Engineering lattice-based cryptography

Peter Schwabe
peter@cryptojedi.org
https://cryptojedi.org
September 30, 2019

Crypto today

5 building blocks for a "secure channel"
Symmetric crypto

- Block or stream cipher (e.g., AES, ChaCha20)
- Authenticator (e.g., HMAC, GMAC, Poly1305)
- Hash function (e.g., SHA-2, SHA-3)

Crypto today

5 building blocks for a "secure channel"

Symmetric crypto

- Block or stream cipher (e.g., AES, ChaCha20)
- Authenticator (e.g., HMAC, GMAC, Poly1305)
- Hash function (e.g., SHA-2, SHA-3)

Asymmetric crypto

- Key agreement / public-key encryption (e.g., RSA, Diffie-Hellman, ECDH)
- Signatures (e.g., RSA, DSA, ECDSA, EdDSA)

Crypto today

5 building blocks for a "secure channel"

Symmetric crypto

- Block or stream cipher (e.g., AES, ChaCha20)
- Authenticator (e.g., HMAC, GMAC, Poly1305)
- Hash function (e.g., SHA-2, SHA-3)

Asymmetric crypto

- Key agreement / public-key encryption (e.g., RSA, Diffie-Hellman, ECDH)
- Signatures (e.g., RSA, DSA, ECDSA, EdDSA)

The asymmetric monoculture

- All widely deployed asymmetric crypto relies on
  - the **hardness of factoring**, or
  - the **hardness of (elliptic-curve) discrete logarithms**

. . . Shor, 1996

Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*

Peter W. Shor†

Abstract

A digital computer is generally believed to be an efficient universal computing device; that is, it is believed able to simulate any physical computing device with an increase in computation time by at most a polynomial factor. This may not be true when quantum mechanics is taken into consideration. This paper considers factoring integers and finding discrete logarithms, two problems which are generally thought to be hard on a classical computer and which have been used as the basis of several proposed cryptosystems. Efficient randomized algorithms are given for these two problems on a hypothetical quantum computer. These algorithms take a number of steps polynomial in the input size, e.g., the number of digits of the integer to be factored.

Will there be quantum computers?

"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

Post-quantum crypto

Definition

Post-quantum crypto is (asymmetric) crypto that resists attacks using classical *and quantum* computers.

Post-quantum crypto

Definition
Post-quantum crypto is (asymmetric) crypto that resists attacks using classical *and quantum* computers.

5 main directions

- Lattice-based crypto (PKE and Sigs)
- Code-based crypto (mainly PKE)
- Multivariate-based crypto (mainly Sigs)
- Hash-based signatures (only Sigs)
- Isogeny-based crypto (so far, mainly PKE)

The NIST competition, initial overview

Count of Problem Category	Column Labels		
Row Labels	Key Exchange	Signature	Grand Total
?	1		1
Braids	1	1	2
Chebychev	1		1
Codes	19	5	24
Finite Automata	1	1	2
Hash		4	4
Hypercomplex Numbers	1		1
Isogeny	1		1
Lattice	24	4	28
Mult. Var	6	7	13
Rand. walk	1		1
RSA	1	1	2
Grand Total	**57**	**23**	**80**

Overview tweeted by Jacob Alperin-Sheriff on Dec 4, 2017.

The NIST competition (ctd.)

"Key exchange"

- What is meant is **key encapsulation mechanisms** (KEMs)
  - $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$
  - $(c, k) \leftarrow \mathsf{Encaps}(\mathsf{vk})$
  - $k \leftarrow \mathsf{Decaps}(c, \mathsf{sk})$

The NIST competition (ctd.)

"Key exchange"

- What is meant is **key encapsulation mechanisms** (KEMs)
  - (vk, sk)←KeyGen()
  - (c, k)←Encaps(vk)
  - k←Decaps(c, sk)

Status of the NIST competition

- In total 69 submissions accepted as "complete and proper"
- Several broken, 5 withdrawn
- Jan 2019: NIST announces 26 round-2 candidates
  - 17 KEMs and PKEs
  - 9 signature schemes

Round-2 overview

Signature schemes

- 3 lattice-based
- 2 symmetric-crypto based
- 4 MQ-based

Round-2 overview

Signature schemes

- 3 lattice-based
- 2 symmetric-crypto based
- 4 MQ-based

KEMs/PKE

- 9 lattice-based
- 7 code-based
- 1 isogeny-based

Round-2 overview

Signature schemes

- 3 lattice-based
- 2 symmetric-crypto based
- 4 MQ-based

KEMs/PKE

- 9 lattice-based
- 7 code-based
- 1 isogeny-based

Lattice-based KEMs


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

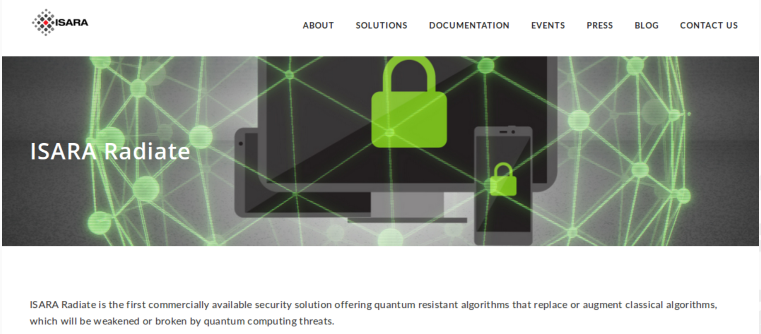
Posted by Matt Braithwaite, Software Engineer

Search blog ...

Archive

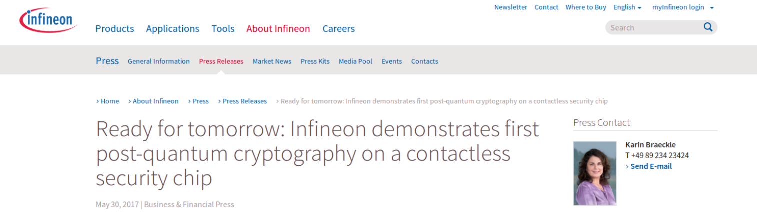
“We’re indebted to Erdem Alkim, Léo Ducas, Thomas Pöppelmann and Peter Schwabe, the researchers who developed “New Hope”, the post-quantum algorithm that we selected for this experiment.”

