Radboud University

Lattice-based cryptography – Episode IV

A new hope

Peter Schwabe
Joint work with Erdem Alkim, Léo Ducas, and Thomas Pöppelmann
peter@cryptojedi.org
https://cryptojedi.org

June 23, 2017

Google Security Blog

The latest news and insights from Google on security and safety on the Internet

Experimenting with Post-Quantum Cryptography

July 7, 2016

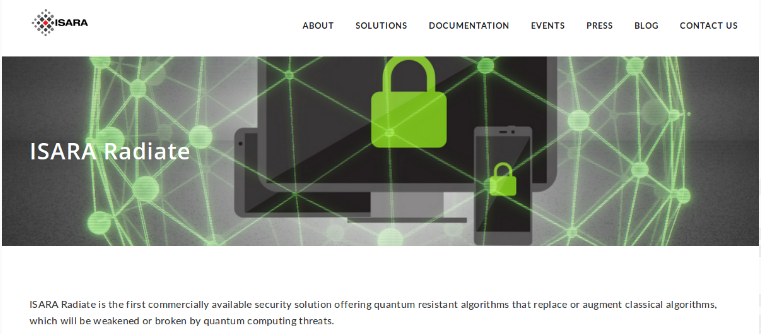
Posted by Matt Braithwaite, Software Engineer

Search blog ...

Archive

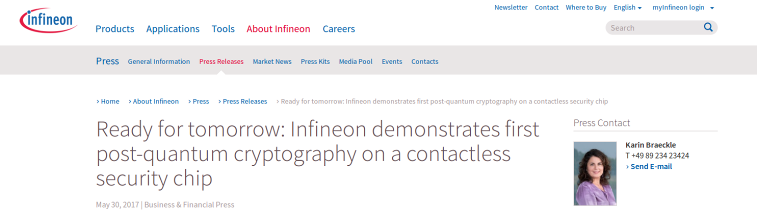
"We're indebted to Erdem Alkim, Léo Ducas, Thomas Pöppelmann and Peter Schwabe, the researchers who developed "New Hope", the post-quantum algorithm that we selected for this experiment."

https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html



"Key Agreement using the 'NewHope' lattice-based algorithm detailed in the New Hope paper, and LUKE (Lattice-based Unique Key Exchange), an ISARA speed-optimized version of the NewHope algorithm."

https://www.isara.com/isara-radiate/



Newsletter Contact Where to Buy English myInfineon login

infineon

Products Applications Tools About Infineon Careers

Search

Press General Information Press Releases Market News Press Kits Media Pool Events Contacts

› Home › About Infineon › Press › Press Releases › Ready for tomorrow: Infineon demonstrates first post-quantum cryptography on a contactless security chip

Ready for tomorrow: Infineon demonstrates first post-quantum cryptography on a contactless security chip

May 30, 2017 | Business & Financial Press

Press Contact

Karin Braeckle
T +49 89 234 23424
› Send E-mail

"The deployed algorithm is a variant of "New Hope", a quantum-resistant cryptosystem"

https://www.infineon.com/cms/en/about-infineon/press/press-releases/2017/INFCCS201705-056.html

A bit of (R)LWE history

- Hoffstein, Pipher, Silverman, 1996: NTRU cryptosystem
- Regev, 2005: Introduce LWE-based encryption
- Lyubashevsky, Peikert, Regev, 2010: Ring-LWE and Ring-LWE encryption
- Ding, Xie, Lin, 2012: Transform to (R)LWE-based key exchange
- Peikert, 2014: Improved RLWE-based key exchange
- Bos, Costello, Naehrig, Stebila, 2015: Instantiate and implement Peikert's key exchange in TLS:
- Alkim, Ducas, Pöppelmann, Schwabe, Aug. 2016: NewHope
- Alkim, Ducas, Pöppelmann, Schwabe, Dec. 2016: NewHope-Simple

Ring-Learning-with-errors (RLWE)

