintains its advantage over the classical variant when it is instantiated in a quantum circuit model.
5 Analysis with respect to known attacks

5.1 Lattice attacks

Some background on lattices is assumed. In this section we denote the degree of \( \Phi_n \) by \( n' = n - 1 \).

Each element of \( S \) can be uniquely represented by a polynomial of the form \( v = \sum_{i=0}^{n'-1} v_i x^i \). The correspondence between these representatives and their coefficient vectors, \( v \mapsto (v_0, \ldots, v_{n'-1}) \in \mathbb{R}^{n'} \), allows one to view \( S \) as a euclidean lattice. Lattice attacks on NTRU begin from the observation that for any \( h \in S \) the set

\[
L_h := \{(a, b) \in S^2 : b \equiv a \cdot h \pmod{q}\}
\]

is, likewise, a lattice in \( \mathbb{R}^{2n'} \). A basis for \( L_h \) is given by the rows of the \( 2n' \times 2n' \) matrix

\[
[L_h] = \begin{pmatrix} I_{n'} & H \\ 0 & q \cdot I_{n'} \end{pmatrix},
\]

where the \( i \)-th row of \( H \) for \( 0 \leq i \leq n' - 1 \) is the coefficient vector of \( \text{Mod} \, Sq(x^i \cdot h) \). If \( h \) is an NTRU public key then, by construction\(^1\), the secret key corresponds to a vector \((f, g)\) in \( L_h \). Since \((f, g)\) is known to have small norm, one might hope to recover it using lattice reduction.

Attacks involving \( L_h \) have gone through several reformulations. The earliest such attack, due to Hoffstein, Pipher, and Silverman, was framed as an exact key recovery problem \([11]\). Coppersmith and Shamir later observed that any short vector in \( L_h \), not just \((f, g)\), could be used to invert the NTRU one way function \([7]\). It was this reformulation, as an approximation problem, that initiated the first serious efforts to understand the difficulty of finding short vectors in \( L_h \) \([12, 18]\).

Two additional observations, which appear as early as May’s work on the cryptanalysis of NTRU-107 \([18]\), bring us to the modern attack strategy. The first is that the ratio of successive minima \( \lambda_2(L_h)/\lambda_1(L_h) \) is a parameter of interest in assessing the difficulty of finding a short vector in \( L_h \). That is to say that the NTRU problem reduces to unique SVP. Surprisingly, this brings us back to an exact search for \((f, g)\), or one of a few related vectors, but unique SVP is known to be easier than approximate SVP in practice \([9]\). The second observation is that, when lattice reduction is expensive, it may be fruitful to guess a subset of the coefficients of \((f, g)\). May’s dimension reduction technique \([18]\) and May and Silverman’s pattern method \([19]\) trade success probability in guessing coefficients of \((f, g)\) against the cost of solving unique SVP in a sublattice of \( L_h \).

The most effective lattice attacks in the literature treat NTRU key recovery as a unique shortest vector problem. Depending on the assumed non-asymptotic cost of lattice reduction, these attacks make use of guessing (or ignoring) coefficients to reduce the lattice dimension. Recently introduced sieve algorithms have single exponential cost that is small enough to call into question the effectiveness of guessing coefficients, however it is not clear that combinatorial techniques are irrelevant in realistic models of computation.

In Section 5.3.1 we estimate the cost of solving unique SVP in \( L_h \) in the Core-SVP cost model from the NewHope paper \([3]\). We do not aim to provide a complete description of the cost model, or of unique SVP methods in general. Background and references on solving unique SVP, as well as substantial discussion of the Core-SVP cost model, can be found in Albrecht–Göpfert–Virdia–Wunderer \([1]\). While Albrecht et al. focus on LWE, their analysis of LWE with short secrets applies directly to NTRU. In fact, the attack described in \([1, \text{Section 5.1}]\) is identical\(^2\) to May’s “dimension reduction” attack applied

---

\(^1\)The output of Generate Public Key is the \( R/q \)-representative of \( 3 \cdot (x - 1) \cdot g/f \), but \( h \) here should be thought of as the \( S/q \)-representative of \( g/f \). Since \( 3 \cdot (x - 1) \) is a unit in \( S/q \) this change is purely syntactic.

