POST-QUANTUM KEY EXCHANGE

A NEW HOPE

ERDEM ALKIM

LÉO DUCAS

THOMAS PÖPPELMANN

PETER SCHWABE

Quantum computers – should we be worried?

"In the past, people have said, maybe it's 50 years away, it's a dream, maybe it'll happen sometime. I used to think it was 50. Now I'm thinking like it's 15 or a little more. It's within reach. It's within our lifetime. It's going to happen."

—Mark Ketchen (IBM), Feb. 2012, about quantum computers

Quantum computers – should we be worried?

“Now, we’re aiming to build the first quantum integrated circuit, which we’re aiming for by 2020.

Beyond that, we must do error correction, so that if errors come into the chip, you can run multiple processes in parallel to eliminate those errors and that error correction will take another five years or so.”

—Michelle Simmons (UNSW), Jan. 2016

NSA's data center in Bluffdale

NSA's data center in Bluffdale

Estimated numbers

- Electricity consumption: 65 MW
- Energy bill: US$\$40,000,000$/year
- Storage: 3–12 EB

NSA's data center in Bluffdale

Estimated numbers

- Electricity consumption: 65 MW
- Energy bill: US$\$40,000,000$/year
- Storage: 3–12 EB

The attack scenario

- Store encrypted data now
- Decrypt in 15 (?) years

NSA's data center in Bluffdale

Estimated numbers

- Electricity consumption: 65 MW
- Energy bill: US$40,000,000$/year
- Storage: 3–12 EB

The attack scenario

- Store encrypted data now
- Decrypt in 15 (?) years
- Consequence:

Need post-quantum (ephemeral) encryption keys now!

NSA's data center in Bluffdale

Estimated numbers

- Electricity consumption: 65 MW
- Energy bill: US$\$40,000,000$/year
- Storage: 3–12 EB

The attack scenario

- Store encrypted data now
- Decrypt in 15 (?) years
- Consequence:

  Need post-quantum (ephemeral) encryption keys now!
- Combine with pre-quantum key exchange to not lower security

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

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 $\mathbf{a}$

Peikert's RLWE-based KEM

Parameters: q, n, χ

KEM.Setup() :

$\mathbf{a} \xleftarrow{\$} \mathcal{R}_q$

Alice (server)		Bob (client)
KEM.Gen($\mathbf{a}$) :		KEM.Encaps($\mathbf{a}, \mathbf{b}$) :
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \chi$		$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \chi$
$\mathbf{b} \leftarrow \mathbf{as} + \mathbf{e}$	$\xrightarrow{\mathbf{b}}$	$\mathbf{u} \leftarrow \mathbf{as}' + \mathbf{e}'$
		$\mathbf{v} \leftarrow \mathbf{bs}' + \mathbf{e}''$
		$\bar{\mathbf{v}} \xleftarrow{\$} \mathsf{dbl}(\mathbf{v})$
KEM.Decaps($\mathbf{s}, (\mathbf{u}, \mathbf{v}')$) :	$\xleftarrow{\mathbf{u}, \mathbf{v}'}$	$\mathbf{v}' = \langle \bar{\mathbf{v}} \rangle_2$
$\mu \leftarrow \mathsf{rec}(2\mathbf{us}, \mathbf{v}')$		$\mu \leftarrow \lfloor \bar{\mathbf{v}} \rceil_2$

BCNS key exchange

- Bos, Costello, Naehrig, Stebila, IEEE S&P 2015:
  - Phrase the KEM as key exchange
  - Instantiate with concrete parameters
  - Integrate with OpenSSL $\rightarrow$ post-quantum TLS key exchange
  - Also: combined ECDH+RLWE key exchange

BCNS key exchange

- Bos, Costello, Naehrig, Stebila, IEEE S&P 2015:
  - Phrase the KEM as key exchange
  - Instantiate with concrete parameters
  - Integrate with OpenSSL $\rightarrow$ post-quantum TLS key exchange
  - Also: combined ECDH+RLWE key exchange
- Parameters chosen by BCNS:
  - $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$
  - $n = 1024$
  - $q = 2^{32} - 1$
  - $\chi = D_{\mathbb{Z},\sigma}$
  - $\sigma = 8\sqrt{2\pi} \approx 3.192$

BCNS key exchange

- Bos, Costello, Naehrig, Stebila, IEEE S&P 2015:
  - Phrase the KEM as key exchange
  - Instantiate with concrete parameters
  - Integrate with OpenSSL $\rightarrow$ post-quantum TLS key exchange
  - Also: combined ECDH+RLWE key exchange
- Parameters chosen by BCNS:
  - $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$
  - $n = 1024$
  - $q = 2^{32} - 1$
  - $\chi = D_{\mathbb{Z},\sigma}$
  - $\sigma = 8\sqrt{2\pi} \approx 3.192$
- Claimed security level: 128 bits pre-quantum
- Failure probability: $\approx 2^{-131072}$