- Let $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$
- Let χ be an *error distribution* on $\mathcal{R}_q$
- Let $\mathbf{s} \in \mathcal{R}_q$ be secret
- Attacker is given pairs $(\mathbf{a}, \mathbf{as} + \mathbf{e})$ with
    - $\mathbf{a}$ uniformly random from $\mathcal{R}_q$
    - $\mathbf{e}$ sampled from χ
- Task for the attacker: find $\mathbf{s}$

Ring-Learning-with-errors (RLWE)

- Let $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$
- Let χ be an *error distribution* on $\mathcal{R}_q$
- Let $\mathbf{s} \in \mathcal{R}_q$ be secret
- Attacker is given pairs $(\mathbf{a}, \mathbf{as} + \mathbf{e})$ with
  - $\mathbf{a}$ uniformly random from $\mathcal{R}_q$
  - $\mathbf{e}$ sampled from χ
- Task for the attacker: find $\mathbf{s}$
- Common choice for χ: discrete Gaussian
- Common optimization for protocols: fix $\mathbf{a}$

RLWE-based Encryption, KEM, KEX

Alice (server)		Bob (client)
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \chi$		$\mathbf{s}', \mathbf{e}' \xleftarrow{\$} \chi$
$\mathbf{b} \leftarrow \mathbf{a}\mathbf{s} + \mathbf{e}$	$\xrightarrow{\quad \mathbf{b} \quad}$	$\mathbf{u} \leftarrow \mathbf{a}\mathbf{s}' + \mathbf{e}'$
	$\xleftarrow{\quad \mathbf{u} \quad}$	

$$
\begin{aligned}
&\text{Alice has} \quad \mathbf{v} &&= \mathbf{u}\mathbf{s} &&= \mathbf{a}\mathbf{s}\mathbf{s}' + \mathbf{e}'\mathbf{s} \\
&\text{Bob has} \quad \mathbf{v}' &&= \mathbf{b}\mathbf{s}' &&= \mathbf{a}\mathbf{s}\mathbf{s}' + \mathbf{e}\mathbf{s}'
\end{aligned}
$$

- Secret and noise polynomials $\mathbf{s}, \mathbf{s}', \mathbf{e}, \mathbf{e}'$ are small
- $\mathbf{v}$ and $\mathbf{v}'$ are *approximately* the same

NewHope-Simple key exchange (simplified)

Alice		Bob
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \chi$		$\mathbf{s}', \mathbf{e}' \xleftarrow{\$} \chi$
$\mathbf{b} \leftarrow \mathbf{a}\mathbf{s} + \mathbf{e}$	$\xrightarrow{(\mathbf{b}\quad)}$	
		$\mathbf{u} \leftarrow \mathbf{a}\mathbf{s}' + \mathbf{e}'$
		$\mathbf{v} \leftarrow \mathbf{b}\mathbf{s}'$
$\mathbf{v}' \leftarrow \mathbf{u}\mathbf{s}$	$\xleftarrow{(\mathbf{u}\quad)}$	

Alice		Bob
$\mathit{seed} \xleftarrow{\$} \{0,1\}^{256}$		
$\mathbf{a} \leftarrow \mathrm{Parse}(\mathrm{SHAKE\text{-}128}(\mathit{seed}))$		
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \chi$		$\mathbf{s}', \mathbf{e}' \xleftarrow{\$} \chi$
$\mathbf{b} \leftarrow \mathbf{as} + \mathbf{e}$	$\xrightarrow{(\mathbf{b}, \mathit{seed})}$	$\mathbf{a} \leftarrow \mathrm{Parse}(\mathrm{SHAKE\text{-}128}(\mathit{seed}))$
		$\mathbf{u} \leftarrow \mathbf{as}' + \mathbf{e}'$
		$\mathbf{v} \leftarrow \mathbf{bs}'$
$\mathbf{v}' \leftarrow \mathbf{us}$	$\xleftarrow{(\mathbf{u} \quad)}$	

NewHope-Simple key exchange (simplified)

