NTRU

Cong Chen, Oussama Danba, Jeffrey Hoffstein,
Andreas Hülsing, Joost Rijneveld, **John M. Schanck**,
Peter Schwabe, William Whyte, Zhenfei Zhang

Second round update
2019-08-24

NTRU-HRSS-KEM (NIST Round 1)

- Perfect correctness
- Arbitrary-weight trinary vectors
- One nice parameter set
- Probabilistic encryption
- CCA2 KEM via Dent "Table 5" / Targhi–Unruh

NTRUEncrypt (NIST Round 1)

- Imperfect correctness
- Fixed-weight trinary vectors
- Many nice parameter sets
- Probabilistic encryption
- CCA2 PKE via NAEP

NTRU-HRSS-KEM (NIST Round 1)

- Perfect correctness
- Arbitrary-weight trinary vectors
- One nice parameter set
- ~~Probabilistic encryption~~
- ~~CCA2 KEM via Dent "Table 5" / Targhi–Unruh~~

↓

Saito–Xagawa–Yamakawa (Eurocrypt 2018)

- Deterministic encryption
- CCA2 KEM via re-encryption and implicit rejection

NTRUEncrypt (NIST Round 1)

- Imperfect correctness
- Fixed-weight trinary vectors
- Many nice parameter sets
- Probabilistic encryption
- CCA2 PKE via NAEP

NTRU-HRSS-KEM (NIST Round 1)

- Perfect correctness
- Arbitrary-weight trinary vectors
- One nice parameter set
- ~~Probabilistic encryption~~
- ~~CCA2 KEM via Dent "Table 5" / Targhi–Unruh~~

NTRUEncrypt (NIST Round 1)

- Imperfect correctness
- Fixed-weight trinary vectors
- Many nice parameter sets
- Probabilistic encryption
- CCA2 PKE via NAEP

↓

Saito–Xagawa–Yamakawa (Eurocrypt 2018)

- Deterministic encryption
- CCA2 KEM via re-encryption and implicit rejection

⟶ **NTRU** (NIST Round 2)

NTRU-HRSS-KEM (NIST Round 1)

- Perfect correctness
- Arbitrary-weight trinary vectors
- One nice parameter set
- ~~Probabilistic encryption~~
- ~~CCA2 KEM via Dent “Table 5” / Targhi–Unruh~~

NTRUEncrypt (NIST Round 1)

- ~~Imperfect correctness~~
- Fixed-weight trinary vectors
- Many nice parameter sets
- ~~Probabilistic encryption~~
- ~~CCA2 PKE via NAEP~~

Saito–Xagawa–Yamakawa (Eurocrypt 2018)

- Deterministic encryption
- CCA2 KEM via re-encryption and implicit rejection

→ **NTRU** (NIST Round 2)

NTRU-HRSS-KEM (NIST Round 1)

- Perfect correctness
- Arbitrary-weight trinary vectors
- One nice parameter set
- ~~Probabilistic encryption~~
- ~~CCA2 KEM via Dent "Table 5" / Targhi–Unruh~~

↓

Saito–Xagawa–Yamakawa (Eurocrypt 2018)

- Deterministic encryption
- CCA2 KEM via ~~re-encryption and~~ implicit rejection

NTRUEncrypt (NIST Round 1)

- ~~Imperfect correctness~~
- Fixed-weight trinary vectors
- Many nice parameter sets
- ~~Probabilistic encryption~~
- ~~CCA2 PKE via NAEP~~

↓

⟶ **NTRU** (NIST Round 2)

Eliminating re-encryption: rigidity

Bernstein–Persichetti (ePrint 2019/256):

ROM CCA2 KEM $\leq$ correct rigid deterministic PKE $+$ implicit rejection

Eliminating re-encryption: rigidity

Bernstein–Persichetti (ePrint 2019/256):

ROM CCA2 KEM $\leq$ correct rigid deterministic PKE + implicit rejection

Correct: $(\text{Encrypt}(m) = c) \Rightarrow (\text{Decrypt}(c) = m)$

Eliminating re-encryption: rigidity

Bernstein–Persichetti (ePrint 2019/256):

ROM CCA2 KEM $\leq$ correct rigid deterministic PKE + implicit rejection

Correct: (Encrypt$(m) = c) \Rightarrow ($Decrypt$(c) = m)$

Rigid: (Encrypt$(m) = c) \Leftarrow ($Decrypt$(c) = m)$

Eliminating re-encryption: rigidity

Bernstein–Persichetti (ePrint 2019/256):

ROM CCA2 KEM $\leq$ correct rigid deterministic PKE + implicit rejection

$$\text{Correct: } (\text{Encrypt}(m) = c) \Rightarrow (\text{Decrypt}(c) = m)$$
$$\text{Rigid: } (\text{Encrypt}(m) = c) \Leftarrow (\text{Decrypt}(c) = m)$$

Rigidity is often enforced through re-encryption...
some schemes can avoid it.