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/



"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

Learning with errors (LWE)

- Given uniform $\mathbf{A} \in \mathbb{Z}_q^{k \times \ell}$
- Given "noise distribution" χ
- Given samples $\mathbf{A}\mathbf{s} + \mathbf{e}$, with $\mathbf{e} \leftarrow \chi$

Learning with errors (LWE)

- Given uniform $\mathbf{A} \in \mathbb{Z}_q^{k \times \ell}$
- Given "noise distribution" χ
- Given samples $\mathbf{As} + \mathbf{e}$, with $\mathbf{e} \leftarrow \chi$
- Search version: find $\mathbf{s}$
- Decision version: distinguish from uniform random

Learning with rounding (LWR)

- Given uniform $\mathbf{A} \in \mathbb{Z}_q^{k \times \ell}$
- Given samples $\lceil \mathbf{As} \rfloor_p$, with $p < q$

Learning with rounding (LWR)

- Given uniform $\mathbf{A} \in \mathbb{Z}_q^{k \times \ell}$
- Given samples $\lceil \mathbf{As} \rfloor_p$, with $p < q$
- Search version: find $\mathbf{s}$
- Decision version: distinguish from uniform random

Using structured lattices

- Problem with LWE-based cryptosystems: public-key size
- Only NIST candidate exclusively using standard LWE: FrodoKEM

Using structured lattices

- Problem with LWE-based cryptosystems: public-key size
- Only NIST candidate exclusively using standard LWE: FrodoKEM
- Idea to solve this: allow structured matrix $\mathbf{A}$, e.g.,

Using structured lattices

- Problem with LWE-based cryptosystems: public-key size
- Only NIST candidate exclusively using standard LWE: FrodoKEM
- Idea to solve this: allow structured matrix **A**, e.g.,
  - NewHope: work in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$; n a power of 2, q prime

Using structured lattices

- Problem with LWE-based cryptosystems: public-key size
- Only NIST candidate exclusively using standard LWE: FrodoKEM
- Idea to solve this: allow structured matrix $\mathbf{A}$, e.g.,
  - NewHope: work in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$; n a power of 2, q prime
  - NTRU: work in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n - 1)$; n prime, q a power of 2

Using structured lattices

- Problem with LWE-based cryptosystems: public-key size
- Only NIST candidate exclusively using standard LWE: FrodoKEM
- Idea to solve this: allow structured matrix **A**, e.g.,
  - NewHope: work in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$; n a power of 2, q prime
  - NTRU: work in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n - 1)$; n prime, q a power of 2
  - NTRU Prime: work in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n - X - 1)$; q prime, n prime

Using structured lattices

- Problem with LWE-based cryptosystems: public-key size
- Only NIST candidate exclusively using standard LWE: FrodoKEM
- Idea to solve this: allow structured matrix $\mathbf{A}$, e.g.,
  - NewHope: work in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$; n a power of 2, q prime
  - NTRU: work in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n - 1)$; n prime, q a power of 2
  - NTRU Prime: work in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n - X - 1)$; q prime, n prime
  - Kyber/Saber: use small-dimension matrices and vectors over $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^{256} + 1)$

Using structured lattices

- Problem with LWE-based cryptosystems: public-key size
- Only NIST candidate exclusively using standard LWE: FrodoKEM
- Idea to solve this: allow structured matrix **A**, e.g.,
  - NewHope: work in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$; n a power of 2, q prime
  - NTRU: work in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n - 1)$; n prime, q a power of 2
  - NTRU Prime: work in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n - X - 1)$; q prime, n prime
  - Kyber/Saber: use small-dimension matrices and vectors over $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^{256} + 1)$
- Perform arithmetic on (vectors of) polynomials instead of vectors/matrices over $\mathbb{Z}_q$

How to build a KEM?

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

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

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

How to build a KEM, part 2

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

How to build a KEM, part 2

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

How to build a KEM, part 2

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

How to build a KEM, part 2

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

How to build a KEM, part 2

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

How to build a KEM, part 2

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

How to build a KEM, part 2

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

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

From passive to CCA security

- The base scheme does not have active security
- Attacker can choose arbitrary noise, learns **s** from failures

From passive to CCA security

- The base scheme does not have active security
- Attacker can choose arbitrary noise, learns **s** from failures
- Fujisaki-Okamoto transform (sketched):

Alice (Server)		Bob (Client)
Gen():		Enc(seed, $\mathbf{b}$):
$\text{pk}, \text{sk} \leftarrow \text{KeyGen}()$		$x \leftarrow \{0, \ldots, 255\}^{32}$
$\text{seed}, \mathbf{b} \leftarrow \text{pk}$	$\xrightarrow{\text{seed}, \mathbf{b}}$	$x \leftarrow \text{SHA3-256}(x)$
		$k, \text{coins} \leftarrow \text{SHA3-512}(x)$
	$\xleftarrow{\mathbf{u}, v}$	$\mathbf{u}, v \leftarrow \text{Encrypt}((\text{seed}, \mathbf{b}), x, \text{coins})$
Dec($\mathbf{s}$, ($\mathbf{u}$, v)):		
$x' \leftarrow \text{Decrypt}(\mathbf{s}, (\mathbf{u}, v))$		
$k', coins' \leftarrow \text{SHA3-512}(x')$		
$\mathbf{u}', v' \leftarrow \text{Encrypt}((\text{seed}, \mathbf{b}), x', \text{coins}')$		
verify if $(\mathbf{u}', v') = (\mathbf{u}, v)$		

Design space 0: The NTRU approach

- Historically first: NTRU
- Use parameters q and $p = 3$

Design space 0: The NTRU approach

- Historically first: NTRU
- Use parameters q and $p = 3$
- **Keygen:**
  - Find $\mathbf{f}, \mathbf{g} \in \mathcal{R}_q$ and $\mathbf{f}_q = \mathbf{f}^{-1} \mod q, \mathbf{f}_p = \mathbf{f}^{-1} \mod p$
  - public key: $\mathbf{h} = p\mathbf{f}_q\mathbf{g}$, secret key: $(\mathbf{f}, \mathbf{f}_p)$

Design space 0: The NTRU approach

- Historically first: NTRU
- Use parameters q and $p = 3$
- **Keygen:**
  - Find $\mathbf{f}, \mathbf{g} \in \mathcal{R}_q$ and $\mathbf{f}_q = \mathbf{f}^{-1} \mod q, \mathbf{f}_p = \mathbf{f}^{-1} \mod p$
  - public key: $\mathbf{h} = p\mathbf{f}_q\mathbf{g}$, secret key: $(\mathbf{f}, \mathbf{f}_p)$
- **Encrypt:**
  - Map message m to $\mathbf{m} \in \mathcal{R}_q$ with coefficients in $\{-1, 0, 1\}$
  - Sample random small-coefficient polynomial $\mathbf{r} \in \mathcal{R}_q$
  - Compute ciphertext $\mathbf{e} = \mathbf{r} \cdot \mathbf{h} + \mathbf{m}$