Alice		Bob
$seed \xleftarrow{\$} \{0,1\}^{256}$		
$\mathbf{a} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$		
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \chi$		$\mathbf{s}', \mathbf{e}' \xleftarrow{\$} \chi$
$\mathbf{b} \leftarrow \mathbf{as} + \mathbf{e}$	$\xrightarrow{(\mathbf{b}, seed)}$	$\mathbf{a} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$
		$\mathbf{u} \leftarrow \mathbf{as}' + \mathbf{e}'$
		$\mathbf{v} \leftarrow \mathbf{bs}'$
		$k \xleftarrow{\$} \{0,1\}^n$
		$\mathbf{k} \leftarrow \text{Encode}(k)$
$\mathbf{v}' \leftarrow \mathbf{us}$	$\xleftarrow{(\mathbf{u}, \mathbf{c})}$	$\mathbf{c} \leftarrow \mathbf{v} + \mathbf{k}$

NewHope-Simple key exchange (simplified)

Alice		Bob
$seed \xleftarrow{\$} \{0,1\}^{256}$		
$\mathbf{a} \leftarrow \mathrm{Parse}(\text{SHAKE-128}(seed))$		
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \chi$		$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \chi$
$\mathbf{b} \leftarrow \mathbf{as} + \mathbf{e}$	$\xrightarrow{(\mathbf{b}, seed)}$	$\mathbf{a} \leftarrow \mathrm{Parse}(\text{SHAKE-128}(seed))$
		$\mathbf{u} \leftarrow \mathbf{as}' + \mathbf{e}'$
		$\mathbf{v} \leftarrow \mathbf{bs}' + \mathbf{e}''$
		$k \xleftarrow{\$} \{0,1\}^n$
		$\mathbf{k} \leftarrow \mathrm{Encode}(k)$
$\mathbf{v}' \leftarrow \mathbf{us}$	$\xleftarrow{(\mathbf{u}, \mathbf{c})}$	$\mathbf{c} \leftarrow \mathbf{v} + \mathbf{k}$

NewHope-Simple key exchange (simplified)

Alice		Bob
$seed \xleftarrow{\$} \{0,1\}^{256}$		
$\mathbf{a} \leftarrow \mathrm{Parse}(\mathrm{SHAKE\text{-}128}(seed))$		
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \chi$		$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \chi$
$\mathbf{b} \leftarrow \mathbf{as} + \mathbf{e}$	$\xrightarrow{(\mathbf{b}, seed)}$	$\mathbf{a} \leftarrow \mathrm{Parse}(\mathrm{SHAKE\text{-}128}(seed))$
		$\mathbf{u} \leftarrow \mathbf{as}' + \mathbf{e}'$
		$\mathbf{v} \leftarrow \mathbf{bs}' + \mathbf{e}''$
		$k \xleftarrow{\$} \{0,1\}^n$
		$\mathbf{k} \leftarrow \mathrm{Encode}(k)$
$\mathbf{v}' \leftarrow \mathbf{us}$	$\xleftarrow{(\mathbf{u}, \mathbf{c})}$	$\mathbf{c} \leftarrow \mathbf{v} + \mathbf{k}$
$\mathbf{k}' \leftarrow \mathbf{c} - \mathbf{v}'$		

NewHope-Simple key exchange (simplified)

Alice		Bob
$seed \xleftarrow{\$} \{0,1\}^{256}$		
$\mathbf{a} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$		
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \chi$		$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \chi$
$\mathbf{b} \leftarrow \mathbf{as} + \mathbf{e}$	$\xrightarrow{(\mathbf{b}, seed)}$	$\mathbf{a} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$
		$\mathbf{u} \leftarrow \mathbf{as}' + \mathbf{e}'$
		$\mathbf{v} \leftarrow \mathbf{bs}' + \mathbf{e}''$
		$k \xleftarrow{\$} \{0,1\}^n$
		$\mathbf{k} \leftarrow \text{Encode}(k)$
$\mathbf{v}' \leftarrow \mathbf{us}$	$\xleftarrow{(\mathbf{u}, \mathbf{c})}$	$\mathbf{c} \leftarrow \mathbf{v} + \mathbf{k}$
$\mathbf{k}' \leftarrow \mathbf{c} - \mathbf{v}'$		$\mu \leftarrow \text{Extract}(\mathbf{k})$
$\mu \leftarrow \text{Extract}(\mathbf{k}')$		