\(^2\)The attack in \([1, \text{Section 5.1}]\) uses a lattice generated by the rows of

\[
\begin{pmatrix} \nu I_{n'} & -A^T \\ 0 & q \cdot I_m \\
0 & c & 1 \end{pmatrix}
\]

where \((A, c)\) is an instance of the LWE problem. By replacing \(-A^T\) by \( H \) and \( c \) by an NTRU ciphertext we get a message recovery attack on NTRU. By replacing \(-A^T\) by \( H \) and omitting the row containing \( c \) we get May’s attack. Taking \( m < n' \) corresponds to dimension reduction. The use of the parameter \( \nu \) is a standard lattice reduction trick that also appears in \([7]\) and \([18]\).
to $[L_h]$, despite being developed independently and in a different context.

In Section 5.4, we consider Howgrave-Graham’s hybrid attack [13] in the Core-SVP cost model. The hybrid attack is a unique SVP attack that uses a meet-in-the-middle strategy for guessing coefficients.

Both analyses require the following fact, which is easily derived by series multisection of $(1 + x)^{2k}$.

**Fact 1.** Let $Y$ be a random variable distributed according to the centered binomial distribution of parameter $k$. Then

\[
\begin{align*}
\Pr[Y \equiv 0 \mod 3] &= \left(1 + \frac{2}{2^{2k}}\right), \\
\Pr[Y \equiv \pm 1 \mod 3] &= \left(1 - \frac{1}{2^{2k}}\right).
\end{align*}
\]

For the recommended parameter, $k = 2$, this gives a distribution on $(-1, 0, 1)$ of $(\frac{5}{16}, \frac{6}{16}, \frac{5}{16})$. The expected euclidean length of $m$ coefficients is therefore $\sqrt{m \cdot 10/16}$, and the entropy per coefficient is approximately $1.579$ bits.

### 5.2 Cost of SVP Algorithms.

In the following sections we use four different cost estimates for SVP-$b$. The cost of the List-Decoding Sieve from [4] is summarized in Table 2. The cost of the List-Decoding Sieve when Grover search is used to answer nearest-neighbor queries is summarized in Table 3.

We estimate the cost of solving SVP by enumeration in dimension $b$ using a quasilinear fit to the experimental data of Chen and Nguyen [6]. Following [2] we use the trend line:

\[
\text{enum}(b) = 0.18728 \cdot b \log_2(b) - 1.0192 \cdot b + 16.10.
\]

The cost of using an enumeration algorithm for SVP-$b$ is then estimated as $2^{\text{enum}(b)}$.

Finally we consider a hypothetical square-root speedup in the cost of enumeration on a quantum computer, for a cost of $2^{\text{enum}(b)/2}$. While this is a purely hypothetical improvement, it is no less hypothetical than a quantum variant of the List-Decoding Sieve which maintains its asymptotic cost in a quantum circuit model. As we shall see, even this dramatic speedup in enumeration is unlikely to be competitive with the classical List-Decoding sieve, especially if a restriction is imposed on the depth of quantum computations.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Pre-quantum cost</th>
<th>Post-quantum cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Space</td>
</tr>
<tr>
<td>Balanced</td>
<td>$2^{0.292 \cdot b}$</td>
<td>$2^{0.292 \cdot b}$</td>
</tr>
<tr>
<td>Min. Space</td>
<td>$2^{0.368 \cdot b}$</td>
<td>$2^{0.208 \cdot b}$</td>
</tr>
</tbody>
</table>

Table 2: Cost of the List-Decoding Sieve as a function of the dimension, with all subexponential factors suppressed [4].

Table 3: Cost of the List-Decoding Sieve, as a function of the dimension, when Grover search is used to answer nearest neighbor queries [17]. Again subexponential factors are suppressed, but we have units of Grover iterations rather than time. This is because it is not clear that the algorithm can be instantiated in a quantum circuit model without re-evaluating its asymptotic cost.
5.3 Core-SVP Cost Estimates.

The Core-SVP estimate was introduced in the security evaluation of NewHope [3]. A thorough description can be found in [1]. Success criteria for the primal and hybrid attacks are given in the following sections.

5.3.1 Primal Attack.

The primal attack has two parameters: $b$, the blocksize used for lattice reduction, and $m$, the dimension reduction parameter\(^1\). The primal attack attempts to solve unique SVP in a sublattice $\Lambda_{\text{primal}} \subseteq L_h$ of rank $d = n' + m$ and volume $q^m$. A BKZ-$b$ reduced basis $V = \{v_1, \ldots, v_d\}$ for $\Lambda_{\text{primal}}$ is computed. Following this, a single call to an SVP-$b$ routine is made on $\{v'_{d-b+1}, v'_{d-b+2}, \ldots, v'_d\}$, where $v'_i$ is $v_i$ projected orthogonally to the first $d - b$ vectors of $V$. The parameters $b$ and $m$ are chosen so that a short vector in this projected sublattice is likely to be mapped to a short vector in $\Lambda_{\text{primal}}$ by Babai’s nearest plane algorithm.