A new hope

- Improve failure analysis and error reconciliation
- Choose parameters for failure probability $\approx 2^{-60}$

A new hope

- Improve failure analysis and error reconciliation
- Choose parameters for failure probability $\approx 2^{-60}$
- Drastically reduce q to $12289 < 2^{14}$
- Still use $n = 1024$

A new hope

- Improve failure analysis and error reconciliation
- Choose parameters for failure probability $\approx 2^{-60}$
- Drastically reduce q to $12289 < 2^{14}$
- Still use $n = 1024$
- Analysis of *post-quantum* security

A new hope

- Improve failure analysis and error reconciliation
- Choose parameters for failure probability $\approx 2^{-60}$
- Drastically reduce q to $12289 < 2^{14}$
- Still use $n = 1024$
- Analysis of *post-quantum* security
- Use centered binomial noise ψ_k ($\sum_{i=1}^{k} b_i - b'_i$ for $b_i, b'_i \in \{0,1\}$)

A new hope

- Improve failure analysis and error reconciliation
- Choose parameters for failure probability $\approx 2^{-60}$
- Drastically reduce q to $12289 < 2^{14}$
- Still use $n = 1024$
- Analysis of *post-quantum* security
- Use centered binomial noise ψ_k ($\sum_{i=1}^{k} b_i - b'_i$ for $b_i, b'_i \in \{0, 1\}$)
- Choose a fresh parameter $\mathbf{a}$ for every protocol run

A new hope

- Improve failure analysis and error reconciliation
- Choose parameters for failure probability $\approx 2^{-60}$
- Drastically reduce q to $12289 < 2^{14}$
- Still use $n = 1024$
- Analysis of *post-quantum* security
- Use centered binomial noise ψ_k ($\sum_{i=1}^{k} b_i - b'_i$ for $b_i, b'_i \in \{0,1\}$)
- Choose a fresh parameter $\mathbf{a}$ for every protocol run
- Encode polynomials in NTT domain

A new hope

- Improve failure analysis and error reconciliation
- Choose parameters for failure probability $\approx 2^{-60}$
- Drastically reduce q to $12289 < 2^{14}$
- Still use $n = 1024$
- Analysis of *post-quantum* security
- Use centered binomial noise ψ_k ($\sum_{i=1}^{k} b_i - b'_i$ for $b_i, b'_i \in \{0, 1\}$)
- Choose a fresh parameter $\mathbf{a}$ for every protocol run
- Encode polynomials in NTT domain
- Multiple implementations

A new hope – protocol

Parameters: $q = 12289 < 2^{14},\ n = 1024$

Error distribution: ψ_{16}

Alice (server)		**Bob (client)**
$seed \xleftarrow{\$} \{0,1\}^{256}$		
$\mathbf{a} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$		
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \psi_{16}^n$		$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \psi_{16}^n$
$\mathbf{b} \leftarrow \mathbf{as} + \mathbf{e}$	$\xrightarrow{(\mathbf{b}, seed)}$	$\mathbf{a} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$
		$\mathbf{u} \leftarrow \mathbf{as}' + \mathbf{e}'$
		$\mathbf{v} \leftarrow \mathbf{bs}' + \mathbf{e}''$
$\mathbf{v}' \leftarrow \mathbf{us}$	$\xleftarrow{(\mathbf{u}, \mathbf{r})}$	$\mathbf{r} \xleftarrow{\$} \text{HelpRec}(\mathbf{v})$
$k \leftarrow \text{Rec}(\mathbf{v}', \mathbf{r})$		$k \leftarrow \text{Rec}(\mathbf{v}, \mathbf{r})$
$\mu \leftarrow \text{SHA3-256}(k)$		$\mu \leftarrow \text{SHA3-256}(k)$

Error reconciliation

- After running the protocol
  - Alice has $\mathbf{x}_A = \mathbf{ass}' + \mathbf{e}'\mathbf{s}$
  - Bob has $\mathbf{x}_B = \mathbf{ass}' + \mathbf{es}' + \mathbf{e}''$
- Those elments are similar, but not the same
- Problem: How to agree on *the same* key from these noisy vectors?