Design space 0: The NTRU approach

- Historically first: NTRU
- Use parameters q and $p = 3$
- **Keygen:**
  - Find $\mathbf{f}, \mathbf{g} \in \mathcal{R}_q$ and $\mathbf{f}_q = \mathbf{f}^{-1} \mod q, \mathbf{f}_p = \mathbf{f}^{-1} \mod p$
  - public key: $\mathbf{h} = p\mathbf{f}_q\mathbf{g}$, secret key: $(\mathbf{f}, \mathbf{f}_p)$
- **Encrypt:**
  - Map message m to $\mathbf{m} \in \mathcal{R}_q$ with coefficients in $\{-1, 0, 1\}$
  - Sample random small-coefficient polynomial $\mathbf{r} \in \mathcal{R}_q$
  - Compute ciphertext $\mathbf{e} = \mathbf{r} \cdot \mathbf{h} + \mathbf{m}$
- **Decrypt:**
  - Compute $\mathbf{v} = \mathbf{f} \cdot \mathbf{e}$

Design space 0: The NTRU approach

- Historically first: NTRU
- Use parameters q and $p = 3$
- **Keygen:**
  - Find $\mathbf{f}, \mathbf{g} \in \mathcal{R}_q$ and $\mathbf{f}_q = \mathbf{f}^{-1} \mod q, \mathbf{f}_p = \mathbf{f}^{-1} \mod p$
  - public key: $\mathbf{h} = p\mathbf{f}_q\mathbf{g}$, secret key: $(\mathbf{f}, \mathbf{f}_p)$
- **Encrypt:**
  - Map message m to $\mathbf{m} \in \mathcal{R}_q$ with coefficients in $\{-1, 0, 1\}$
  - Sample random small-coefficient polynomial $\mathbf{r} \in \mathcal{R}_q$
  - Compute ciphertext $\mathbf{e} = \mathbf{r} \cdot \mathbf{h} + \mathbf{m}$
- **Decrypt:**
  - Compute $\mathbf{v} = \mathbf{f} \cdot \mathbf{e} = \mathbf{f} \cdot (\mathbf{r} \cdot \mathbf{h} + \mathbf{m})$

Design space 0: The NTRU approach

- Historically first: NTRU
- Use parameters q and $p = 3$
- **Keygen:**
  - Find $\mathbf{f}, \mathbf{g} \in \mathcal{R}_q$ and $\mathbf{f}_q = \mathbf{f}^{-1} \mod q, \mathbf{f}_p = \mathbf{f}^{-1} \mod p$
  - public key: $\mathbf{h} = p\mathbf{f}_q\mathbf{g}$, secret key: $(\mathbf{f}, \mathbf{f}_p)$
- **Encrypt:**
  - Map message m to $\mathbf{m} \in \mathcal{R}_q$ with coefficients in $\{-1, 0, 1\}$
  - Sample random small-coefficient polynomial $\mathbf{r} \in \mathcal{R}_q$
  - Compute ciphertext $\mathbf{e} = \mathbf{r} \cdot \mathbf{h} + \mathbf{m}$
- **Decrypt:**
  - Compute $\mathbf{v} = \mathbf{f} \cdot \mathbf{e} = \mathbf{f} \cdot (\mathbf{r} \cdot \mathbf{h} + \mathbf{m}) = \mathbf{f}(\mathbf{r} \cdot (p\mathbf{f}_q\mathbf{g}) + \mathbf{m})$

Design space 0: The NTRU approach

- Historically first: NTRU
- Use parameters q and $p = 3$
- **Keygen:**
  - Find $\mathbf{f}, \mathbf{g} \in \mathcal{R}_q$ and $\mathbf{f}_q = \mathbf{f}^{-1} \mod q, \mathbf{f}_p = \mathbf{f}^{-1} \mod p$
  - public key: $\mathbf{h} = p\mathbf{f}_q\mathbf{g}$, secret key: $(\mathbf{f}, \mathbf{f}_p)$
- **Encrypt:**
  - Map message m to $\mathbf{m} \in \mathcal{R}_q$ with coefficients in $\{-1, 0, 1\}$
  - Sample random small-coefficient polynomial $\mathbf{r} \in \mathcal{R}_q$
  - Compute ciphertext $\mathbf{e} = \mathbf{r} \cdot \mathbf{h} + \mathbf{m}$
- **Decrypt:**
  - Compute $\mathbf{v} = \mathbf{f} \cdot \mathbf{e} = \mathbf{f} \cdot (\mathbf{r} \cdot \mathbf{h} + \mathbf{m}) = \mathbf{f}(\mathbf{r} \cdot (p\mathbf{f}_q\mathbf{g}) + \mathbf{m}) = p\mathbf{r}\mathbf{g} + \mathbf{f} \cdot \mathbf{m}$

Design space 0: The NTRU approach

- Historically first: NTRU
- Use parameters q and $p = 3$
- **Keygen:**
  - Find $\mathbf{f}, \mathbf{g} \in \mathcal{R}_q$ and $\mathbf{f}_q = \mathbf{f}^{-1} \mod q, \mathbf{f}_p = \mathbf{f}^{-1} \mod p$
  - public key: $\mathbf{h} = p\mathbf{f}_q\mathbf{g}$, secret key: $(\mathbf{f}, \mathbf{f}_p)$
- **Encrypt:**
  - Map message m to $\mathbf{m} \in \mathcal{R}_q$ with coefficients in $\{-1, 0, 1\}$
  - Sample random small-coefficient polynomial $\mathbf{r} \in \mathcal{R}_q$
  - Compute ciphertext $\mathbf{e} = \mathbf{r} \cdot \mathbf{h} + \mathbf{m}$
- **Decrypt:**
  - Compute $\mathbf{v} = \mathbf{f} \cdot \mathbf{e} = \mathbf{f} \cdot (\mathbf{r} \cdot \mathbf{h} + \mathbf{m}) = \mathbf{f}(\mathbf{r} \cdot (p\mathbf{f}_q\mathbf{g}) + \mathbf{m}) = p\mathbf{r}\mathbf{g} + \mathbf{f} \cdot \mathbf{m}$
  - Compute $\mathbf{m} = \mathbf{v} \cdot \mathbf{f}_p \mod p$

