Radboud University

On implementation issues of post-quantum cryptography

Peter Schwabe

peter@cryptojedi.org

https://cryptojedi.org

June 13, 2019

The NIST competition

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**

4 31 27

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

"Key exchange"

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

The NIST competition (ctd.)

“Key exchange”

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

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- Several broken, 5 withdrawn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

NIST reference and "optimized" implementations

" Two implementations are required in the submission package: a reference implementation and an optimized implementation.

[...]

Both implementations shall consist of source code written in ANSI C"

NIST reference and "optimized" implementations

" Two implementations are required in the submission package: a reference implementation and an optimized implementation.

[...]

Both implementations shall consist of source code written in ANSI C"

- Allowed to use some third-party libraries:
  - NTL Version 10.5.0
  - GMP Version 6.1.2
  - OpenSSL
  - Keccak Code package

NIST reference and “optimized” implementations

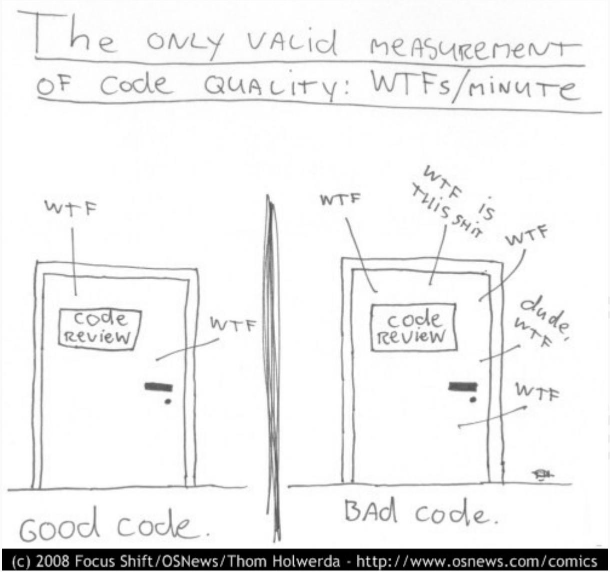
“ Two implementations are required in the submission package: a reference implementation and an optimized implementation.

[...]

Both implementations shall consist of source code written in ANSI C”

- Allowed to use some third-party libraries:
  - NTL Version 10.5.0
  - GMP Version 6.1.2
  - OpenSSL
  - Keccak Code package
- *Not* allowed to use intrinsics or assembly
- Can include additional (e.g., architecture-specific) implementations

Code quality



PQClean

- Joint work with
  Matthias Kannwischer, Joost Rijneveld, Douglas Stebila, Thom Wiggers
- GitHub repo with extensive CI to ensure "clean" implementations

PQClean

- Joint work with
  Matthias Kannwischer, Joost Rijneveld, Douglas Stebila, Thom Wiggers
- GitHub repo with extensive CI to ensure "clean" implementations
- Goal: collect "clean C" code of all round-2 candidates
- Make it easy to use in other projects
- Make it easy to use as starting point for optimization

PQClean

- Joint work with
  Matthias Kannwischer, Joost Rijneveld, Douglas Stebila, Thom Wiggers
- GitHub repo with extensive CI to ensure "clean" implementations
- Goal: collect "clean C" code of all round-2 candidates
- Make it easy to use in other projects
- Make it easy to use as starting point for optimization
- Longer-term, if there is interest:
  - implementations with architecture-specific optimizations?
  - implementations in other languages?

The definition of “clean”

- Code is valid C99
- Passes functional tests
- API functions do not write outside provided buffers
- API functions do not need pointers to be aligned
- Compiles with -Wall -Wextra -Wpedantic -Werror with gcc and clang
- Compiles with `/W4 /WX` with MS compiler
- Consistent test vectors across runs
- Consistent test vectors on big-endian and little-endian machines
- Consistent test vectors on 32-bit and 64-bit machines

The definition of “clean”

- No errors/warnings reported by valgrind
- No errors/warnings reported by address sanitizer
- No errors/warnings reported by undefined-behavior sanitizer
- Only dependencies:
  - `fips202.c`
  - `sha2.c`
  - `aes.c`
  - `randombytes.c`

The definition of “clean”

- API functions return 0 on success, negative on failure
- No dynamic memory allocations

The definition of “clean”

- API functions return 0 on success, negative on failure
- No dynamic memory allocations
- Builds under Linux, MacOS, and Windows without warnings
- All exported symbols are namespaced with `PQCLEAN_SCHEMENAME_`
- Each implementation comes with license and meta information in `META.yml`