Error reconciliation

- After running the protocol
  - Alice has $\mathbf{x}_A = \mathbf{ass}' + \mathbf{e}'\mathbf{s}$
  - Bob has $\mathbf{x}_B = \mathbf{ass}' + \mathbf{es}' + \mathbf{e}''$
- Those elments are similar, but not the same
- Problem: How to agree on *the same* key from these noisy vectors?
- Known: Extract one bit from each coefficient
- Also known: Extract multiple bits from each coefficient (decrease security)

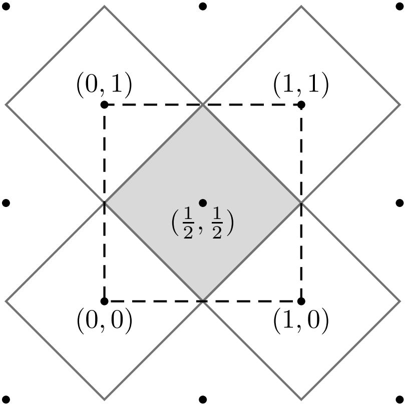
Error reconciliation

- After running the protocol
  - Alice has $\mathbf{x}_A = \mathbf{ass}' + \mathbf{e}'\mathbf{s}$
  - Bob has $\mathbf{x}_B = \mathbf{ass}' + \mathbf{es}' + \mathbf{e}''$
- Those elments are similar, but not the same
- Problem: How to agree on *the same* key from these noisy vectors?
- Known: Extract one bit from each coefficient
- Also known: Extract multiple bits from each coefficient (decrease security)
- Newhope: extract one bit from multiple coefficients (increase security)
- Specifically: 1 bit from 4 coefficients $\rightarrow$ 256-bit key from 1024 coefficients

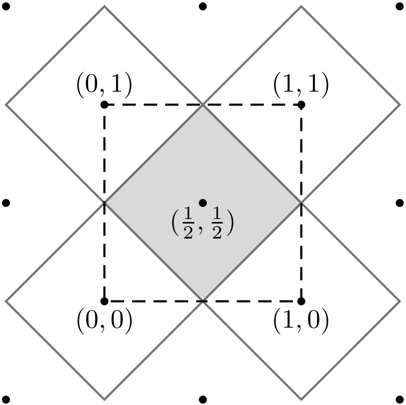
Error reconciliation

- After running the protocol
  - Alice has $\mathbf{x}_A = \mathbf{a}\mathbf{s}\mathbf{s}' + \mathbf{e}'\mathbf{s}$
  - Bob has $\mathbf{x}_B = \mathbf{a}\mathbf{s}\mathbf{s}' + \mathbf{e}\mathbf{s}' + \mathbf{e}''$
- Those elments are similar, but not the same
- Problem: How to agree on *the same* key from these noisy vectors?
- Known: Extract one bit from each coefficient
- Also known: Extract multiple bits from each coefficient (decrease security)
- Newhope: extract one bit from multiple coefficients (increase security)
- Specifically: 1 bit from 4 coefficients $\rightarrow$ 256-bit key from 1024 coefficients
- In the following: 2-dimensional intuition (4-dim. case very similar)
- "Scale" vector $\mathbf{x}$ to $[0, 1)^2$

2D Error reconciliation

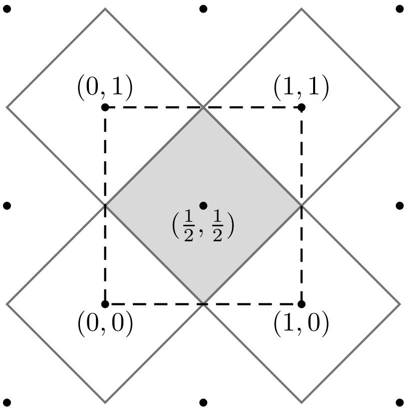


2D Error reconciliation



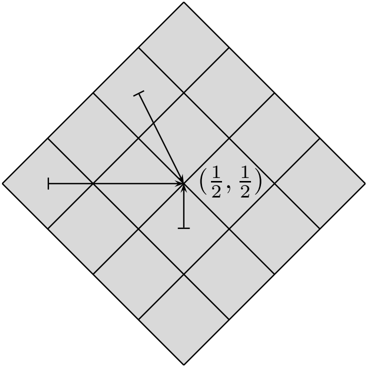
- If $\mathbf{x}$ is in the grey Voronoi cell: pick key bit 1
- If $\mathbf{x}$ is in the white Voronoi cell: pick key bit 0

2D Error reconciliation



- If $\mathbf{x}$ is in the grey Voronoi cell: pick key bit 1
- If $\mathbf{x}$ is in the white Voronoi cell: pick key bit 0
- Reconciliation: Bob sends difference vector from $\mathbf{x}_B$ to center of his Voronoi cell
- Alice adds this difference vector to her vector $\mathbf{x}_A$

Discretization of reconciliation



- Sending difference vector means doubling communcation
- Idea: chop Voronoi cell into 2^{dr} subcells
  - d: dimension (4 for NewHope)
  - r: discretization level
- Need to send only rd bits per d coefficients
- NewHope: $r = 2$; hence 256 bytes of reconciliation information