NTRU

- Integer parameters n and q.
- Polynomial arithmetic modulo $\mathbf{x}^n - 1 = \mathbf{\Phi}_1 \mathbf{\Phi}_n$.
- **Private key:** A pair of polynomials $(\mathbf{f}, \mathbf{g})$.
- **Public key:** A polynomial $\mathbf{h}$ that satisfies
  - $\mathbf{hf} \equiv 3\mathbf{g} \pmod{(q, \mathbf{\Phi}_1 \mathbf{\Phi}_n)}$, and
  - $\mathbf{h} \equiv 0 \pmod{(q, \mathbf{\Phi}_1)}$.
- **Plaintext:** A pair of polynomials $(\mathbf{r}, \mathbf{m})$, with
  - $\mathbf{m} \equiv 0 \pmod{(q, \mathbf{\Phi}_1)}$.
- **Ciphertext:** $\mathbf{c} = \mathbf{rh} + \mathbf{m} \bmod (q, \mathbf{\Phi}_1 \mathbf{\Phi}_n)$.
  - $\mathbf{c} \equiv 0 \pmod{(q, \mathbf{\Phi}_1)}$.

NTRU

- Integer parameters n and q.
- Polynomial arithmetic modulo $\mathbf{x}^n - 1 = \mathbf{\Phi}_1 \mathbf{\Phi}_n$.
- **Private key:** A pair of polynomials $(\mathbf{f}, \mathbf{g})$.
- **Public key:** A polynomial $\mathbf{h}$ that satisfies
  - $\mathbf{hf} \equiv 3\mathbf{g} \pmod{(q, \mathbf{\Phi}_1 \mathbf{\Phi}_n)}$, and
  - $\mathbf{h} \equiv 0 \pmod{(q, \mathbf{\Phi}_1)}$.
- **Plaintext:** A pair of polynomials $(\mathbf{r}, \mathbf{m})$, with
  - $\mathbf{m} \equiv 0 \pmod{(q, \mathbf{\Phi}_1)}$.
- **Ciphertext:** $\mathbf{c} = \mathbf{rh} + \mathbf{m} \bmod (q, \mathbf{\Phi}_1 \mathbf{\Phi}_n)$.
  - $\mathbf{c} \equiv 0 \pmod{(q, \mathbf{\Phi}_1)}$.

Eliminating re-encryption: rigidity

- Check: (Decrypt$(c) = m$) $\Rightarrow$ (Encrypt$(m) = c$).

Decrypt$((\mathbf{f}, \mathbf{h}), \mathbf{c})$

1: $\mathbf{a} = (\mathbf{c}\mathbf{f}) \bmod (q, \Phi_1\Phi_n)$
2: $\mathbf{m} = (\mathbf{a}/\mathbf{f}) \bmod (3, \Phi_n)$
3: $\mathbf{r} = ((\mathbf{c} - \mathbf{m})/\mathbf{h}) \bmod (q, \Phi_n)$
4: **if** $\mathbf{c} \equiv 0 \pmod{(q, \Phi_1)}$ and $(\mathbf{r}, \mathbf{m})$ is in the message space **then**
5: return $(\mathbf{r}, \mathbf{m})$
6: **end if**
7: return $\perp$

Suppose Decrypt$((\mathbf{f}, \mathbf{h}), \mathbf{c}) = (\mathbf{r}, \mathbf{m})$. Then, by Line 3,

$$\begin{aligned} \text{Encrypt}(\mathbf{h}, (\mathbf{r}, \mathbf{m})) &= \mathbf{r}\mathbf{h} + \mathbf{m} \bmod (q, \Phi_1\Phi_n) \\ &\equiv \mathbf{c} \pmod{(q, \Phi_n)} \end{aligned}$$

Lines 4-7 provide rigidity because

1. $\mathbf{h} \equiv 0 \bmod (q, \Phi_1)$, and
2. valid $\mathbf{m}$ satisfy $\mathbf{m} \equiv 0 \bmod (q, \Phi_1)$.