NewHope-Simple key exchange (simplified)

Alice		Bob
$seed \xleftarrow{\$} \{0,1\}^{256}$		
$\mathbf{a} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$		
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \chi$		$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \chi$
$\mathbf{b} \leftarrow \mathbf{as} + \mathbf{e}$	$\xrightarrow{(\mathbf{b}, seed)}$	$\mathbf{a} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$
		$\mathbf{u} \leftarrow \mathbf{as}' + \mathbf{e}'$
		$\mathbf{v} \leftarrow \mathbf{bs}' + \mathbf{e}''$
		$k \xleftarrow{\$} \{0,1\}^n$
		$\mathbf{k} \leftarrow \text{Encode}(k)$
$\mathbf{v}' \leftarrow \mathbf{us}$	$\xleftarrow{(\mathbf{u}, \mathbf{c})}$	$\mathbf{c} \leftarrow \mathbf{v} + \mathbf{k}$
$\mathbf{k}' \leftarrow \mathbf{c} - \mathbf{v}'$		$\mu \leftarrow \text{Extract}(\mathbf{k})$
$\mu \leftarrow \text{Extract}(\mathbf{k}')$		

This is LPR encryption, written as KEX (except for generation of **a**)

Against all authority

- Standard approach to choosing **a**:

  "Let **a** *be a uniformly random..."*

Against all authority

- Standard approach to choosing **a**:

  "Let **a** *be a uniformly random. . . "*

- Standard *real-world* approach: generate fixed **a** once

Against all authority

- Standard approach to choosing $\mathbf{a}$:

  "Let $\mathbf{a}$ *be a uniformly random..."*

- Standard *real-world* approach: generate fixed $\mathbf{a}$ once
- What if $\mathbf{a}$ is backdoored?
- Parameter-generating authority can break key exchange
- "Solution": Nothing-up-my-sleeves (involves endless discussion!)

Against all authority

- Standard approach to choosing **a**:

  *"Let **a** be a uniformly random…"*

- Standard *real-world* approach: generate fixed **a** once
- What if **a** is backdoored?
- Parameter-generating authority can break key exchange
- "Solution": Nothing-up-my-sleeves (involves endless discussion!)
- Even without backdoor:
  - Perform massive precomputation based on **a**
  - Use precomputation to break *all* key exchanges
  - Infeasible today, but who knows…
  - Attack in the spirit of Logjam

Against all authority

- Standard approach to choosing **a**:

  *"Let **a** be a uniformly random..."*

- Standard *real-world* approach: generate fixed **a** once
- What if **a** is backdoored?
- Parameter-generating authority can break key exchange
- "Solution": Nothing-up-my-sleeves (involves endless discussion!)
- Even without backdoor:
  - Perform massive precomputation based on **a**
  - Use precomputation to break *all* key exchanges
  - Infeasible today, but who knows...
  - Attack in the spirit of Logjam
- Solution in NewHope(-Simple): Choose a fresh **a** every time
- Server can cache **a** for some time (e.g., 1h)

Against all authority

- Standard approach to choosing **a**:

  *"Let **a** be a uniformly random…"*

- Standard *real-world* approach: generate fixed **a** once
- What if **a** is backdoored?
- Parameter-generating authority can break key exchange
- "Solution": Nothing-up-my-sleeves (involves endless discussion!)
- Even without backdoor:
  - Perform massive precomputation based on **a**
  - Use precomputation to break *all* key exchanges
  - Infeasible today, but who knows…
  - Attack in the spirit of Logjam
- Solution in NewHope(-Simple): Choose a fresh **a** every time
- Server can cache **a** for some time (e.g., 1h)
- **Must not reuse keys/noise!**

Isn't SHAKE slow?