“Blurring the edges”

- This would all work if $\mathbf{x}$ was continuous uniform from $[0, 1)$
- We start with $\mathbf{x} \in \{0, \ldots, q-1\}^2$, q odd
- Odd number of possible values; no way to pick key bit without bias!
- This is the same for dimension 4

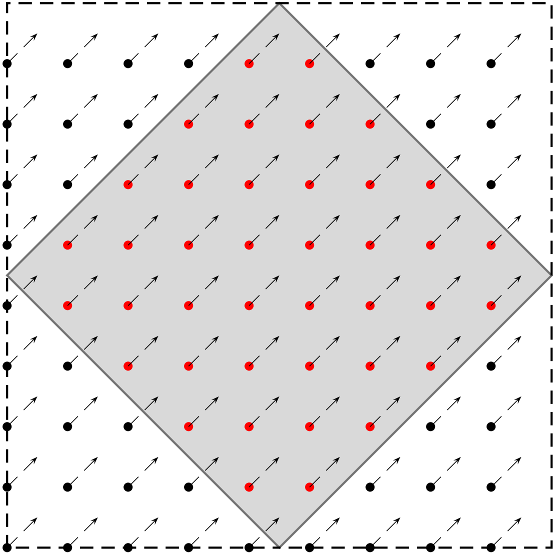
"Blurring the edges"

- This would all work if $\mathbf{x}$ was continuous uniform from $[0, 1)$
- We start with $\mathbf{x} \in \{0, \ldots, q-1\}^2$, q odd
- Odd number of possible values; no way to pick key bit without bias!
- This is the same for dimension 4
- Idea: randomly "blur the edges"
- Add vector $(1/2q, 1/2q)$ with probability $1/2$ before reconciliation

“Blurring the edges”

- This would all work if $\mathbf{x}$ was continuous uniform from $[0, 1)$
- We start with $\mathbf{x} \in \{0, \ldots, q-1\}^2$, q odd
- Odd number of possible values; no way to pick key bit without bias!
- This is the same for dimension 4
- Idea: randomly “blur the edges”
- Add vector $(1/2q, 1/2q)$ with probability $1/2$ before reconciliation
- This is a generalization of Peikert’s “randomized doubling” trick

"Blurring the edges"



Security analysis

- 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”)

Security analysis

- 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): $2^{0.2075n}$

Security analysis

- 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): $2^{0.2075n}$
- Primal attack: unique-SVP from LWE; solve using BKZ

Security analysis

- 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): $2^{0.2075n}$
- Primal attack: unique-SVP from LWE; solve using BKZ
- Dual attack: find short vector in dual lattice
- Length determines complexity and attacker's advantage ϵ

JarJar

> *"I don't like is the way that the parameters are set [. . .] I think that setting them too high impedes research."*
>
> —*anonymous reviewer*

JarJar

"I don't like is the way that the parameters are set [. . .] I think that setting them too high impedes research."

—anonymous reviewer

- JarJar: instantiation with $n = 512$
- Same $q = 12289$
- Use root lattice D_2 instead of D_4
- Use $k = 24$ for the centered binomial distribution

JarJar

"I don't like is the way that the parameters are set [...] I think that setting them too high impedes research."

—anonymous reviewer

- JarJar: instantiation with $n = 512$
- Same $q = 12289$
- Use root lattice D_2 instead of D_4
- Use $k = 24$ for the centered binomial distribution

JarJar is not recommended for use!

Post-quantum security

Attack	m	b	Known Classical	Known Quantum	Best Plausible
BCNS proposal: $q = 2^{32} - 1$, $n = 1024$, $\sigma = 3.192$					
Primal	1062	296	86	77	61
Dual	1042	259	84	**76**	62
JarJar: $q = 12289$, $n = 512$, $\sigma = \sqrt{12}$					
Primal	623	449	131	117	93
Dual	531	341	106	**95**	77
NewHope: $q = 12289$, $n = 1024$, $\sigma = \sqrt{8}$					
Primal	1100	967	282	253	200
Dual	938	761	229	**206**	165

- b: Block size for BKZ
- m: Number of used samples

Against all authority

- Remember the optimization of fixed $\mathbf{a}$?
- What if $\mathbf{a}$ is backdoored?
- Parameter-generating authority can break key exchange
- "Solution": Nothing-up-my-sleeves (involves endless discussion!)