Eliminating re-encryption: rigidity

- Check: $(\text{Decrypt}(c) = m) \Rightarrow (\text{Encrypt}(m) = c)$.

Decrypt(($\mathbf{f}$, $\mathbf{h}$), $\mathbf{c}$)

1: $\mathbf{a} = (\mathbf{cf}) \bmod (q, \mathbf{\Phi}_1\mathbf{\Phi}_n)$
2: $\mathbf{m} = (\mathbf{a}/\mathbf{f}) \bmod (3, \mathbf{\Phi}_n)$
3: $\mathbf{r} = ((\mathbf{c} - \mathbf{m})/\mathbf{h}) \bmod (q, \mathbf{\Phi}_n)$
4: **if** $\mathbf{c} \equiv 0 \pmod{(q, \mathbf{\Phi}_1)}$ and $(\mathbf{r}, \mathbf{m})$ is in the message space **then**
5: return $(\mathbf{r}, \mathbf{m})$
6: **end if**
7: return $\perp$

Suppose Decrypt$((\mathbf{f}, \mathbf{h}), \mathbf{c}) = (\mathbf{r}, \mathbf{m})$. Then, by Line 3,

$$\begin{aligned} \text{Encrypt}(\mathbf{h}, (\mathbf{r}, \mathbf{m})) &= \mathbf{rh} + \mathbf{m} \bmod (q, \mathbf{\Phi}_1\mathbf{\Phi}_n) \\ &\equiv \mathbf{c} \pmod{(q, \mathbf{\Phi}_n)} \end{aligned}$$

Lines 4-7 provide rigidity because

1. $\mathbf{h} \equiv 0 \bmod (q, \mathbf{\Phi}_1)$, and
2. valid $\mathbf{m}$ satisfy $\mathbf{m} \equiv 0 \bmod (q, \mathbf{\Phi}_1)$.

Eliminating re-encryption: rigidity

- Check: (Decrypt(c) $= m$) $\Rightarrow$ (Encrypt(m) $= c$).

Decrypt(($\mathbf{f}$, $\mathbf{h}$), $\mathbf{c}$)

1: $\mathbf{a} = (\mathbf{c}\mathbf{f}) \bmod (q, \mathbf{\Phi}_1\mathbf{\Phi}_n)$
2: $\mathbf{m} = (\mathbf{a}/\mathbf{f}) \bmod (3, \mathbf{\Phi}_n)$
3: $\mathbf{r} = ((\mathbf{c} - \mathbf{m})/\mathbf{h}) \bmod (q, \mathbf{\Phi}_n)$
4: **if** $\mathbf{c} \equiv 0 \pmod{(q, \mathbf{\Phi}_1)}$ and $(\mathbf{r}, \mathbf{m})$ is in the message space **then**
5: return $(\mathbf{r}, \mathbf{m})$
6: **end if**
7: return $\perp$

Suppose Decrypt$((\mathbf{f}, \mathbf{h}), \mathbf{c}) = (\mathbf{r}, \mathbf{m})$. Then, by Line 3,

$$\begin{aligned} \text{Encrypt}(\mathbf{h}, (\mathbf{r}, \mathbf{m})) &= \mathbf{r}\mathbf{h} + \mathbf{m} \bmod (q, \mathbf{\Phi}_1\mathbf{\Phi}_n) \\ &\equiv \mathbf{c} \pmod{(q, \mathbf{\Phi}_n)} \end{aligned}$$

Lines 4-7 provide rigidity because

1. $\mathbf{h} \equiv 0 \bmod (q, \mathbf{\Phi}_1)$, and
2. valid $\mathbf{m}$ satisfy $\mathbf{m} \equiv 0 \bmod (q, \mathbf{\Phi}_1)$.

Eliminating re-encryption: rigidity

- Check: (Decrypt$(c) = m$) $\Rightarrow$ (Encrypt$(m) = c$).

