On implementation issues of post-quantum cryptography

Peter Schwabe
peter@cryptojedi.org
https://cryptojedi.org
June 13, 2019
The NIST competition

<table>
<thead>
<tr>
<th>Row Labels</th>
<th>Key Exchange</th>
<th>Signature</th>
<th>Grand Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Braids</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Chebychev</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Codes</td>
<td>19</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>Finite Automata</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Hash</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Hypercomplex Numbers</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Isogeny</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Lattice</td>
<td>24</td>
<td>4</td>
<td>28</td>
</tr>
<tr>
<td>Mult. Var</td>
<td>6</td>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td>Rand. walk</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RSA</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><strong>Grand Total</strong></td>
<td><strong>57</strong></td>
<td><strong>23</strong></td>
<td><strong>80</strong></td>
</tr>
</tbody>
</table>

Overview tweeted by Jacob Alperin-Sheriff on Dec 4, 2017.
“Key exchange”

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

The NIST competition (ctd.)

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- Several broken, 5 withdrawn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
“Key exchange”

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

**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- Several broken, 5 withdrawn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
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”
“Two implementations are required in the submission package: a reference implementation and an optimized implementation.

[...]
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
The only valid measurement of code quality: WTFs/minute

(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics
• Joint work with
  Matthias Kannwischer, Joost Rijneveld, Douglas Stebila, Thom Wiggers
• GitHub repo with extensive CI to ensure “clean” implementations

5
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
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_SCHMENAME_
- 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)
• 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 → no variable-length arrays
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)
- Tricky to even find the right definition(s)
- Valgrind does not work with environments running on qemu
• 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)
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)
  - Tricky to even find the right definition(s)
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)
  - Tricky to even find the right definition(s)
- Valgrind does not work with environments running on qemu
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYS-TALS-Dilithium</td>
<td>✓</td>
</tr>
<tr>
<td>FALCON</td>
<td>—</td>
</tr>
<tr>
<td>GeMSS</td>
<td>—</td>
</tr>
<tr>
<td>LUOV</td>
<td>WIP</td>
</tr>
<tr>
<td>MQDSS</td>
<td>✓</td>
</tr>
<tr>
<td>Picnic</td>
<td>—</td>
</tr>
<tr>
<td>qTESLA</td>
<td>—</td>
</tr>
<tr>
<td>Rainbow</td>
<td>WIP</td>
</tr>
<tr>
<td>SPHINCS+</td>
<td>✓</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Status</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>BIKE</td>
<td>—</td>
</tr>
<tr>
<td>Classic McEliece</td>
<td>WIP</td>
</tr>
<tr>
<td>CRYSYS-Kyber</td>
<td>✓</td>
</tr>
<tr>
<td>Frodo-KEM</td>
<td>✓</td>
</tr>
<tr>
<td>HQC</td>
<td>—</td>
</tr>
<tr>
<td>LAC</td>
<td>—</td>
</tr>
<tr>
<td>LEDAcrypt</td>
<td>WIP</td>
</tr>
<tr>
<td>NewHope</td>
<td>✓</td>
</tr>
<tr>
<td>NTRU</td>
<td>✓</td>
</tr>
<tr>
<td>NTRU Prime</td>
<td>WIP</td>
</tr>
<tr>
<td>NTS-KEM</td>
<td>—</td>
</tr>
<tr>
<td>ROLLO</td>
<td>—</td>
</tr>
<tr>
<td>Round5</td>
<td>—</td>
</tr>
<tr>
<td>RQC</td>
<td>—</td>
</tr>
<tr>
<td>SABER</td>
<td>—</td>
</tr>
<tr>
<td>SIKE</td>
<td>—</td>
</tr>
<tr>
<td>ThreeBears</td>
<td>WIP</td>
</tr>
</tbody>
</table>
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:

```python
python3 test.py
```
pqm4 usage

- Run functional tests of all primitives and implementations:
  
  \texttt{python3 test.py}

- Generate testvectors, compare for consistency (also with host):
  
  \texttt{python3 testvectors.py}
- 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
<table>
<thead>
<tr>
<th>Signature Scheme</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYSSTALS-Dilithium</td>
<td>✓</td>
</tr>
<tr>
<td>FALCON</td>
<td></td>
</tr>
<tr>
<td>GeMSS</td>
<td>✗</td>
</tr>
<tr>
<td>LUOV</td>
<td>✓</td>
</tr>
<tr>
<td>MQDSS</td>
<td>✓</td>
</tr>
<tr>
<td>Picnic</td>
<td>✗</td>
</tr>
<tr>
<td>qTESLA</td>
<td>✓</td>
</tr>
<tr>
<td>Rainbow</td>
<td>✗</td>
</tr>
<tr>
<td>SPHINCS+</td>
<td>✓</td>
</tr>
</tbody>
</table>
KEMs *(not)* in pqm4