Against all authority

- Remember the optimization of fixed $\mathbf{a}$?
- What if $\mathbf{a}$ 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 $\mathbf{a}$
  - Use precomputation to break *all* key exchanges
  - Infeasible today, but who knows...
  - Attack in the spirit of Logjam

Against all authority

- Remember the optimization of fixed $\mathbf{a}$?
- What if $\mathbf{a}$ 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 $\mathbf{a}$
  - Use precomputation to break *all* key exchanges
  - Infeasible today, but who knows...
  - Attack in the spirit of Logjam
- Solution in NewHope: Choose a fresh $\mathbf{a}$ every time
- Use SHAKE-128 to expand a 32-byte seed

Against all authority

- Remember the optimization of fixed $\mathbf{a}$?
- What if $\mathbf{a}$ 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 $\mathbf{a}$
  - Use precomputation to break *all* key exchanges
  - Infeasible today, but who knows...
  - Attack in the spirit of Logjam
- Solution in NewHope: Choose a fresh $\mathbf{a}$ every time
- Use SHAKE-128 to expand a 32-byte seed
- Server can cache $\mathbf{a}$ for some time (e.g., 1h)

Against all authority

- Remember the optimization of fixed $\mathbf{a}$?
- What if $\mathbf{a}$ 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 $\mathbf{a}$
  - Use precomputation to break *all* key exchanges
  - Infeasible today, but who knows...
  - Attack in the spirit of Logjam
- Solution in NewHope: Choose a fresh $\mathbf{a}$ every time
- Use SHAKE-128 to expand a 32-byte seed
- Server can cache $\mathbf{a}$ for some time (e.g., 1h)
- **Must not reuse keys/noise!**

Implementation

- Very fast multiplication in $\mathcal{R}_q$: use NTT
- Define message format:
  - Send polynomials in NTT domain
  - Eliminate two of the required NTTs

Implementation

- Very fast multiplication in $\mathcal{R}_q$: use NTT
- Define message format:
  - Send polynomials in NTT domain
  - Eliminate two of the required NTTs
- C reference implementation:
  - Arithmetic on 16-bit and 32-bit integers
  - No division (/) or modulo (%) operator
  - Use Montgomery reductions inside NTT
  - Use ChaCha20 for noise sampling

Implementation

- Very fast multiplication in $\mathcal{R}_q$: use NTT
- Define message format:
  - Send polynomials in NTT domain
  - Eliminate two of the required NTTs
- C reference implementation:
  - Arithmetic on 16-bit and 32-bit integers
  - No division (/) or modulo (%) operator
  - Use Montgomery reductions inside NTT
  - Use ChaCha20 for noise sampling
- AVX2 implementation:
  - Speed up NTT using vectorized `double` arithmetic
  - Use AES-256 for noise sampling
  - Use AVX2 for centered binomial

The protocol revisited

Parameters: $q = 12289 < 2^{14},\ n = 1024$

Error distribution: ψ_{16}^n

Alice (server)		Bob (client)
$seed \xleftarrow{\$} \{0, \ldots, 255\}^{32}$		
$\hat{\mathbf{a}} \leftarrow \mathsf{Parse}(\mathsf{SHAKE\text{-}128}(seed))$		
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \psi_{16}^n$		$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \psi_{16}^n$
$\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$		
$\hat{\mathbf{b}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{s}} + \mathsf{NTT}(\mathbf{e})$	$\xrightarrow[1824\ \mathsf{Bytes}]{m_a = \mathsf{encodeA}(seed, \hat{\mathbf{b}})}$	$(\hat{\mathbf{b}}, seed) \leftarrow \mathsf{decodeA}(m_a)$
		$\hat{\mathbf{a}} \leftarrow \mathsf{Parse}(\mathsf{SHAKE\text{-}128}(seed))$
		$\hat{\mathbf{t}} \leftarrow \mathsf{NTT}(\mathbf{s}')$
		$\hat{\mathbf{u}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{t}} + \mathsf{NTT}(\mathbf{e}')$
		$\mathbf{v} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{b}} \circ \hat{\mathbf{t}}) + \mathbf{e}''$
$(\hat{\mathbf{u}}, \mathbf{r}) \leftarrow \mathsf{decodeB}(m_b)$	$\xleftarrow[2048\ \mathsf{Bytes}]{m_b = \mathsf{encodeB}(\hat{\mathbf{u}}, \mathbf{r})}$	$\mathbf{r} \xleftarrow{\$} \mathsf{HelpRec}(\mathbf{v})$
$\mathbf{v}' \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{u}} \circ \hat{\mathbf{s}})$		$k \leftarrow \mathsf{Rec}(\mathbf{v}, \mathbf{r})$
$k \leftarrow \mathsf{Rec}(\mathbf{v}', \mathbf{r})$		$\mu \leftarrow \mathsf{SHA3\text{-}256}(k)$
$\mu \leftarrow \mathsf{SHA3\text{-}256}(k)$		