Decrypt(($\mathbf{f}$, $\mathbf{h}$), $\mathbf{c}$)

1: $\mathbf{a} = (\mathbf{cf}) \bmod (q, \mathbf{\Phi}_1\mathbf{\Phi}_n)$
2: $\mathbf{m} = (\mathbf{a}/\mathbf{f}) \bmod (3, \mathbf{\Phi}_n)$
3: $\mathbf{r} = ((\mathbf{c} - \mathbf{m})/\mathbf{h}) \bmod (q, \mathbf{\Phi}_n)$
4: **if** $\mathbf{c} \equiv 0 \pmod{(q, \mathbf{\Phi}_1)}$ and $(\mathbf{r}, \mathbf{m})$ is in the message space **then**
5: return $(\mathbf{r}, \mathbf{m})$
6: **end if**
7: return $\perp$

Suppose Decrypt$((\mathbf{f}, \mathbf{h}), \mathbf{c}) = (\mathbf{r}, \mathbf{m})$. Then, by Line 3,

$$\begin{aligned} \text{Encrypt}(\mathbf{h}, (\mathbf{r}, \mathbf{m})) &= \mathbf{rh} + \mathbf{m} \bmod (q, \mathbf{\Phi}_1\mathbf{\Phi}_n) \\ &\equiv \mathbf{c} \pmod{(q, \mathbf{\Phi}_n)} \end{aligned}$$

Lines 4-7 provide rigidity because

1. $\mathbf{h} \equiv 0 \bmod (q, \mathbf{\Phi}_1)$, and
2. valid $\mathbf{m}$ satisfy $\mathbf{m} \equiv 0 \bmod (q, \mathbf{\Phi}_1)$.

Eliminating re-encryption: rigidity

- Check: (Decrypt(c) $= m$) $\Rightarrow$ (Encrypt(m) $= c$).

Decrypt(($\mathbf{f}$, $\mathbf{h}$), $\mathbf{c}$)

1: $\mathbf{a} = (\mathbf{cf}) \bmod (q, \mathbf{\Phi}_1\mathbf{\Phi}_n)$
2: $\mathbf{m} = (\mathbf{a}/\mathbf{f}) \bmod (3, \mathbf{\Phi}_n)$
3: $\mathbf{r} = ((\mathbf{c} - \mathbf{m})/\mathbf{h}) \bmod (q, \mathbf{\Phi}_n)$
4: **if** $\mathbf{c} \equiv 0 \pmod{(q, \mathbf{\Phi}_1)}$ and $(\mathbf{r}, \mathbf{m})$ is in the message space **then**
5: return $(\mathbf{r}, \mathbf{m})$
6: **end if**
7: return $\perp$

Suppose Decrypt$((\mathbf{f}, \mathbf{h}), \mathbf{c}) = (\mathbf{r}, \mathbf{m})$. Then, by Line 3,

$$\begin{aligned} \text{Encrypt}(\mathbf{h}, (\mathbf{r}, \mathbf{m})) &= \mathbf{rh} + \mathbf{m} \bmod (q, \mathbf{\Phi}_1\mathbf{\Phi}_n) \\ &\equiv \mathbf{c} \pmod{(q, \mathbf{\Phi}_n)} \end{aligned}$$

Lines 4-7 provide rigidity because

1. $\mathbf{h} \equiv 0 \bmod (q, \mathbf{\Phi}_1)$, and
2. valid $\mathbf{m}$ satisfy $\mathbf{m} \equiv 0 \bmod (q, \mathbf{\Phi}_1)$.

Eliminating re-encryption: implicit rejection

- The user stores an additional 256 bit secret, s.

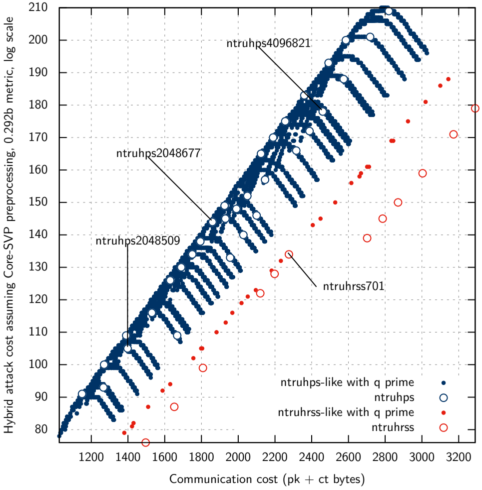
Encaps($\mathbf{h}$):

Sample $\mathbf{r}$ and $\mathbf{m}$.
return $\mathbf{rh} + \mathbf{m} \bmod (q, \mathbf{\Phi}_1\mathbf{\Phi}_n)$.