<table>
<thead>
<tr>
<th></th>
<th>ref/clean</th>
<th>opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIKE</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Classic McEliece</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>CRYSTALS-Kyber</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Frodo-KEM</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>HQC</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>LAC</td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>LEDAcrypt</td>
<td>WIP</td>
<td>WIP</td>
</tr>
<tr>
<td>NewHope</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NTRU</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NTRU Prime</td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>NTS-KEM</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>ROLLO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Round5</td>
<td>WIP</td>
<td>WIP</td>
</tr>
<tr>
<td>RQC</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>SABER</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SIKE</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>ThreeBears</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Algorithm</td>
<td>ref/clean</td>
<td>opt</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------</td>
<td>-----</td>
</tr>
<tr>
<td>BIKE</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Classic McEliece</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><strong>CRYSTALS-Kyber</strong></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Frodo-KEM</td>
<td>✓</td>
<td>(✓)</td>
</tr>
<tr>
<td>HQC</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>LAC</strong></td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>LEDAcrypt</td>
<td>WIP</td>
<td>WIP</td>
</tr>
<tr>
<td><strong>NewHope</strong></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>NTRU</strong></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>NTRU Prime</strong></td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>NTS-KEM</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>ROLLO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>Round5</strong></td>
<td>WIP</td>
<td>WIP</td>
</tr>
<tr>
<td>RQC</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>SABER</strong></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SIKE</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>ThreeBears</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Learning with errors (LWE)

- Given uniform $A \in \mathbb{Z}_q^{k \times \ell}$
- Given "noise distribution" $\chi$
- Given samples $As + e$, with $e \leftarrow \chi$
Learning with errors (LWE)

- Given uniform $A \in \mathbb{Z}_q^{k \times \ell}$
- Given “noise distribution” $\chi$
- Given samples $As + e$, with $e \leftarrow \chi$
- Search version: find $s$
- Decision version: distinguish from uniform random
Learning with errors (LWE)

- Given uniform $A \in \mathbb{Z}_q^{k \times \ell}$
- Given “noise distribution” $\chi$
- Given samples $As + e$, with $e \leftarrow \chi$
- Search version: find $s$
- Decision version: distinguish from uniform random
- Structured lattices: work in $\mathbb{Z}_q[x]/f$
Learning with rounding (LWR)

• Given uniform $A \in \mathbb{Z}_q^{k \times \ell}$
• Given samples $\lceil A_s \rceil_p$, with $p < q$
Learning with rounding (LWR)

- Given uniform $A \in \mathbb{Z}_q^{k \times \ell}$
- Given samples $\left\lfloor A s \right\rfloor_p$, with $p < q$
- Search version: find $s$
- 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)
--- | ---
\( s, e \leftarrow \chi \) | \( s', e' \leftarrow \chi \)
\( b \leftarrow as + e \) | \( b \)
\( u \leftarrow as' + e' \) | \( u \)

Alice has \( v = us = ass' + e's \)
Bob has \( v' = bs' = ass' + es' \)

- Secret and noise \( s, s', e, e' \) are small
- \( v \) and \( v' \) 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 $A, B \in \mathcal{R}_q$, $A \cdot B = \text{NTT}^{-1}(\text{NTT}(A) \circ \text{NTT}(B))$
- 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 $q = 2^m$
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
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:

\[
(a_\ell + X^k a_h) \cdot (b_\ell + X^k b_h) \\
= a_\ell b_\ell + X^k(a_\ell b_h + a_h b_\ell) + X^n a_h b_h \\
= a_\ell b_\ell + X^k((a_\ell + a_h)(b_\ell + b_h) - a_\ell b_\ell - a_h b_h) + X^n a_h b_h
\]

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:
  \[
  (a_\ell + X^k a_h) \cdot (b_\ell + X^k b_h) \\
  = a_\ell b_\ell + X^k(a_\ell b_h + a_h b_\ell) + X^n a_h b_h \\
  = a_\ell b_\ell + X^k((a_\ell + a_h)(b_\ell + b_h) - a_\ell b_\ell - a_h b_h) + X^n a_h b_h
  \]
- 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

- 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

- 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 $n$?
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!
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 $16 \times 16$-bit multiply-accumulate in one cycle
  - Carefully schedule instructions to minimize loads/stores
- Benchmark different options, pick fastest
<table>
<thead>
<tr>
<th>Multiplication results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Saber</td>
</tr>
<tr>
<td>$n = 256$, $q = 2^{13}$</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Kindi-256-3-4-2</td>
</tr>
<tr>
<td>$n = 256$, $q = 2^{14}$</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>NTRU-HRSS</td>
</tr>
<tr>
<td>$n = 701$, $q = 2^{13}$</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>NTRU-KEM-743</td>
</tr>
<tr>
<td>$n = 743$, $q = 2^{11}$</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>RLizard-1024</td>
</tr>
<tr>
<td>$n = 1024$, $q = 2^{11}$</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
NTT-based multiplication