The definition of “clean” – the controversial bits

- No variable-length arrays (required to build under Windows)

The definition of “clean” – the controversial bits

- No variable-length arrays (required to build under Windows)
- Separate subdirectories (without symlinks) for each parameter set of each scheme

The definition of “clean” – the controversial bits

- No variable-length arrays (required to build under Windows)
- Separate subdirectories (without symlinks) for each parameter set of each scheme
- `#ifdefs` only for header encapsulation

The definition of "clean" – the controversial bits

- No variable-length arrays (required to build under Windows)
- Separate subdirectories (without symlinks) for each parameter set of each scheme
- `#ifdefs` only for header encapsulation
- No stringification macros

The definition of “clean” – the controversial bits

- No variable-length arrays (required to build under Windows)
- Separate subdirectories (without symlinks) for each parameter set of each scheme
- `#ifdefs` only for header encapsulation
- No stringification macros
- Dealing with controversial warnings (unary minus on unsigned integers)

The definition of "clean" – the controversial bits

- No variable-length arrays (required to build under Windows)
- Separate subdirectories (without symlinks) for each parameter set of each scheme
- `#ifdefs` only for header encapsulation
- No stringification macros
- Dealing with controversial warnings (unary minus on unsigned integers)
- Argument names consistent between `.h` and `.c` files

Limitations and lessons learned

- MS compiler does not support C99 $\rightarrow$ no variable-length arrays

Limitations and lessons learned

- MS compiler does not support C99 $\rightarrow$ no variable-length arrays
- Public CI services impose serious limitations through timeouts

Limitations and lessons learned

- MS compiler does not support C99 → no variable-length arrays
- Public CI services impose serious limitations through timeouts
- Not yet testing for "constant-time" behavior
  - Could use valgrind with uninitialized secret data (dynamic)
  - Alternative: `ct-verif` (static)

Limitations and lessons learned

- MS compiler does not support C99 $\rightarrow$ no variable-length arrays
- Public CI services impose serious limitations through timeouts
- Not yet testing for "constant-time" behavior
  - Could use valgrind with uninitialized secret data (dynamic)
  - Alternative: `ct-verif` (static)
  - Tricky to even find the right definition(s)

Limitations and lessons learned

- MS compiler does not support C99 $\rightarrow$ no variable-length arrays
- Public CI services impose serious limitations through timeouts
- Not yet testing for "constant-time" behavior
  - Could use valgrind with uninitialized secret data (dynamic)
  - Alternative: `ct-verif` (static)
  - Tricky to even find the right definition(s)
- Valgrind does not work with environments running on qemu

PQClean status quo – Signatures

CRYSTALS-Dilithium	✓
FALCON	—
GeMSS	—
LUOV	WIP
MQDSS	✓
Picnic	—
qTESLA	—
Rainbow	WIP
SPHINCS+	✓

PQClean status quo – KEMs

BIKE	—
Classic McEliece	WIP
CRYSTALS-Kyber	✓
Frodo-KEM	✓
HQC	—
LAC	—
LEDAcrypt	WIP
NewHope	✓
NTRU	✓
NTRU Prime	WIP
NTS-KEM	—
ROLLO	—
Round5	—
RQC	—
SABER	—
SIKE	—
ThreeBears	WIP

Using code from PQClean

- Copy files from origin directory

Using code from PQClean

- Copy files from origin directory
- Instantiate SHA-3, SHA-2, AES (or copy from PQClean)

Using code from PQClean

- Copy files from origin directory
- Instantiate SHA-3, SHA-2, AES (or copy from PQClean)
- Add `.c` and `.h` files to build system