Decaps($(\mathbf{f}, \mathbf{h}, s), \mathbf{c}$):

$result =$ Decrypt($(\mathbf{f}, \mathbf{h}), \mathbf{c}$)
if $result = \perp$ **then**
    return $\mathrm{SHA3\text{-}256}(s \mid \mathbf{c})$
else
    return $\mathrm{SHA3\text{-}256}(result)$
end if

Parameter selection process



4pt 0pt

Recommended parameters

	pk bytes	ct bytes	Core-SVP dim.
ntruhps2048509	699	699	364
ntruhps2048677	930	930	496
ntruhrss701	1138	1138	470
ntruhps4096821	1230	1230	612

Recommended parameters

	pk bytes	ct bytes	Core-SVP dim.
ntruhps2048509	699	699	364
ntruhps2048677	930	930	496
ntruhrss701	1138	1138	470
ntruhps4096821	1230	1230	612

	Key Gen	Encaps	Decaps
ntruhps2048509	171**k**	38**k**	49**k**
ntruhps2048677	292**k**	53**k**	73**k**
ntruhrss701	283**k**	52**k**	76**k**
ntruhps4096821	-	-	-

$\mathbf{k} = 1000$ Haswell cycles.

one second $= 3\,100\,000\mathbf{k}$

Recommended parameters

	pk bytes	ct bytes	Core-SVP dim.
ntruhps2048509	699	699	364
ntruhps2048677	930	930	496
ntruhrss701	1138	1138	470
ntruhps4096821	1230	1230	612

	Key Gen	Encaps	Decaps
ntruhps2048509			
ntruhps2048677			
ntruhrss701			
ntruhps4096821	-	-	-

$\mathbf{k} = 1000$ Haswell cycles.

one second $= 3\,100\,000\mathbf{k}$

Recommended parameters

	pk bytes	ct bytes	Core-SVP dim.
ntruhps2048509	699	699	364
ntruhps2048677	930	930	496
ntruhrss701	1138	1138	470
ntruhps4096821	1230	1230	612

	Key Gen	Encaps	Decaps
ntruhps2048509	167**k**	25**k**	49**k**
ntruhps2048677	277**k**	35**k**	69**k**
ntruhrss701	255**k**	27**k**	71**k**
ntruhps4096821	-	-	-

$\mathbf{k} = 1000$ Haswell cycles.

one second $= 3\,100\,000\mathbf{k}$

Faster key generation?

Faster key generation?

Optimize! Most expensive component is inversion mod $(3, \mathbf{\Phi}_n)$:

- Original ntruhrss701 software:
  150**k** Haswell cycles
- New software from Dan Bernstein and Bo-Yin Yang, ePrint 2019/266:
  90**k** Haswell cycles

Faster key generation?

Optimize! Most expensive component is inversion mod $(3, \mathbf{\Phi}_n)$:

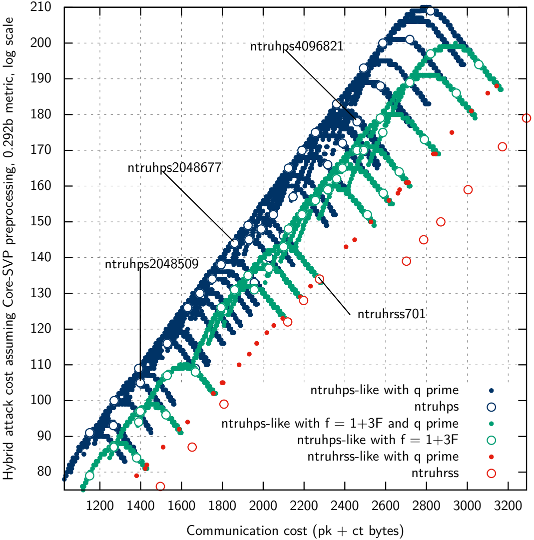
- Original ntruhrss701 software:
  150**k** Haswell cycles
- New software from Dan Bernstein and Bo-Yin Yang, ePrint 2019/266:
  90**k** Haswell cycles

Other avenues to explore:

- Use $\mathbf{f} = 1 + 3\mathbf{F}$ in an ephemeral-only setting.
- Choose perfectly correct parameters compatible with $\mathbf{f} = 1 + 3\mathbf{F}$.

Neither option is currently recommended.

Correct parameters with faster key gen



4pt 0pt