The success condition is with respect to the length of the first vector in the last block of the reduced basis, $v_{d-b+1}$. The Gram-Schmidt vectors of the reduced basis, $v^*_1, v^*_2, \ldots, v^*_d$, are expected to satisfy

$$||v_{d-b+1}|| \leq \delta^{d/2} \cdot q^{m/d}$$

where

$$\delta = \left( \frac{\pi \cdot b^{1/b}}{b/(2 \pi \cdot e)^{1/(2(b-1))}} \right)^{1/(2(b-1))}.$$ 

The assumption that this is the case is known as the Geometric Series Assumption.

In [3], it was suggested that an attacker could expect to recover $(f, g)$ from its projection orthogonal to the first $d - b$ vectors of $V$ if

$$\sqrt{b/d} \cdot ||(f, g)|| \leq ||v^*_d|| \approx \delta^{2b-d} \cdot q^{m/d}. \quad (3)$$

Further evidence for this claim was given in [1].

From Fact 1 we have $\sqrt{b/d} \cdot ||(f, g)|| \approx \sqrt{b} \cdot \sqrt{10/16}$. The Core-SVP cost of the attack is found by minimizing the cost of one call to an algorithm for SVP-$b$ over all choices of $m$ and $b$ for which (3) is satisfied.

In Tables 4 and 5 we give optimal parameters for the primal attack in the Core-SVP model. The main entries of interest are for $n' = 700$, corresponding to our recommended parameter set. We also list the cost for $n' = 940$ and $n' = 1372$, as these give some indication for how security scales with $n$ and may be useful in comparisons with other proposals.

<table>
<thead>
<tr>
<th>Primal Attack with List-Decoding Sieve</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n'$</td>
</tr>
<tr>
<td>700</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>940</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>1372</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Table 4: Optimal parameters for the primal attack when the cost of SVP$-b$ is as given in Table 2.

\(^1\)In the LWE context $m$ is the number of LWE samples used by the attacker.
Primal Attack with List-Decoding Sieve and Grover Search

<table>
<thead>
<tr>
<th>$n'$</th>
<th>$m$</th>
<th>$b$</th>
<th>Metric</th>
<th>Grover Iterations</th>
<th>Iteration Depth</th>
<th>Vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>700</td>
<td>626</td>
<td>465</td>
<td>Balanced</td>
<td>$2^{123}$</td>
<td>$2^{26}$</td>
<td>$2^{123}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Min. Space</td>
<td>$2^{138}$</td>
<td>$2^{41}$</td>
<td>$2^{96}$</td>
</tr>
<tr>
<td>940</td>
<td>824</td>
<td>616</td>
<td>Balanced</td>
<td>$2^{163}$</td>
<td>$2^{35}$</td>
<td>$2^{163}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Min. Space</td>
<td>$2^{183}$</td>
<td>$2^{55}$</td>
<td>$2^{127}$</td>
</tr>
<tr>
<td>1372</td>
<td>1150</td>
<td>969</td>
<td>Balanced</td>
<td>$2^{257}$</td>
<td>$2^{55}$</td>
<td>$2^{257}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Min. Space</td>
<td>$2^{288}$</td>
<td>$2^{87}$</td>
<td>$2^{201}$</td>
</tr>
</tbody>
</table>

Table 5: Optimal parameters for the primal attack when the cost of SVP−b is as given in Table 3.

Cost of quantum queries The quantum cost of the list decoding sieve depends, crucially, on the use of the quantum RAM model of computation. Determining its cost in a circuit model is an open problem of considerable interest. Each Grover iteration accesses a block of memory of size roughly equal to the square of the figure given in the “Iteration Depth” column. If quantum RAM requires active error correction, then the advantage over the Table 4 cost would be lost.

Size of vectors Even assuming just $b$ bits per vector, the Table 4 cost of attacking our recommended parameter set would inflate to $2^{145}$ bits. Merely populating this memory would already be as expensive than a key search on AES-128.