Design space 0: The NTRU approach

- Historically first: NTRU
- Use parameters q and $p = 3$
- **Keygen:**
  - Find $\mathbf{f}, \mathbf{g} \in \mathcal{R}_q$ and $\mathbf{f}_q = \mathbf{f}^{-1} \mod q, \mathbf{f}_p = \mathbf{f}^{-1} \mod p$
  - public key: $\mathbf{h} = p\mathbf{f}_q\mathbf{g}$, secret key: $(\mathbf{f}, \mathbf{f}_p)$
- **Encrypt:**
  - Map message m to $\mathbf{m} \in \mathcal{R}_q$ with coefficients in $\{-1, 0, 1\}$
  - Sample random small-coefficient polynomial $\mathbf{r} \in \mathcal{R}_q$
  - Compute ciphertext $\mathbf{e} = \mathbf{r} \cdot \mathbf{h} + \mathbf{m}$
- **Decrypt:**
  - Compute $\mathbf{v} = \mathbf{f} \cdot \mathbf{e} = \mathbf{f} \cdot (\mathbf{r} \cdot \mathbf{h} + \mathbf{m}) = \mathbf{f}(\mathbf{r} \cdot (p\mathbf{f}_q\mathbf{g}) + \mathbf{m}) = p\mathbf{r}\mathbf{g} + \mathbf{f} \cdot \mathbf{m}$
  - Compute $\mathbf{m} = \mathbf{v} \cdot \mathbf{f}_p \mod p$
- Advantages/Disadvantages compared to LPR:
  - Asymptotically weaker than Ring-LWE approach
  - Slower keygen, but faster encryption/decryption

Design space 1: What ring?

- Structured lattice-based schemes use ring $\mathcal{R}_q = \mathbb{Z}_q[X]/f$
  - q typically either prime or a power of two
  - f typically of degree between 512 and 1024

Design space 1: What ring?

- Structured lattice-based schemes use ring $\mathcal{R}_q = \mathbb{Z}_q[X]/f$
  - q typically either prime or a power of two
  - f typically of degree between 512 and 1024
- **First option:** $q = 2^k$, $f = (X^n - 1)$, n prime (NTRU)

Design space 1: What ring?

- Structured lattice-based schemes use ring $\mathcal{R}_q = \mathbb{Z}_q[X]/f$
  - q typically either prime or a power of two
  - f typically of degree between 512 and 1024
- **First option:** $q = 2^k$, $f = (X^n - 1)$, n prime (NTRU)
- **Second option:** $q = 2^k$, $f = (X^n + 1)$, $n = 2^m$ (Saber)

Design space 1: What ring?

- Structured lattice-based schemes use ring $\mathcal{R}_q = \mathbb{Z}_q[X]/f$
  - q typically either prime or a power of two
  - f typically of degree between 512 and 1024
- **First option:** $q = 2^k$, $f = (X^n - 1)$, n prime (NTRU)
- **Second option:** $q = 2^k$, $f = (X^n + 1)$, $n = 2^m$ (Saber)
- **Third option:** $q = 2^k$, $f = \Phi_{n+1}$, $n + 1$ prime (Round5)

Design space 1: What ring?

- Structured lattice-based schemes use ring $\mathcal{R}_q = \mathbb{Z}_q[X]/f$
  - q typically either prime or a power of two
  - f typically of degree between 512 and 1024
- **First option:** $q = 2^k$, $f = (X^n - 1)$, n prime (NTRU)
- **Second option:** $q = 2^k$, $f = (X^n + 1)$, $n = 2^m$ (Saber)
- **Third option:** $q = 2^k$, $f = \Phi_{n+1}$, $n+1$ prime (Round5)
- **Fourth option:** q prime, $f = (X^n + 1) = \Phi_{2n}$, $n = 2^m$ (NewHope, Kyber, LAC)

Design space 1: What ring?

- Structured lattice-based schemes use ring $\mathcal{R}_q = \mathbb{Z}_q[X]/f$
  - q typically either prime or a power of two
  - f typically of degree between 512 and 1024
- **First option:** $q = 2^k$, $f = (X^n - 1)$, n prime (NTRU)
- **Second option:** $q = 2^k$, $f = (X^n + 1)$, $n = 2^m$ (Saber)
- **Third option:** $q = 2^k$, $f = \Phi_{n+1}$, $n + 1$ prime (Round5)
- **Fourth option:** q prime, $f = (X^n + 1) = \Phi_{2n}$, $n = 2^m$ (NewHope, Kyber, LAC)
- **Fifth option:** q prime, $f = (X^n - X - 1)$ irreducible, n prime (NTRU Prime)

Design space 1: What ring?

- Structured lattice-based schemes use ring $\mathcal{R}_q = \mathbb{Z}_q[X]/f$
  - q typically either prime or a power of two
  - f typically of degree between 512 and 1024
- **First option:** $q = 2^k$, $f = (X^n - 1)$, n prime (NTRU)
- **Second option:** $q = 2^k$, $f = (X^n + 1)$, $n = 2^m$ (Saber)
- **Third option:** $q = 2^k$, $f = \Phi_{n+1}$, $n + 1$ prime (Round5)
- **Fourth option:** q prime, $f = (X^n + 1) = \Phi_{2n}$, $n = 2^m$ (NewHope, Kyber, LAC)
- **Fifth option:** q prime, $f = (X^n - X - 1)$ irreducible, n prime (NTRU Prime)
- **Sixth option:** ThreeBears works on large integers instead of polynomials

Design space 1: What ring?

- Structured lattice-based schemes use ring $\mathcal{R}_q = \mathbb{Z}_q[X]/f$
  - q typically either prime or a power of two
  - f typically of degree between 512 and 1024
- **First option:** $q = 2^k$, $f = (X^n - 1)$, n prime (NTRU)
- **Second option:** $q = 2^k$, $f = (X^n + 1)$, $n = 2^m$ (Saber)
- **Third option:** $q = 2^k$, $f = \Phi_{n+1}$, $n + 1$ prime (Round5)
- **Fourth option:** q prime, $f = (X^n + 1) = \Phi_{2n}$, $n = 2^m$ (NewHope, Kyber, LAC)
- **Fifth option:** q prime, $f = (X^n - X - 1)$ irreducible, n prime (NTRU Prime)
- **Sixth option:** ThreeBears works on large integers instead of polynomials
- No proof that any option is more or less secure

Design space 1: What ring?

- Structured lattice-based schemes use ring $\mathcal{R}_q = \mathbb{Z}_q[X]/f$
  - q typically either prime or a power of two
  - f typically of degree between 512 and 1024
- **First option:** $q = 2^k$, $f = (X^n - 1)$, n prime (NTRU)
- **Second option:** $q = 2^k$, $f = (X^n + 1)$, $n = 2^m$ (Saber)
- **Third option:** $q = 2^k$, $f = \Phi_{n+1}$, $n + 1$ prime (Round5)
- **Fourth option:** q prime, $f = (X^n + 1) = \Phi_{2n}$, $n = 2^m$ (NewHope, Kyber, LAC)
- **Fifth option:** q prime, $f = (X^n - X - 1)$ irreducible, n prime (NTRU Prime)
- **Sixth option:** ThreeBears works on large integers instead of polynomials
- No proof that any option is more or less secure
- NTRU Prime advertises "less structure" in their $\mathcal{R}_q$

Design space 1: What ring?