Vector computations

Scalar computation

- Load 32-bit integer a
- Load 32-bit integer b
- Perform addition $c \leftarrow a + b$
- Store 32-bit integer c

Vectorized computation

- Load 8 consecutive 32-bit integers $(a_0, a_1, \ldots, a_7)$
- Load 8 consecutive 32-bit integers $(b_0, b_1, \ldots, b_7)$
- Perform addition $(c_0, c_1, \ldots, c_7) \leftarrow (a_0 + b_0, a_1 + b_1, \ldots, a_7 + b_7)$
- Store 256-bit vector $(c_0, c_1, \ldots, c_7)$

Vector computations

Scalar computation

- Load 32-bit integer a
- Load 32-bit integer b
- Perform addition $c \leftarrow a + b$
- Store 32-bit integer c

Vectorized computation

- Load 8 consecutive 32-bit integers $(a_0, a_1, \ldots, a_7)$
- Load 8 consecutive 32-bit integers $(b_0, b_1, \ldots, b_7)$
- Perform addition $(c_0, c_1, \ldots, c_7) \leftarrow (a_0 + b_0, a_1 + b_1, \ldots, a_7 + b_7)$
- Store 256-bit vector $(c_0, c_1, \ldots, c_7)$

- Perform the same operations on independent data streams (SIMD)
- Vector instructions available on most “large” processors
- Instructions for vectors of bytes, integers, floats. . .

Vector computations

Scalar computation

- Load 32-bit integer a
- Load 32-bit integer b
- Perform addition $c \leftarrow a + b$
- Store 32-bit integer c

Vectorized computation

- Load 8 consecutive 32-bit integers $(a_0, a_1, \ldots, a_7)$
- Load 8 consecutive 32-bit integers $(b_0, b_1, \ldots, b_7)$
- Perform addition $(c_0, c_1, \ldots, c_7) \leftarrow (a_0 + b_0, a_1 + b_1, \ldots, a_7 + b_7)$
- Store 256-bit vector $(c_0, c_1, \ldots, c_7)$

- Perform the same operations on independent data streams (SIMD)
- Vector instructions available on most "large" processors
- Instructions for vectors of bytes, integers, floats...
- Need to interleave data items (e.g., 32-bit integers) in memory
- Compilers will not help with vectorization

Vector computations

Scalar computation

- Load 32-bit integer a
- Load 32-bit integer b
- Perform addition $c \leftarrow a + b$
- Store 32-bit integer c

Vectorized computation

- Load 8 consecutive 32-bit integers $(a_0, a_1, \ldots, a_7)$
- Load 8 consecutive 32-bit integers $(b_0, b_1, \ldots, b_7)$
- Perform addition $(c_0, c_1, \ldots, c_7) \leftarrow (a_0 + b_0, a_1 + b_1, \ldots, a_7 + b_7)$
- Store 256-bit vector $(c_0, c_1, \ldots, c_7)$

- Perform the same operations on independent data streams (SIMD)
- Vector instructions available on most "large" processors
- Instructions for vectors of bytes, integers, floats...
- Need to interleave data items (e.g., 32-bit integers) in memory
- Compilers will not really help with vectorization

Why would you care?

- Consider the Intel Haswell processor

Why would you care?

- Consider the Intel Haswell processor
  - 32-bit load throughput: 2 per cycle
  - 32-bit add throughput: 4 per cycle
  - 32-bit store throughput: 1 per cycle

Why would you care?

- Consider the Intel Haswell processor
  - 32-bit load throughput: 2 per cycle
  - 32-bit add throughput: 4 per cycle
  - 32-bit store throughput: 1 per cycle
  - 256-bit load throughput: 1 per cycle
  - $8\times$ 32-bit add throughput: 2 per cycle
  - 256-bit store throughput: 1 per cycle

Why would you care?

- Consider the Intel Haswell processor
  - 32-bit load throughput: 2 per cycle
  - 32-bit add throughput: 4 per cycle
  - 32-bit store throughput: 1 per cycle
  - 256-bit load throughput: 1 per cycle
  - $8\times$ 32-bit add throughput: 2 per cycle
  - 256-bit store throughput: 1 per cycle
- **Vector instructions are almost as fast as scalar instructions but do 4–8$\times$ the work**

Why would you care?