Effect of MAXDEPTH The List Decoding Sieve allows for a large degree of parallelization and can be tuned to avoid reasonable MAXDEPTH bounds in a classical RAM model. To a lesser extent, this is also true when Grover search is used to perform individual near-neighbor searches. We have listed the “Iteration Depth” of these searches, i.e. the number of Grover iterations required to perform each near-neighbor query, in Table 5.

Depending on the circuit cost of one Grover iteration, it is plausible that a small MAXDEPTH could be saturated. For example, the time optimal parameterization in dimension 465 has iteration depth $2^{26}$. This would saturate a $2^{40}$ limit on MAXDEPTH if the circuit for a Grover iteration had depth $2^{14}$ quantum gates. For comparison, the circuit for one round of AES-128 given in [10] has depth $2^{13.4}$ logical quantum gates. That said, limiting MAXDEPTH to $2^{64}$ would likely have no impact on the Core-SVP security estimate for our $n = 701$ parameter set.

5.4 Hybrid attack

The hybrid attack targets a sublattice $\Lambda_{\text{hybrid}} \subseteq L_h$ of rank $d = n' + m$. A BKZ-$b$ reduced basis $V = \{v_1, \ldots, v_{d-s}\}$ for a rank $d - s$ sublattice of $\Lambda_{\text{hybrid}}$ is computed. Suppose $w$ is a short vector in $L_h$. The attacker attempts to guess $s$ coefficients of the projection of $w$ orthogonal to $V$. If the guess is correct, then it can be lifted to a short vector in $\Lambda_{\text{hybrid}}$ using Babai’s nearest plane algorithm. The success condition is with respect to the length of the last Gram-Schmidt vector in the last block of the reduced basis, $v^*_{d-s}$. Heuristically, one can expect the attack to work when

$$||v^*_{d-s}|| = 2^{s^2 - d + 2} \cdot q^m/d \geq 2 \cdot ||w||_\infty = 2.$$  

We have found that the hybrid attack is not competitive with the primal attack when both attacks use the List-Decoding Sieve with cost given by Table 3. However there are still some interesting trade-offs to consider. Recall that the costs in Table 3 depend on the use of the quantum RAM model. It is not clear whether quantum RAM is less expensive than general purpose quantum circuitry. This leads us to consider parallel hybrid attacks that use less quantum circuitry than would be required to run the List-Decoding Sieve on a quantum computer.
Table 6: Cost of the hybrid attack using the List-Decoding Sieve with cost given by Table 2. The coefficient guessing stage of the attack is done with Grover search. The classical time and space required for the list-decoding sieve is matched to the number of Grover iterations. The $MAXDEPTH$ limit only affects the coefficient guessing stage.

Table 7: Optimal parameters for the hybrid attack when SVP$-b$ is solved by enumeration of cost $2^{\text{enum}(b)}$. The formula for $\text{enum}(b)$ is Equation (2). The coefficient guessing stage of the attack is done with Grover search. The $MAXDEPTH$ limit only affects the coefficient guessing stage.

Finally in Table 8 we consider the effect of a hypothetical square-root speedup in the cost of enumeration. We assume that this speedup is due to a quantum algorithm, so it is limited by $MAXDEPTH$. Even with this massive speedup, it seems that reasonable limits on $MAXDEPTH$ force a high degree of parallelism. The entry with $MAXDEPTH = 2^{128}$ is interesting, but it seems likely that the attack in Table 4 is better.
Hybrid Attack with √Enumeration and Grover Search

<table>
<thead>
<tr>
<th>n'</th>
<th>MAXDEPTH</th>
<th>m</th>
<th>b</th>
<th>s</th>
<th>Grover Iterations</th>
<th>Iteration Depth</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>700</td>
<td>∞</td>
<td>602</td>
<td>440</td>
<td>184</td>
<td>2^{145}</td>
<td>2^{145}</td>
<td>2^0</td>
</tr>
<tr>
<td>700</td>
<td>2^{128}</td>
<td>602</td>
<td>440</td>
<td>184</td>
<td>2^{163}</td>
<td>2^{128}</td>
<td>2^{14}</td>
</tr>
<tr>
<td>700</td>
<td>2^{96}</td>
<td>602</td>
<td>440</td>
<td>184</td>
<td>2^{195}</td>
<td>2^{96}</td>
<td>2^{98}</td>
</tr>
<tr>
<td>700</td>
<td>2^{64}</td>
<td>602</td>
<td>440</td>
<td>184</td>
<td>2^{227}</td>
<td>2^{64}</td>
<td>2^{162}</td>
</tr>
<tr>
<td>940</td>
<td>∞</td>
<td>781</td>
<td>572</td>
<td>263</td>
<td>2^{207}</td>
<td>2^{207}</td>
<td>2^0</td>
</tr>
<tr>
<td>1372</td>
<td>∞</td>
<td>1060</td>
<td>866</td>
<td>453</td>
<td>2^{357}</td>
<td>2^{357}</td>
<td>2^9</td>
</tr>
</tbody>
</table>