- Structured lattice-based schemes use ring $\mathcal{R}_q = \mathbb{Z}_q[X]/f$
  - q typically either prime or a power of two
  - f typically of degree between 512 and 1024
- **First option:** $q = 2^k$, $f = (X^n - 1)$, n prime (NTRU)
- **Second option:** $q = 2^k$, $f = (X^n + 1)$, $n = 2^m$ (Saber)
- **Third option:** $q = 2^k$, $f = \Phi_{n+1}$, $n + 1$ prime (Round5)
- **Fourth option:** q prime, $f = (X^n + 1) = \Phi_{2n}$, $n = 2^m$ (NewHope, Kyber, LAC)
- **Fifth option:** q prime, $f = (X^n - X - 1)$ irreducible, n prime (NTRU Prime)
- **Sixth option:** ThreeBears works on large integers instead of polynomials
- No proof that any option is more or less secure
- NTRU Prime advertises “less structure” in their $\mathcal{R}_q$
- NewHope and Kyber have fastest (NTT-based) arithmetic

Design space 2: module vs. ring?

- "Traditionally", work directly with elements of $\mathcal{R}_q$ ("Ring-LWE")
- Alternative: Module-LWE (MLWE):
  - Choose smaller n, e.g., $n = 256$ (Kyber, Saber, ThreeBears)
  - Work with small-dimension matrices and vectors over $\mathcal{R}_q$

Design space 2: module vs. ring?

- “Traditionally”, work directly with elements of $\mathcal{R}_q$ (“Ring-LWE”)
- Alternative: Module-LWE (MLWE):
  - Choose smaller n, e.g., $n = 256$ (Kyber, Saber, ThreeBears)
  - Work with small-dimension matrices and vectors over $\mathcal{R}_q$
- MLWE encrypts shorter messages than Ring-LWE

Design space 2: module vs. ring?

- "Traditionally", work directly with elements of $\mathcal{R}_q$ ("Ring-LWE")
- Alternative: Module-LWE (MLWE):
    - Choose smaller n, e.g., $n = 256$ (Kyber, Saber, ThreeBears)
    - Work with small-dimension matrices and vectors over $\mathcal{R}_q$
- MLWE encrypts shorter messages than Ring-LWE
- MLWE eliminates some of the structure of Ring-LWE

Design space 2: module vs. ring?

- "Traditionally", work directly with elements of $\mathcal{R}_q$ ("Ring-LWE")
- Alternative: Module-LWE (MLWE):
  - Choose smaller n, e.g., $n = 256$ (Kyber, Saber, ThreeBears)
  - Work with small-dimension matrices and vectors over $\mathcal{R}_q$
- MLWE encrypts shorter messages than Ring-LWE
- MLWE eliminates some of the structure of Ring-LWE
- MLWE can very easily scale security (change dimension of matrix):
  - Optimize arithmetic in $\mathcal{R}_q$ once
  - Use same optimized $\mathcal{R}_q$ arithmetic for all security levels

Design space 3: what noise?

- Need to sample noise (for LWE schemes) and small secrets
- More noise means
  - more security from the underlying hard problem
  - higher failure probability of decryption

Design space 3: what noise?

- Need to sample noise (for LWE schemes) and small secrets
- More noise means
  - more security from the underlying hard problem
  - higher failure probability of decryption
- Three main choices to make:
  - **Narrow or wide noise**
    - Narrow noise (e.g., in $\{-1, 0, 1\}$) not conservative
    - Wide noise requires larger q (or more failures)
    - Larger q means larger public key and ciphertext

Design space 3: what noise?

- Need to sample noise (for LWE schemes) and small secrets
- More noise means
  - more security from the underlying hard problem
  - higher failure probability of decryption
- Three main choices to make:
  - **Narrow or wide noise**
    - Narrow noise (e.g., in $\{-1, 0, 1\}$) not conservative
    - Wide noise requires larger q (or more failures)
    - Larger q means larger public key and ciphertext
  - **LWE or LWR**
    - LWE considered more conservative (independent noise)
    - LWR easier to implement (no noise sampling)
    - LWR allows more compact public key and ciphertext

Design space 3: what noise?

- Need to sample noise (for LWE schemes) and small secrets
- More noise means
  - more security from the underlying hard problem
  - higher failure probability of decryption
- Three main choices to make:
  - **Narrow or wide noise**
    - Narrow noise (e.g., in $\{-1, 0, 1\}$) not conservative
    - Wide noise requires larger q (or more failures)
    - Larger q means larger public key and ciphertext
  - **LWE or LWR**
    - LWE considered more conservative (independent noise)
    - LWR easier to implement (no noise sampling)
    - LWR allows more compact public key and ciphertext
  - **Fixed-weight noise or not?**
    - Fixed-weight noise needs random permutation (sorting)
    - Naive implementations leak secrets through timing
    - Advantage of fixed-weight: easier to bound (or eliminate) decryption failures

Design space 4: allow failures?

- Can avoid decryption failures entirely (NTRU, NTRU Prime)
- Advantage:
  - Easier CCA security transform and analysis
- Disadvantage:
  - Need to limit noise (or have larger q)

Design space 4: allow failures?

- Can avoid decryption failures entirely (NTRU, NTRU Prime)
- Advantage:
  - Easier CCA security transform and analysis
- Disadvantage:
  - Need to limit noise (or have larger q)
- For passive-security-only can go the other way:
  - Allow failure probability of, e.g., 2^{-30}
  - Reduce size of public key and ciphertext

Design space 4: allow failures?

- Can avoid decryption failures entirely (NTRU, NTRU Prime)
- Advantage:
  - Easier CCA security transform and analysis
- Disadvantage:
  - Need to limit noise (or have larger q)
- For passive-security-only can go the other way:
  - Allow failure probability of, e.g., 2^{-30}
  - Reduce size of public key and ciphertext
- Active (CCA) security needs negligible failure prob.

Design space 5: public parameters?

- "Traditional" approach to choosing $\mathbf{a}$ in LWE/LWR schemes:

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

Design space 5: public parameters?

- "Traditional" approach to choosing **a** in LWE/LWR schemes:

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

- Before NewHope: *real-world* approach: generate fixed **a** once

Design space 5: public parameters?