- 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

- 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
```

Signatures (not) in `pqm4`

CRYSTALS-Dilithium	✓
FALCON	—
GeMSS	✗
LUOV	✓
MQDSS	✓
Picnic	✗
qTESLA	✓
Rainbow	✗
SPHINCS+	✓

KEMs (not) in pqm4

	ref/clean	opt
BIKE	—	—
Classic McEliece	✗	✗
CRYSTALS-Kyber	✓	✓
Frodo-KEM	✓	(✓)
HQC	—	—
LAC	✓	—
LEDAcrypt	WIP	WIP
NewHope	✓	✓
NTRU	✓	✓
NTRU Prime	✓	—
NTS-KEM	✗	✗
ROLLO	—	—
Round5	WIP	WIP
RQC	—	—
SABER	✓	✓
SIKE	—	—
ThreeBears	✓	✓

KEMs (not) in pqm4

	ref/clean	**opt**
BIKE	—	—
Classic McEliece	✗	✗
CRYSTALS-Kyber	✓	✓
Frodo-KEM	✓	(✓)
HQC	—	—
LAC	✓	—
LEDAcrypt	WIP	WIP
NewHope	✓	✓
NTRU	✓	✓
NTRU Prime	✓	—
NTS-KEM	✗	✗
ROLLO	—	—
Round5	WIP	WIP
RQC	—	—
SABER	✓	✓
SIKE	—	—
ThreeBears	✓	✓

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 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
- Structured lattices: work in $\mathbb{Z}_q[x]/f$

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
- Structured lattices: work in $\mathbb{Z}_q[x]/f$

Lattice-based KEMs – the basic idea

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

$$\begin{array}{llll} \text{Alice has} & \mathbf{v} & = \mathbf{us} & = \mathbf{ass}' + \mathbf{e}'\mathbf{s} \\ \text{Bob has} & \mathbf{v}' & = \mathbf{bs}' & = \mathbf{ass}' + \mathbf{es}' \end{array}$$

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

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

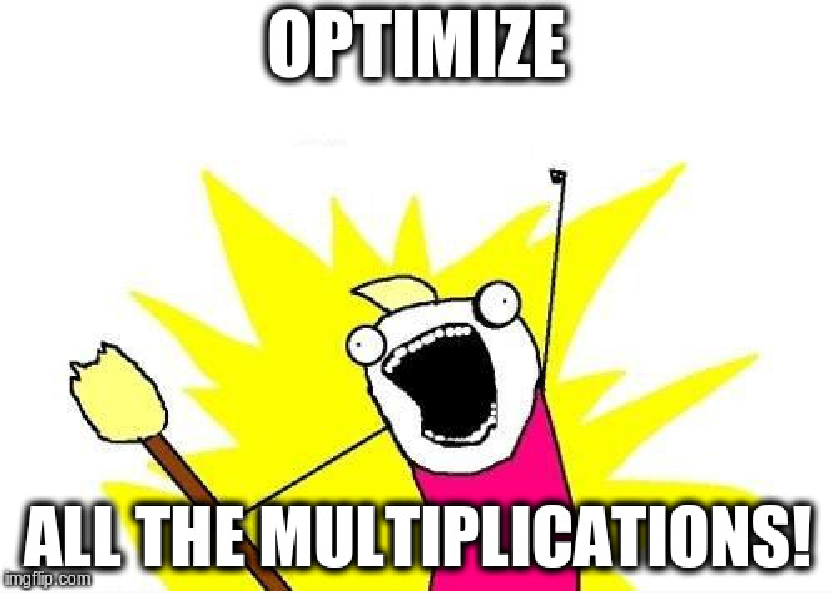
- Karatsuba/Toom is asymptotically faster, but isn't for "small" polynomials
- Toom-3 needs division by 2, loses 1 bit of precision
- Toom-4 needs division by 8, loses 3 bits of precision
- This limits recursive application when using 16-bit integers
- Can use Toom-4 only for $q \leq 2^{13}$

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

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	1 034 955	1 495 457	206 112
newhope1024cca	1 219 908	1 903 231	1 927 505

Comparison: Curve25519 scalarmult: 625 358 cycles

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	11 128	17 288	8 328
newhope1024cca	11 152	17 400	19 640

Conclusions and open questions

- Speed-bottleneck of lattice-based KEMs is Keccak
- Long-term solution: hardware acceleration for Keccak

Conclusions and open questions

- Speed-bottleneck of lattice-based KEMs is Keccak
- Long-term solution: hardware acceleration for Keccak
- Much more work to be done on code-based KEMs

Conclusions and open questions

- Speed-bottleneck of lattice-based KEMs is Keccak
- Long-term solution: hardware acceleration for Keccak
- Much more work to be done on code-based KEMs
- So far very little work on SCA *protection*
- Start with "constant-time" software for all candidates

Conclusions and open questions

- Speed-bottleneck of lattice-based KEMs is Keccak
- Long-term solution: hardware acceleration for Keccak
- Much more work to be done on code-based KEMs
- So far very little work on SCA *protection*
- Start with "constant-time" software for all candidates
- Formally verify constant-time behavior? Definition?
- Would be great to have **hacspec** implementations of all NIST candidates

Resources online

- PQClean repository:
  `https://github.com/PQClean/PQClean`
- `pqm4` library and benchmarking suite:
  `https://github.com/mupq/pqm4`
- `pqriscv` library and benchmarking suite:
  `https://github.com/mupq/pqriscv`
- 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`