- Consider the Intel Haswell processor
  - 32-bit load throughput: 2 per cycle
  - 32-bit add throughput: 4 per cycle
  - 32-bit store throughput: 1 per cycle
  - 256-bit load throughput: 1 per cycle
  - $8\times$ 32-bit add throughput: 2 per cycle
  - 256-bit store throughput: 1 per cycle
- **Vector instructions are almost as fast as scalar instructions but do 4–8$\times$ the work**
- Situation on other architectures/microarchitectures is similar

Why would you care? (Part II)

- Data-dependent branches are expensive in SIMD
- Variably indexed loads (lookups) into vectors are expensive
- Need to rewrite algorithms to eliminate branches and lookups

Why would you care? (Part II)

- Data-dependent branches are expensive in SIMD
- Variably indexed loads (lookups) into vectors are expensive
- Need to rewrite algorithms to eliminate branches and lookups
- Secret-data-dependent branches and secret branch conditions are the major sources of timing-attack vulnerabilities

Why would you care? (Part II)

- Data-dependent branches are expensive in SIMD
- Variably indexed loads (lookups) into vectors are expensive
- Need to rewrite algorithms to eliminate branches and lookups
- Secret-data-dependent branches and secret branch conditions are the major sources of timing-attack vulnerabilities
- Strong synergies between speeding up code with vector instructions and protecting code!

Vectorizing the NTT

- Essentially reuse code from PQCrypto 2013
- Represent elements of $\mathbb{Z}_q$ as double-precision floats

Vectorizing the NTT

- Essentially reuse code from PQCrypto 2013
- Represent elements of $\mathbb{Z}_q$ as double-precision floats
- Reduction mod q:
  - Multiply by approximate inverse of q
  - Round/Truncate
  - Multiply by q
  - Subtract

Vectorizing the NTT

- Essentially reuse code from PQCrypto 2013
- Represent elements of $\mathbb{Z}_q$ as double-precision floats
- Reduction mod q:
  - Multiply by approximate inverse of q
  - Round/Truncate
  - Multiply by q
  - Subtract
- Performance: $\approx 11,000$ cycles,
  - including multiplication by powers of γ
  - excluding bit reversal

Vectorizing the NTT

- Essentially reuse code from PQCrypto 2013
- Represent elements of $\mathbb{Z}_q$ as double-precision floats
- Reduction mod q:
  - Multiply by approximate inverse of q
  - Round/Truncate
  - Multiply by q
  - Subtract
- Performance: $\approx 11,000$ cycles,
  - including multiplication by powers of γ
  - excluding bit reversal
- TODO: Revisit NTT, use integer arithmetic

Performance

	BCNS	**C ref**	**AVX2**
Key generation (server)	$\approx 2\,477\,958$	$271\,650$ $(272\,174)$	$115\,414$ $(115\,746)$
Key gen + shared key (client)	$\approx 3\,995\,977$	$402\,058$ $(402\,285)$	$144\,788$ $(144\,957)$
Shared key (server)	$\approx 481\,937$	$86\,584$	$23\,988$

- Benchmarks on one core of an Intel i7-4770K (Haswell)
- BCNS benchmarks are derived from `openssl speed`
- Numbers in parantheses are average; all other numbers are median.
- Includes around $\approx 57\,000$ cycles for generation of $\mathbf{a}$ on each side

Performance

	BCNS	C ref	AVX2
Key generation (server)	$\approx 2\,477\,958$	$271\,650$ $(272\,174)$	$115\,414$ $(115\,746)$
Key gen + shared key (client)	$\approx 3\,995\,977$	$402\,058$ $(402\,285)$	$144\,788$ $(144\,957)$
Shared key (server)	$\approx 481\,937$	$86\,584$	$23\,988$

- Benchmarks on one core of an Intel i7-4770K (Haswell)
- BCNS benchmarks are derived from `openssl speed`
- Numbers in parantheses are average; all other numbers are median.
- Includes around $\approx 57\,000$ cycles for generation of $\mathbf{a}$ on each side
- Faster than state-of-the art ECC

NewHope on ARM Cortex M*

- Joint work with Erdem Alkim and Philipp Jakubeit
- Optimize NewHope on Cortex M0 and M4
- 32-bit state-of-the art microcontrollers

NewHope on ARM Cortex M*

- Joint work with Erdem Alkim and Philipp Jakubeit
- Optimize NewHope on Cortex M0 and M4
- 32-bit state-of-the art microcontrollers
- Start with C reference implementation
- New speed records for NTT from:
  - Montgomery reductions after constant multiplications
  - "Short Barrett reductions" after additions
  - Lazy reductions
  - Serious hand optimization in assembly

NewHope on ARM Cortex M*

- Joint work with Erdem Alkim and Philipp Jakubeit
- Optimize NewHope on Cortex M0 and M4
- 32-bit state-of-the art microcontrollers
- Start with C reference implementation
- New speed records for NTT from:
  - Montgomery reductions after constant multiplications
  - "Short Barrett reductions" after additions
  - Lazy reductions
  - Serious hand optimization in assembly
- Also optimize other building blocks of NewHope

The NTT (in C)