- "Traditional" approach to choosing **a** in LWE/LWR schemes:

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

- Before NewHope: *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!)

Design space 5: public parameters?

- "Traditional" approach to choosing **a** in LWE/LWR schemes:

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

- Before NewHope: *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

Design space 5: public parameters?

- "Traditional" approach to choosing **a** in LWE/LWR schemes:

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

- Before NewHope: *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: Choose a fresh **a** every time
- Server can cache **a** for some time (e.g., 1h)
- All NIST PQC candidates now use this approach

Design space 6: error-correcting codes?

- Ring-LWE/LWR schemes work with polynomials of > 256 coefficients
- "Encrypt" messages of > 256 bits
- **Need to encrypt** only 256-bit key
- Question: How do we put those additional bits to use?
- Answer: Use error-correcting code (ECC) to reduce failure probability

Design space 6: error-correcting codes?

- Ring-LWE/LWR schemes work with polynomials of > 256 coefficients
- "Encrypt" messages of > 256 bits
- **Need to encrypt** only 256-bit key
- Question: How do we put those additional bits to use?
- Answer: Use error-correcting code (ECC) to reduce failure probability
- NewHope: very simple threshold decoding

Design space 6: error-correcting codes?

- Ring-LWE/LWR schemes work with polynomials of > 256 coefficients
- "Encrypt" messages of > 256 bits
- **Need to encrypt** only 256-bit key
- Question: How do we put those additional bits to use?
- Answer: Use error-correcting code (ECC) to reduce failure probability
- NewHope: very simple threshold decoding
- LAC, Round5: more advanced ECC
  - Correct more error, obtain smaller public key and ciphertext
  - More complex to implement, in particular without leaking through timing

Design space 7: CCA security?

- Ephemeral key exchange does not need CCA security
- Can offer passively secure version
- Protocols will combine this with signatures for authentication

Design space 7: CCA security?

- Ephemeral key exchange does not need CCA security
- Can offer passively secure version
- Protocols will combine this with signatures for authentication
- **Advantages:**
  - Higher failure probability $\rightarrow$ more compact
  - Simpler to implement, no CCA transform

Design space 7: CCA security?

- Ephemeral key exchange does not need CCA security
- Can offer passively secure version
- Protocols will combine this with signatures for authentication
- **Advantages:**
  - Higher failure probability $\rightarrow$ more compact
  - Simpler to implement, no CCA transform
- **Disadvantages:**
  - Less robust (will somebody reuse keys?)
  - More options (CCA vs. CPA): easier to make mistakes

Design space 8: CCA transforms

- General Fujisaki-Okamoto principle is the same for most KEMs (exception: NTRU)
- Tweaks to FO transform:
  - Hash public-key into coins: multitarget protection (for non-zero failure probability)

Design space 8: CCA transforms

- General Fujisaki-Okamoto principle is the same for most KEMs (exception: NTRU)
- Tweaks to FO transform:
  - Hash public-key into coins: multitarget protection (for non-zero failure probability)
  - Hash public-key into shared key: KEM becomes contributory

Design space 8: CCA transforms

- General Fujisaki-Okamoto principle is the same for most KEMs (exception: NTRU)
- Tweaks to FO transform:
  - Hash public-key into coins: multitarget protection (for non-zero failure probability)
  - Hash public-key into shared key: KEM becomes contributory
  - Hash ciphertext into shared key: more robust (?)

Design space 8: CCA transforms

- General Fujisaki-Okamoto principle is the same for most KEMs (exception: NTRU)
- Tweaks to FO transform:
  - Hash public-key into coins: multitarget protection (for non-zero failure probability)
  - Hash public-key into shared key: KEM becomes contributory
  - Hash ciphertext into shared key: more robust (?)
- How to handle rejection?
  - Return special symbol (`return -1`): explicit
  - Return $\mathsf{H}(s, C)$ for secret s: implicit

Design space 8: CCA transforms

- General Fujisaki-Okamoto principle is the same for most KEMs (exception: NTRU)
- Tweaks to FO transform:
  - Hash public-key into coins: multitarget protection (for non-zero failure probability)
  - Hash public-key into shared key: KEM becomes contributory
  - Hash ciphertext into shared key: more robust (?)
- How to handle rejection?
  - Return special symbol (`return -1`): explicit
  - Return $\mathrm{H}(s, C)$ for secret s: implicit
- As of round 2, no proposal uses explicit rejection
  - Would break some security reduction
  - More robust in practice (return value alwas 0)

Implementing Lattice-based KEMs

(on embedded microcontrollers)

- Joint work with
  Matthias Kannwischer, Joost Rijneveld, and Ko Stoffelen.
- Started as part of PQCRYPTO H2020 project
- Continued within EPOQUE ERC StG
- Library and testing/benchmarking framework
  - PQ-crypto on ARM Cortex-M4
  - Uses STM32F4 Discovery board
  - 192 KB of RAM, benchmarks at 24 MHz
- Easy to add schemes using NIST API
- Optimized SHA3 and AES shared across primitives

pqm4 usage

- Run functional tests of all primitives and implementations:

```
python3 test.py
```

pqm4 usage

- Run functional tests of all primitives and implementations:

```
python3 test.py
```

- Generate testvectors, compare for consistency (also with host):

```
python3 testvectors.py
```

pqm4 usage

- Run functional tests of all primitives and implementations:

```
python3 test.py
```

- Generate testvectors, compare for consistency (also with host):

```
python3 testvectors.py
```

- Run speed and stack benchmarks:

```
python3 benchmarks.py
```

pqm4 usage

- Run functional tests of all primitives and implementations:

```
python3 test.py
```

- Generate testvectors, compare for consistency (also with host):

```
python3 testvectors.py
```

- Run speed and stack benchmarks:

```
python3 benchmarks.py
```

- Easy to evaluate only subset of schemes, e.g.:

```
python3 test.py newhope1024cca sphincs-shake256-128s
```

Core operation: multiplication in $\mathcal{R}_q = \mathbb{Z}_q[X]/f$

Power-of-two q

- Several schemes use $q = 2^m$, for small m
- Examples: Round5, NTRU, Saber
- More round-1 examples: Kindi, RLizard

Core operation: multiplication in $\mathcal{R}_q = \mathbb{Z}_q[X]/f$

Power-of-two q

- Several schemes use $q = 2^m$, for small m
- Examples: Round5, NTRU, Saber
- More round-1 examples: Kindi, RLizard