- Joint work with Leon Botros and Matthias Kannwischer
- Primary goal: optimize Kyber
- Secondary effect: optimize NewHope (with room for improvement)
• 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 \( f = f_0 + f_1X + \cdots + f_{n-1}X^{n-1} \) at all \( n \)-th roots of unity
• Divide-and-conquer approach
  • Write polynomial \( f \) as \( f_0(X^2) + Xf_1(X^2) \)
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 \( f_0(X^2) + Xf_1(X^2) \)
  - Huge overlap between evaluating
    \[
    f(\beta) = f_0(\beta^2) + \beta f_1(\beta^2) \quad \text{and} \quad f(-\beta) = f_0(\beta^2) - \beta f_1(\beta^2)
    \]
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 \( f = f_0 + f_1X + \cdots + f_{n-1}X^{n-1} \) at all \( n \)-th roots of unity
- Divide-and-conquer approach
  - Write polynomial \( f \) as \( f_0(X^2) + Xf_1(X^2) \)
  - Huge overlap between evaluating
    \[
    f(\beta) = f_0(\beta^2) + \beta f_1(\beta^2) \quad \text{and} \quad f(-\beta) = f_0(\beta^2) - \beta f_1(\beta^2)
    \]
  - \( f_0 \) 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 \( \omega \) 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 $\omega$ 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 $\omega$ in Montgomery domain)
  - Lazy reductions
  - Carefully optimize using DSP instructions
### Optimized lattice KEM cycles

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Key Generation</th>
<th>Encapsulation</th>
<th>Decapsulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ntruhps2048509</td>
<td>77 698 713</td>
<td>645 329</td>
<td>542 439</td>
</tr>
<tr>
<td>ntruhps2048677</td>
<td>144 383 491</td>
<td>955 902</td>
<td>836 959</td>
</tr>
<tr>
<td>ntruhps4096821</td>
<td>211 758 452</td>
<td>1 205 662</td>
<td>1 066 879</td>
</tr>
<tr>
<td>ntruhrss701</td>
<td>154 676 705</td>
<td>402 784</td>
<td>890 231</td>
</tr>
<tr>
<td>lightsaber</td>
<td>459 965</td>
<td>651 273</td>
<td>678 810</td>
</tr>
<tr>
<td>saber</td>
<td>896 035</td>
<td>1 161 849</td>
<td>1 204 633</td>
</tr>
<tr>
<td>firesaber</td>
<td>1 448 776</td>
<td>1 786 930</td>
<td>1 853 339</td>
</tr>
<tr>
<td>kyber512</td>
<td>514 291</td>
<td>652 769</td>
<td>621 245</td>
</tr>
<tr>
<td>kyber768</td>
<td>976 757</td>
<td>1 146 556</td>
<td>1 094 849</td>
</tr>
<tr>
<td>kyber1024</td>
<td>1 575 052</td>
<td>1 779 848</td>
<td>1 709 348</td>
</tr>
<tr>
<td>newhope1024cpa</td>
<td>1 034 955</td>
<td>1 495 457</td>
<td>206 112</td>
</tr>
<tr>
<td>newhope1024cca</td>
<td>1 219 908</td>
<td>1 903 231</td>
<td>1 927 505</td>
</tr>
</tbody>
</table>

**Comparison:** Curve25519 scalarmult: 625 358 cycles
<table>
<thead>
<tr>
<th>Scheme</th>
<th>Key Generation</th>
<th>Encapsulation</th>
<th>Decapsulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ntruhs2048509</td>
<td>21 412</td>
<td>15 452</td>
<td>14 828</td>
</tr>
<tr>
<td>ntruhs2048677</td>
<td>28 524</td>
<td>20 604</td>
<td>19 756</td>
</tr>
<tr>
<td>ntruhs4096821</td>
<td>34 532</td>
<td>24 924</td>
<td>23 980</td>
</tr>
<tr>
<td>ntruhrss701</td>
<td>27 580</td>
<td>19 372</td>
<td>20 580</td>
</tr>
<tr>
<td>lightsaber</td>
<td>9 656</td>
<td>11 392</td>
<td>12 136</td>
</tr>
<tr>
<td>saber</td>
<td>13 256</td>
<td>15 544</td>
<td>16 640</td>
</tr>
<tr>
<td>firesaber</td>
<td>20 144</td>
<td>23 008</td>
<td>24 592</td>
</tr>
<tr>
<td>kyber512</td>
<td>2 952</td>
<td>2 552</td>
<td>2 560</td>
</tr>
<tr>
<td>kyber768</td>
<td>3 848</td>
<td>3 128</td>
<td>3 072</td>
</tr>
<tr>
<td>kyber1024</td>
<td>4 360</td>
<td>3 584</td>
<td>3 592</td>
</tr>
<tr>
<td>newhope1024cpa</td>
<td>11 128</td>
<td>17 288</td>
<td>8 328</td>
</tr>
<tr>
<td>newhope1024cca</td>
<td>11 152</td>
<td>17 400</td>
<td>19 640</td>
</tr>
</tbody>
</table>
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