- 10 levels of butterfly transformations
- Each level uses 512 Gentleman-Sande Butterflies

```
W = omega[jTwiddle++]; // W is in Montgomery domain
t = a[j];
a[j] = barrett_red(t + a[j+d]);
a[j+d] = montgomery_red(W * ((uint32_t)t + 3*12289 - a[j+d]));
```

The NTT (in C)

- 10 levels of butterfly transformations
- Each level uses 512 Gentleman-Sande Butterflies

```
W = omega[jTwiddle++]; // W is in Montgomery domain
t = a[j];
a[j] = barrett_red(t + a[j+d]);
a[j+d] = montgomery_red(W * ((uint32_t)t + 3*12289 - a[j+d]));
```

- Every second level omits the `barrett_red`

Montgomery reduction

```
uint16_t montgomery_red(uint32_t a) {
  uint32_t u;
  u = (a * 12287);
  u &= ((1 << 18) - 1);
  a += u * 12289;
  return a >> 18;
}
```

- Use Montgomery parameter $R = 2^{18}$
- Works for inputs in $\{0, \ldots, 2^{32} - q(R-1) - 1\}$
- Output is guaranteed to have at most 14 bits

Short Barrett reduction

```
uint16_t barrett_red(uint16_t a) {
  uint32_t u;
  u = ((uint32_t) a * 5) >> 16;
  a -= u * 12289;
  return a;
}
```

- Accepts any 16-bit unsigned int as input
- Produces outputs of at most 14 bits

ARM Cortex-M0 results

- Server side: ≈ 1.68Mio cycles (M0) and $\approx 870\,000$ cycles (M4)
- Client side: ≈ 2.05Mio cycles (M0) and $\approx 985\,000$ cycles (M4)

ARM Cortex-M0 results

- Server side: ≈ 1.68Mio cycles (M0) and $\approx 870\,000$ cycles (M4)
- Client side: ≈ 2.05Mio cycles (M0) and $\approx 985\,000$ cycles (M4)
- Comparison to ECC: ≈ 3.59 cycles for X25519 scalar mult on M0

ARM Cortex-M0 results

- Server side: ≈ 1.68Mio cycles (M0) and $\approx 870\,000$ cycles (M4)
- Client side: ≈ 2.05Mio cycles (M0) and $\approx 985\,000$ cycles (M4)
- Comparison to ECC: ≈ 3.59 cycles for X25519 scalar mult on M0
- Comparison to HECC: ≈ 2.63 cycles on Kummer surface on M0

Newhope questions

- Try error-correcting codes for reconciliation?

Newhope questions

- Try error-correcting codes for reconciliation?
- Send polynomials in "normal" domain?
  - Decouple protocol from multiplication algorithm
  - Possibly drop least significant bits

Newhope questions

- Try error-correcting codes for reconciliation?
- Send polynomials in "normal" domain?
  - Decouple protocol from multiplication algorithm
  - Possibly drop least significant bits
- Use smaller q?
  - Smaller messages
  - Higher security
  - Does not support efficient negacyclic NTT

Newhope questions

- Try error-correcting codes for reconciliation?
- Send polynomials in “normal” domain?
  - Decouple protocol from multiplication algorithm
  - Possibly drop least significant bits
- Use smaller q?
  - Smaller messages
  - Higher security
  - Does not support efficient negacyclic NTT
- How about Nussbaumer’s algorithm for multiplication?

Newhope questions

- Try error-correcting codes for reconciliation?
- Send polynomials in “normal” domain?
  - Decouple protocol from multiplication algorithm
  - Possibly drop least significant bits
- Use smaller q?
  - Smaller messages
  - Higher security
  - Does not support efficient negacyclic NTT
- How about Nussbaumer’s algorithm for multiplication?
- How about Karatsuba + Toom for multiplication?

Newhope questions

- Try error-correcting codes for reconciliation?
- Send polynomials in "normal" domain?
  - Decouple protocol from multiplication algorithm
  - Possibly drop least significant bits
- Use smaller q?
  - Smaller messages
  - Higher security
  - Does not support efficient negacyclic NTT
- How about Nussbaumer's algorithm for multiplication?
- How about Karatsuba + Toom for multiplication?
- How about smaller n (e.g., $n \approx 800$)?

NewHope online

Paper:	`https://cryptojedi.org/papers/#newhope`
Software:	`https://cryptojedi.org/crypto/#newhope`
ARM software:	`https://github.com/newhopearm/newhopearm.git`
Newhope in Go:	`https://github.com/Yawning/newhope` (by Yawning Angel)
Newhope in Rust:	`https://code.ciph.re/isis/newhopers` (by Isis Lovecruft)
Newhope in D:	`https://cryptojedi.org/crypto/#newhope` (?) (soon???)