- SHAKE-128 is slower than, e.g., AES-NI, Salsa20/ChaCha20, Blake2X,... in software
- First versions of NewHope used Chacha20 to generate **a**
- Gueron, Schlieker, 2016: NewHope becomes faster if we use AES-NI instead of SHAKE-128
- Google actually used Gueron-Schlieker version

Isn't SHAKE slow?

- SHAKE-128 is slower than, e.g., AES-NI, Salsa20/ChaCha20, Blake2X,... in software
- First versions of NewHope used Chacha20 to generate **a**
- Gueron, Schlieker, 2016: NewHope becomes faster if we use AES-NI instead of SHAKE-128
- Google actually used Gueron-Schlieker version
- Problem in modelling:
  - PRG is not the right building block
  - PRG is secure only for *secret* input
  - Could "zoom into" ChaCha20 or AES and argue security

Isn't SHAKE slow?

- SHAKE-128 is slower than, e.g., AES-NI, Salsa20/ChaCha20, Blake2X,... in software
- First versions of NewHope used Chacha20 to generate **a**
- Gueron, Schlieker, 2016: NewHope becomes faster if we use AES-NI instead of SHAKE-128
- Google actually used Gueron-Schlieker version
- Problem in modelling:
  - PRG is not the right building block
  - PRG is secure only for *secret* input
  - Could "zoom into" ChaCha20 or AES and argue security
- Problem in practice:
  - AES is nasty in software, real advantage only with hardware AES
  - ChaCha20 is in TLS, but not that thoroughly analyzed
  - Blake2X: Also not much cryptanalysis
  - Salsa20: Better analysis, no "NIST approval"

Encode and Extract

- Encoding in LPR encryption: map n bits to n coefficients:
  - A zero bit maps to 0
  - A one bit maps to $q/2$
- Idea: Noise affects low bits of coefficients, put data into high bits

Encode and Extract

- Encoding in LPR encryption: map n bits to n coefficients:
    - A zero bit maps to 0
    - A one bit maps to $q/2$
- Idea: Noise affects low bits of coefficients, put data into high bits
- Decode: map coefficient into $[-q/2, q/2]$
    - Closer to 0 (i.e., in $[-1/4q, 1/4q]$): set bit to zero
    - Closer to $\pm q/2$: set bit to one

Encode and Extract

- Encoding in LPR encryption: map n bits to n coefficients:
  - A zero bit maps to 0
  - A one bit maps to $q/2$
- Idea: Noise affects low bits of coefficients, put data into high bits
- Decode: map coefficient into $[-q/2, q/2]$
  - Closer to 0 (i.e., in $[-1/4q, 1/4q]$): set bit to zero
  - Closer to $\pm q/2$: set bit to one
- NewHope-Simple: map $n/4$ bits to n coefficients
- Set 4 coefficients to 0 or to $q/2$
- Decode: map coeffs into $[-q/2, q/2]$, sum up 4 absolute values
  - Closer to 0 (i.e., in $[0, q]$): set bit to zero
  - Closer to $\pm 2q$: set bit to one

Encode and Extract

- Encoding in LPR encryption: map n bits to n coefficients:
  - A zero bit maps to 0
  - A one bit maps to $q/2$
- Idea: Noise affects low bits of coefficients, put data into high bits
- Decode: map coefficient into $[-q/2, q/2]$
  - Closer to 0 (i.e., in $[-1/4q, 1/4q]$): set bit to zero
  - Closer to $\pm q/2$: set bit to one
- NewHope-Simple: map $n/4$ bits to n coefficients
- Set 4 coefficients to 0 or to $q/2$
- Decode: map coeffs into $[-q/2, q/2]$, sum up 4 absolute values
  - Closer to 0 (i.e., in $[0, q]$): set bit to zero
  - Closer to $\pm 2q$: set bit to one
- First proposed by Pöppelmann and Güneysu in 2013.