Prime "NTT-friendly" q

- Kyber and NewHope use prime q supporting fast NTT
- For $A, B \in \mathcal{R}_q$, $A \cdot B = \mathrm{NTT}^{-1}(\mathrm{NTT}(A) \circ \mathrm{NTT}(B))$
- NTT is Fourier Transform over finite field
- Use $f = X^n + 1$ for power-of-two n

Multiplication in $\mathbb{Z}_{2^m}[X]$

- Joint work with **Matthias Kannwischer** and **Joost Rijneveld**
- Represent coefficients as 16-bit integers
- No modular reductions required, 2^{16} is a multiple of $q = 2^m$

Multiplication in $\mathbb{Z}_{2^m}[X]$

- Joint work with **Matthias Kannwischer** and **Joost Rijneveld**
- Represent coefficients as 16-bit integers
- No modular reductions required, 2^{16} is a multiple of $q = 2^m$
- Schoolbook multiplication takes n^2 integer muls, $(n-1)^2$ adds

Multiplication in $\mathbb{Z}_{2^m}[X]$

- Joint work with **Matthias Kannwischer** and **Joost Rijneveld**
- Represent coefficients as 16-bit integers
- No modular reductions required, 2^{16} is a multiple of $q = 2^m$
- Schoolbook multiplication takes n^2 integer muls, $(n-1)^2$ adds
- Can do better using Karatsuba:

$$\begin{aligned}
&(a_\ell + X^k a_h) \cdot (b_\ell + X^k b_h) \\
= &\; a_\ell b_\ell + X^k(a_\ell b_h + a_h b_\ell) + X^n a_h b_h \\
= &\; a_\ell b_\ell + X^k((a_\ell + a_h)(b_\ell + b_h) - a_\ell b_\ell - a_h b_h) + X^n a_h b_h
\end{aligned}$$

- Recursive application yields complexity $\Theta(n^{\log_2 3})$

Multiplication in $\mathbb{Z}_{2^m}[X]$

- Joint work with **Matthias Kannwischer** and **Joost Rijneveld**
- Represent coefficients as 16-bit integers
- No modular reductions required, 2^{16} is a multiple of $q = 2^m$
- Schoolbook multiplication takes n^2 integer muls, $(n-1)^2$ adds
- Can do better using Karatsuba:

$$
\begin{aligned}
&(a_\ell + X^k a_h) \cdot (b_\ell + X^k b_h) \\
= \; &a_\ell b_\ell + X^k(a_\ell b_h + a_h b_\ell) + X^n a_h b_h \\
= \; &a_\ell b_\ell + X^k((a_\ell + a_h)(b_\ell + b_h) - a_\ell b_\ell - a_h b_h) + X^n a_h b_h
\end{aligned}
$$

- Recursive application yields complexity $\Theta(n^{\log_2 3})$
- Generalization: Toom-Cook
  - Toom-3: split into 5 multiplications of 1/3 size
  - Toom-4: split into 7 multiplications of 1/4 size
- Approach: Evaluate, multiply, interpolate

Initial observations

- Karatsuba/Toom is asymptotically faster, but isn't for "small" polynomials

Initial observations

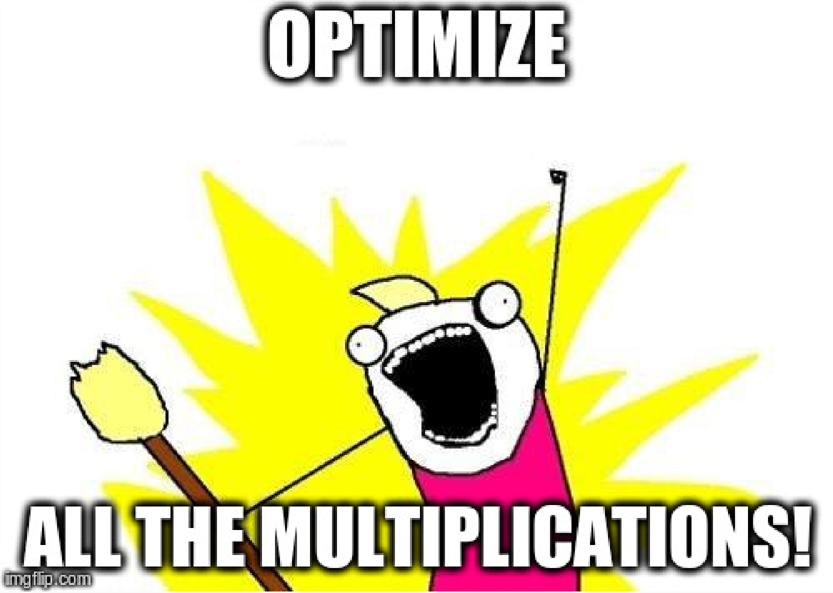
- Karatsuba/Toom is asymptotically faster, but isn't for "small" polynomials
- Toom-3 needs division by 2, loses 1 bit of precision
- Toom-4 needs division by 8, loses 3 bits of precision
- This limits recursive application when using 16-bit integers
- Can use Toom-4 only for $q \leq 2^{13}$

Initial observations

- Karatsuba/Toom is asymptotically faster, but isn't for "small" polynomials
- Toom-3 needs division by 2, loses 1 bit of precision
- Toom-4 needs division by 8, loses 3 bits of precision
- This limits recursive application when using 16-bit integers
- Can use Toom-4 only for $q \leq 2^{13}$
- Karmakar, Bermudo Mera, Sinha Roy, Verbauwhede (CHES 2018):
  - Optimize Saber, $q = 2^{13}, n = 256$
  - Use Toom-4 + two levels of Karatsuba
  - Optimized 16-coefficient schoolbook multiplication

Initial observations

- Karatsuba/Toom is asymptotically faster, but isn't for "small" polynomials
- Toom-3 needs division by 2, loses 1 bit of precision
- Toom-4 needs division by 8, loses 3 bits of precision
- This limits recursive application when using 16-bit integers
- Can use Toom-4 only for $q \leq 2^{13}$
- Karmakar, Bermudo Mera, Sinha Roy, Verbauwhede (CHES 2018):
  - Optimize Saber, $q = 2^{13}, n = 256$
  - Use Toom-4 + two levels of Karatsuba
  - Optimized 16-coefficient schoolbook multiplication
- **Is this the best approach? How about other values of q and n?**


OPTIMIZE
ALL THE MULTIPLICATIONS!
imgflip.com

Our approach

- Generate optimized assembly for Karatsuba/Toom
- Use Python scripts, receive as input n and q
- Hand-optimize "small" schoolbook multiplications
  - Make heavy use of "vector instructions"
  - Perform two 16×16-bit multiply-accumulate in one cycle
  - Carefully schedule instructions to minimize loads/stores
- Benchmark different options, pick fastest