Table 8: Optimal parameters for the hybrid attack when SVP−b is solved by enumeration of cost $2^{\text{enum}(b)/2}$. The formula for $\text{enum}(b)$ is Equation (2). The coefficient guessing stage of the attack is done with Grover search. The MAXDEPTH limit only affects all parts of the computation.

5.5 Attacks on symmetric primitives

The only symmetric primitive we use is SHAKE128. This also meets the Category 1 security level. We note that the KEM can, in principle, be used to exchange close to $n \cdot \log 3$ bits of key material. A more secure symmetric primitive can be substituted without changing any other details of the construction.

6 Advantages and limitations

We focus on comparisons with other lattice based systems.

6.1 Compared with Standard NTRU.

Some advantages and disadvantages of NTRU-HRSS compared with Standard NTRU (as defined in [16]) are as follows.

Advantages.

− **No decryption failures.** NTRU EES parameter sets have small but non-zero decryption failure probability.

− **No padding mechanisms.** By using a direct construction of a KEM, we have avoided the need for a padding mechanism like NAEP [14].

− **No fixed weight distributions.** NTRU EES parameter sets use fixed weight coefficient vectors to ensure that information about secret keys (resp. messages) is not revealed through $h(1)$ (resp. $c(1)$). This is more difficult to implement in constant time than the combination of Sample_T and $S_3 \to R$ used in NTRU-HRSS.

− **No rejection sampling.** NTRU EES uses fixed length strings of uniform random bits to sample uniform random trits and uniform values in $\{0, 1, \ldots, n-1\}$. In order for these processes to succeed with all but negligible probability, many bits must be sampled.

− **Secret keys are always invertible.** The restrictions on $n$ listed in Section 1.3.1 ensure that $f$ is always invertible modulo 2. NTRU EES parameters are chosen so that the probability of generating a non-invertible $f$ is small but not necessarily zero.
Disadvantages.

- **Large modulus.** NTRU-HRSS needs a comparatively large modulus to eliminate decryption failure. This decreases security and increases communication cost. The $n = 701$ parameter set for NTRU-HRSS would gain an estimated 20 bits of security by using modulus $q = 2048$ instead of $q = 8192$. Ciphertext length would also drop by 175 bytes.

- **Inverses mod $p$.** NTRU EES takes $f \equiv 1 \pmod{p}$ and thereby avoids multiplying by $f^{-1} \pmod{p}$ during decryption. Doing the same in NTRU-HRSS would require an even larger modulus.

6.2 Compared with Streamlined NTRUPrime.

Some advantages and disadvantages of NTRU-HRSS compared with Streamlined NTRUPrime (as defined in [5]) are as follows.

6.2.1 Advantages.

- **No fixed weight distributions.** Streamlined NTRUPrime uses relies on fixed weight distributions for its proof of correctness.

- **Private keys are always invertible.** It is possible to pick an $f$ that is not invertible modulo 3 in Streamlined NTRUPrime.

- **Power of 2 modulus.** Streamlined NTRUPrime requires a prime modulus. Some arithmetic operations are faster when $q$ is a power of 2.

6.2.2 Disadvantages.

- **Cyclotomic ring.** NTRUPrime was designed to avoid “worrisome structure” of cyclotomic rings. While algebraic structure does not figure into the cost of the best known attacks on NTRU-HRSS, it is conceivable that better algebraic attacks exist. It is also conceivable that such attacks would not apply to the rings used by NTRUPrime.

- **Probabilistic encryption.** Streamlined NTRUPrime is constructed as a deterministic public key encryption scheme.

- **No “LWR”-style rounding.** Streamlined NTRUPrime ciphertexts can be compressed.

6.3 Compared with LWE systems.

Advantages.

- **No decryption failures.** Most practical LWE schemes opt for a small decryption failure rate rather than for a narrow coefficient distribution or a large modulus.

Disadvantages.

- **Larger dimension for equal security.** In order to eliminate decryption failure, NTRU-HRSS uses trinary secret keys and messages. This results in a lower level of security than could be had with the same dimension and modulus but larger noise.

References