Reducing the size of c

- Remember that Bob sends $\mathbf{c} = \mathbf{v} + \mathbf{k}$
- Alice recovers $\mathbf{k}' = \mathbf{c} - \mathbf{v}' \approx \mathbf{k}$
- Information about $\mathbf{k}$ sits in high bits of $\mathbf{c}$ coefficients
- Noise sits in low bits of $\mathbf{c}$ coefficients

Reducing the size of c

- Remember that Bob sends $\mathbf{c} = \mathbf{v} + \mathbf{k}$
- Alice recovers $\mathbf{k}' = \mathbf{c} - \mathbf{v}' \approx \mathbf{k}$
- Information about $\mathbf{k}$ sits in high bits of $\mathbf{c}$ coefficients
- Noise sits in low bits of $\mathbf{c}$ coefficients
- Idea: **Don't transmit low bits**
- This introduces additional (deterministic, uniform) noise
- In NewHope-Simple compress coefficients to 3 bits:

$$\bar{c}_i \leftarrow \lceil (c_i \cdot 8)/q \rfloor \mod 8$$

Reducing the size of c

- Remember that Bob sends $\mathbf{c} = \mathbf{v} + \mathbf{k}$
- Alice recovers $\mathbf{k}' = \mathbf{c} - \mathbf{v}' \approx \mathbf{k}$
- Information about $\mathbf{k}$ sits in high bits of $\mathbf{c}$ coefficients
- Noise sits in low bits of $\mathbf{c}$ coefficients
- Idea: **Don't transmit low bits**
- This introduces additional (deterministic, uniform) noise
- In NewHope-Simple compress coefficients to 3 bits:

$$\bar{c}_i \leftarrow \lceil (c_i \cdot 8)/q \rfloor \mod 8$$

- Recover $c_i' \approx c_i$ on the receiver side:

$$c_i' \leftarrow \lceil (\bar{c}_i \cdot q)/8 \rfloor$$

Reducing the size of c

- Remember that Bob sends $\mathbf{c} = \mathbf{v} + \mathbf{k}$
- Alice recovers $\mathbf{k}' = \mathbf{c} - \mathbf{v}' \approx \mathbf{k}$
- Information about $\mathbf{k}$ sits in high bits of $\mathbf{c}$ coefficients
- Noise sits in low bits of $\mathbf{c}$ coefficients
- Idea: **Don't transmit low bits**
- This introduces additional (deterministic, uniform) noise
- In NewHope-Simple compress coefficients to 3 bits:

$$\bar{c}_i \leftarrow \lceil (c_i \cdot 8)/q \rfloor \mod 8$$

- Recover $c_i' \approx c_i$ on the receiver side:

$$c_i' \leftarrow \lceil (\bar{c}_i \cdot q)/8 \rfloor$$

- Technique known at least since 2009 (Peikert), used in various other protocols

Parameter choices

BCNS key exchange

- Starting point: Bos, Costello, Naehrig, Stebila 2015:
- $n = 1024$, $q = 2^{32} - 1$
- Error distribution: discrete Gaussian
- Claim: 128-bit pre-quantum security

Parameter choices

BCNS key exchange

- Starting point: Bos, Costello, Naehrig, Stebila 2015:
- $n = 1024$, $q = 2^{32} - 1$
- Error distribution: discrete Gaussian
- Claim: 128-bit pre-quantum security

NewHope(-Simple) key exchange

- $n = 1024$, $q = 12289$ (14 bits)
- Error distribution: centered binomial:
  - Sample uniformly random k-bit integers a and b
  - Output $HW(a) - HW(b)$ (HW = Hamming weight)
  - In NewHope we use $k = 16$
- Claim: $\gg$ 128-bit post-quantum security

Post-quantum security

- Consider RLWE instance as LWE instance
- Attack using BKZ
- BKZ uses SVP oracle in smaller dimension
- Consider only the cost of one call to that oracle ("core-SVP hardness")