Multiplication results

	approach	"small"	cycles	stack
Saber ($n = 256$, $q = 2^{13}$)	Karatsuba only	16	41 121	2 020
	Toom-3	11	41 225	3 480
	Toom-4	**16**	**39 124**	**3 800**
	Toom-4 + Toom-3	-	-	-
Kindi-256-3-4-2 ($n = 256$, $q = 2^{14}$)	**Karatsuba only**	**16**	**41 121**	**2 020**
	Toom-3	11	41 225	3 480
	Toom-4	-	-	-
	Toom-4 + Toom-3	-	-	-
NTRU-HRSS ($n = 701$, $q = 2^{13}$)	Karatsuba only	11	230 132	5 676
	Toom-3	15	217 436	9 384
	Toom-4	**11**	**182 129**	**10 596**
	Toom-4 + Toom-3	-	-	-
NTRU-KEM-743 ($n = 743$, $q = 2^{11}$)	Karatsuba only	12	247 489	6 012
	Toom-3	16	219 061	9 920
	Toom-4	**12**	**196 940**	**11 208**
	Toom-4 + Toom-3	16	197 227	12 152
RLizard-1024 ($n = 1024$, $q = 2^{11}$)	Karatsuba only	16	400 810	8 188
	Toom-3	11	360 589	13 756
	Toom-4	**16**	**313 744**	**15 344**
	Toom-4 + Toom-3	11	315 788	16 816

NTT-based multiplication

- Joint work with **Leon Botros** and **Matthias Kannwischer**
- Primary goal: optimize Kyber
- Secondary effect: optimize NewHope (with room for improvement)

NTT-based multiplication

- Joint work with **Leon Botros** and **Matthias Kannwischer**
- Primary goal: optimize Kyber
- Secondary effect: optimize NewHope (with room for improvement)
- NTT is an FFT in a finite field
- Evaluate polynomial $f = f_0 + f_1 X + \cdots + f_{n-1} X^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(X^2) + X f_1(X^2)$

NTT-based multiplication

- Joint work with **Leon Botros** and **Matthias Kannwischer**
- Primary goal: optimize Kyber
- Secondary effect: optimize NewHope (with room for improvement)
- NTT is an FFT in a finite field
- Evaluate polynomial $f = f_0 + f_1 X + \cdots + f_{n-1} X^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(X^2) + X f_1(X^2)$
  - Huge overlap between evaluating

$$f(\beta) = f_0(\beta^2) + \beta f_1(\beta^2) \text{ and}$$
$$f(-\beta) = f_0(\beta^2) - \beta f_1(\beta^2)$$

NTT-based multiplication

- Joint work with **Leon Botros** and **Matthias Kannwischer**
- Primary goal: optimize Kyber
- Secondary effect: optimize NewHope (with room for improvement)
- NTT is an FFT in a finite field
- Evaluate polynomial $f = f_0 + f_1 X + \cdots + f_{n-1} X^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(X^2) + X f_1(X^2)$
  - Huge overlap between evaluating

$$f(\beta) = f_0(\beta^2) + \beta f_1(\beta^2) \text{ and}$$
$$f(-\beta) = f_0(\beta^2) - \beta f_1(\beta^2)$$

  - f_0 has $n/2$ coefficients
  - Evaluate f_0 at all $(n/2)$-th roots of unity by recursive application
  - Same for f_1

NTT-based multiplication

- First thing to do: replace recursion by iteration
- Loop over $\log n$ levels with $n/2$ "butterflies" each

NTT-based multiplication

- First thing to do: replace recursion by iteration
- Loop over $\log n$ levels with $n/2$ "butterflies" each
- Butterfly on level k:
  - Pick up f_i and f_{i+2^k}
  - Multiply f_{i+2^k} by a power of ω to obtain t
  - Compute $f_{i+2^k} \leftarrow a_i - t$
  - Compute $f_i \leftarrow a_i + t$

NTT-based multiplication

- First thing to do: replace recursion by iteration
- Loop over $\log n$ levels with $n/2$ "butterflies" each
- Butterfly on level k:
  - Pick up f_i and f_{i+2^k}
  - Multiply f_{i+2^k} by a power of ω to obtain t
  - Compute $f_{i+2^k} \leftarrow a_i - t$
  - Compute $f_i \leftarrow a_i + t$
- Main optimizations on Cortex-M4:
  - "Merge" levels: fewer loads/stores
  - Optimize modular arithmetic (precompute powers of ω in Montgomery domain)
  - Lazy reductions
  - Carefully optimize using DSP instructions

Selected optimized lattice KEM cycles

Scheme	Key Generation	Encapsulation	Decapsulation
ntruhps2048509	77 698 713	645 329	542 439
ntruhps2048677	144 383 491	955 902	836 959
ntruhps4096821	211 758 452	1 205 662	1 066 879
ntruhrss701	154 676 705	402 784	890 231
lightsaber	459 965	651 273	678 810
saber	896 035	1 161 849	1 204 633
firesaber	1 448 776	1 786 930	1 853 339
kyber512	514 291	652 769	621 245
kyber768	976 757	1 146 556	1 094 849
kyber1024	1 575 052	1 779 848	1 709 348
newhope1024cpa	975 736	975 452	162 660
newhope1024cca	1 161 112	1 777 918	1 760 470

Comparison: Curve25519 scalarmult: 625 358 cycles

Selected optimized lattice KEM stack bytes

Scheme	Key Generation	Encapsulation	Decapsulation
ntruhps2048509	21 412	15 452	14 828
ntruhps2048677	28 524	20 604	19 756
ntruhps4096821	34 532	24 924	23 980
ntruhrss701	27 580	19 372	20 580
lightsaber	9 656	11 392	12 136
saber	13 256	15 544	16 640
firesaber	20 144	23 008	24 592
kyber512	2 952	2 552	2 560
kyber768	3 848	3 128	3 072
kyber1024	4 360	3 584	3 592
newhope1024cpa	11 096	17 288	8 308
newhope1024cca	11 080	17 360	19 576

Resources online

- Overview NIST round-2 candidates:
  `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/round-2-submissions`
- pqm4 library and benchmarking suite:
  `https://github.com/mupq/pqm4`
- Code of $\mathbb{Z}_{2^m}[x]$ multiplication paper, including scripts:
  `https://github.com/mupq/polymul-z2mx-m4`
- $\mathbb{Z}_{2^m}[x]$ multiplication paper:
  `https://cryptojedi.org/papers/#latticem4`
- Kyber optimization paper:
  `https://cryptojedi.org/papers/#nttm4`