Post-quantum security

- Consider RLWE instance as LWE instance
- Attack using BKZ
- BKZ uses SVP oracle in smaller dimension
- Consider only the cost of one call to that oracle ("core-SVP hardness")
- Consider quantum sieve as SVP oracle
  - Best-known quantum cost (BKC): $2^{0.265n}$
  - Best-plausible quantum cost (BPC): $2^{0.2075n}$

Post-quantum security

- Consider RLWE instance as LWE instance
- Attack using BKZ
- BKZ uses SVP oracle in smaller dimension
- Consider only the cost of one call to that oracle ("core-SVP hardness")
- Consider quantum sieve as SVP oracle
  - Best-known quantum cost (BKC): $2^{0.265n}$
  - Best-plausible quantum cost (BPC): $2^{0.2075n}$
- Obtain lower bounds on the bit security:

	Known Classical	Known Quantum	Best Plausible
BCNS	86	78	61
NewHope	281	255	199

Polynomial multiplication

- Most costly arithmetic: multiply in $\mathcal{R}_q$
- Choose q s.t. $2n \mid (q-1)$
- Use fast negacyclic **number-theoretic transform (NTT)**
- Compute $\mathbf{r} = \mathbf{ab}$ as $\mathbf{r} = \mathsf{NTT}^{-1}(\mathsf{NTT}(\mathbf{a}) \circ \mathsf{NTT}(\mathbf{b}))$

Polynomial multiplication

- Most costly arithmetic: multiply in $\mathcal{R}_q$
- Choose q s.t. $2n \mid (q-1)$
- Use fast negacyclic **number-theoretic transform (NTT)**
- Compute $\mathbf{r} = \mathbf{ab}$ as $\mathbf{r} = \mathrm{NTT}^{-1}(\mathrm{NTT}(\mathbf{a}) \circ \mathrm{NTT}(\mathbf{b}))$
- NTT computation: $\frac{n}{2} \cdot \log(n)$ “butterfly operations”
- Each butterfly: 1 addition, 1 subtraction, 1 multiplication by constant

Polynomial multiplication

- Most costly arithmetic: multiply in $\mathcal{R}_q$
- Choose q s.t. $2n \mid (q-1)$
- Use fast negacyclic **number-theoretic transform (NTT)**
- Compute $\mathbf{r} = \mathbf{ab}$ as $\mathbf{r} = \mathrm{NTT}^{-1}(\mathrm{NTT}(\mathbf{a}) \circ \mathrm{NTT}(\mathbf{b}))$
- NTT computation: $\frac{n}{2} \cdot \log(n)$ “butterfly operations”
- Each butterfly: 1 addition, 1 subtraction, 1 multiplication by constant
- NTT transforms uniform randomness to uniform randomness
- Idea: Assume that $\mathbf{a}$ is directly sampled in NTT domain

Polynomial multiplication

- Most costly arithmetic: multiply in $\mathcal{R}_q$
- Choose q s.t. $2n \mid (q-1)$
- Use fast negacyclic **number-theoretic transform (NTT)**
- Compute $\mathbf{r} = \mathbf{ab}$ as $\mathbf{r} = \mathrm{NTT}^{-1}(\mathrm{NTT}(\mathbf{a}) \circ \mathrm{NTT}(\mathbf{b}))$
- NTT computation: $\frac{n}{2} \cdot \log(n)$ "butterfly operations"
- Each butterfly: 1 addition, 1 subtraction, 1 multiplication by constant
- NTT transforms uniform randomness to uniform randomness
- Idea: Assume that $\mathbf{a}$ is directly sampled in NTT domain
- Further optimization: send messages in NTT domain
- Save two NTT computations

Putting it all together

Alice (keygen):

$seed \xleftarrow{\$} \{0, \ldots, 255\}^{32}$

$\hat{\mathbf{a}} \leftarrow \mathrm{Parse}(\text{SHAKE-128}(seed))$

$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \psi_{16}^{n}$

$\hat{\mathbf{s}} \leftarrow \mathrm{NTT}(\mathbf{s})$

$\hat{\mathbf{b}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{s}} + \mathrm{NTT}(\mathbf{e})$

Send $m_a = \mathrm{encodeA}(seed, \hat{\mathbf{b}})$ (1824 Bytes)

Bob (keygen+sharedkey):

$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \psi_{16}^{n}$
$(\hat{\mathbf{b}}, seed) \leftarrow \text{decodeA}(m_a)$
$\hat{\mathbf{a}} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$
$\hat{\mathbf{t}} \leftarrow \text{NTT}(\mathbf{s}')$
$\hat{\mathbf{u}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{t}} + \text{NTT}(\mathbf{e}')$
$k \xleftarrow{\$} \{0,1\}^{256}$
$k' \leftarrow \text{SHA3-256}(k)$
$\mathbf{k} \leftarrow \text{NHSEncode}(k')$
$\mathbf{c} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{b}} \circ \hat{\mathbf{t}}) + \mathbf{e}'' + \mathbf{k}$
$\bar{\mathbf{c}} \xleftarrow{\$} \text{NHSCompress}(\mathbf{c})$
Send $m_b = \text{encodeB}(\hat{\mathbf{u}}, \bar{\mathbf{c}})$ (2176 Bytes)
$\mu \leftarrow \text{SHA3-256}(k')$

Alice (sharedkey):

$(\hat{\mathbf{u}}, \bar{\mathbf{c}}) \leftarrow \text{decodeB}(m_b)$
$\mathbf{c}' \leftarrow \text{NHSDecompress}(\bar{\mathbf{c}})$
$\mathbf{k}' = \mathbf{c}' - \text{NTT}^{-1}(\hat{\mathbf{u}} \circ \hat{\mathbf{s}})$
$k' \leftarrow \text{NHSDecode}(\mathbf{k}')$
$\mu \leftarrow \text{SHA3-256}(k')$

Performance

	BCNS	C ref	AVX2
Key generation (server)	$\approx$ 2 477 958	258 246	88 920
Key gen + shared key (client)	$\approx$ 3 995 977	384 994	110 986
Shared key (server)	$\approx$ 481 937	86 280	19 422

- Cycle counts for NewHope on one core of an Intel i7-4770K (Haswell)
- BCNS benchmarks are derived from `openssl speed`
- Includes around $\approx$ 37 000 cycles for generation of **a** on each side
- Compare to X25519 elliptic-curve scalar mult: 156 092 cycles

Google's conclusions

"[...] we did not find any unexpected impediment to deploying something like NewHope. There were no reported problems caused by enabling it."

Google's conclusions

"[. . .] if the need arose, it would be practical to quickly deploy NewHope in TLS 1.2. (TLS 1.3 makes things a little more complex and we did not test with CECPQ1 with it.)"

Google's conclusions

"Although the median connection latency only increased by a millisecond, the latency for the slowest 5% increased by 20ms and, for the slowest 1%, by 150ms. Since NewHope is computationally inexpensive, we're assuming that this is caused entirely by the increased message sizes. Since connection latencies compound on the web (because subresource discovery is delayed), the data requirement of NewHope is moderately expensive for people on slower connections."

NewHope(-Simple) online

NewHope Paper:	https://cryptojedi.org/papers/#newhope
NHS Paper:	https://cryptojedi.org/papers/#newhopesimple
Software:	https://cryptojedi.org/crypto/#newhope
Newhope for ARM:	https://github.com/newhopearm/newhopearm.git
	(by Erdem Alkim, Philipp Jakubeit, and Peter Schwabe)

NewHope(-Simple) online

NewHope Paper:	https://cryptojedi.org/papers/#newhope
NHS Paper:	https://cryptojedi.org/papers/#newhopesimple
Software:	https://cryptojedi.org/crypto/#newhope
Newhope for ARM:	https://github.com/newhopearm/newhopearm.git
	(by Erdem Alkim, Philipp Jakubeit, and Peter Schwabe)

More Software:

https://ianix.com/pqcrypto/pqcrypto-deployment.html