Post-quantum crypto on ARM Cortex-M

Peter Schwabe
peter@cryptojedi.org
https://cryptojedi.org
January 23, 2019
• Project funded by EU in Horizon 2020.
• Running from March 2015 until February 2018
• 11 partners from academia and industry, TU/e was coordinator
• 22 submissions to NIST PQC project
Find post-quantum secure cryptosystems suitable for small devices in power and memory requirements (e.g. smart cards with 8-bit or 16-bit or 32-bit architectures, with different amounts of RAM)

Develop efficient implementations of these systems.
• Find post-quantum secure cryptosystems suitable for small devices in power and memory requirements (e.g. smart cards with 8-bit or 16-bit or 32-bit architectures, with different amounts of RAM)

• Develop efficient implementations of these systems.

• Main challenge: memory, e.g.,
  - McEliece (code-based encryption): \(\approx 1\) MB public key
  - GUI (MQ-based signatures) \(\approx 2\) MB public key
  - SPHINCS\^+: 8–50 KB signatures
• Find post-quantum secure cryptosystems suitable for small devices in power and memory requirements (e.g. smart cards with 8-bit or 16-bit or 32-bit architectures, with different amounts of RAM)

• Develop efficient implementations of these systems.

• Main challenge: memory, e.g.,
  • McEliece (code-based encryption): $\approx 1$ MB public key
  • GUI (MQ-based signatures) $\approx 2$ MB public key
  • SPHINCS$^+$: 8–50 KB signatures

• Additional challenges:
  • Computational complexity
  • Implementation security
Primary target platform

- ARM Cortex-M4 on STM32F4-Discovery board
- 192KB RAM, 1MB Flash (ROM)
- Available for <20 Euros from various vendors (e.g., Amazon, RS Components, Conrad)
• Joint work with
  Matthias Kannwischer, Joost Rijneveld, and Ko Stoffelen.
• Library and testing/benchmarking framework
• Easy to add schemes using NIST API
• Optimized SHA3 shared across primitives
• Joint work with
  **Matthias Kannwischer, Joost Rijneveld, and Ko Stoffelen.**
• Library and testing/benchmarking framework
• Easy to add schemes using NIST API
• Optimized SHA3 shared across primitives
• Run functional tests of all primitives and implementations:
  ```python
  python3 test.py
  ```
• Generate testvectors, compare for consistency (also with host):
  ```python
  python3 testvectors.py
  ```
• Run speed and stack benchmarks:
  ```python
  python3 benchmarks.py
  ```
• Easy to evaluate only subset of schemes, e.g.:
  ```python
  python3 test.py newhope1024cca sphincs-shake256-128s
  ```
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIG QUAKE</td>
<td>?</td>
</tr>
<tr>
<td>BIKE</td>
<td>?</td>
</tr>
<tr>
<td>Classic McEliece</td>
<td>✗</td>
</tr>
<tr>
<td>CRYSTALS-Kyber</td>
<td>✓</td>
</tr>
<tr>
<td>DAGS</td>
<td>?</td>
</tr>
<tr>
<td>FrodoKEM</td>
<td>✓</td>
</tr>
<tr>
<td>KINDI</td>
<td>✓</td>
</tr>
<tr>
<td>NewHope</td>
<td>✓</td>
</tr>
<tr>
<td>NTRU-HRSS-KEM</td>
<td>✓</td>
</tr>
<tr>
<td>NTRU Prime</td>
<td>✓</td>
</tr>
<tr>
<td>Post-quantum RSA-Encryption</td>
<td>✗</td>
</tr>
<tr>
<td>Ramstake</td>
<td>✗(?)</td>
</tr>
<tr>
<td>SABER</td>
<td>✓</td>
</tr>
<tr>
<td>SIKE</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Initial pqm4 results signatures

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYSRALS-Dilithium</td>
<td>✔️</td>
</tr>
<tr>
<td>GUI</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>Post-quantum RSA-Signature</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>
From PQCRYPTO to EPOQUE

- Since October 2018 working on ERC project
  \textit{Engineering post-quantum cryptography – EPOQUE}
- WP1: Secure implementations of post-quantum crypto
- Build on results of PQCRYPTO, e.g., extend \texttt{pqm4}:
  - Include more optimized implementations
  - Include implementations with SCA protection

First paper of EPOQUE:
Matthias Kannwicher, Joost Rijneveld, Peter Schwabe.
Faster multiplication in $\mathbb{Z}_{2^m}$ on Cortex-M4 to speed up NIST PQC candidates.

Speed up 5 lattice-based KEMs
From PQCRYPTO to EPOQUE

• Since October 2018 working on ERC project 

   Engineering post-quantum cryptography – EPOQUE

• WP1: Secure implementations of post-quantum crypto

• Build on results of PQCRYPTO, e.g., extend pqm4:
  
  • Include more optimized implementations
  
  • Include implementations with SCA protection

• First paper of EPOQUE:

  Matthias Kannwischer, Joost Rijneveld, Peter Schwabe. Faster
  multiplication in \( \mathbb{Z}_{2^m}[x] \) on Cortex-M4 to speed up NIST PQC
  candidates.

• Speed up 5 lattice-based KEMs
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 $p < q$
Given uniform $A \in \mathbb{Z}_q^{k \times \ell}$

Given samples $\lceil As \rceil_p$, 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

<table>
<thead>
<tr>
<th>Alice (server)</th>
<th>Bob (client)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s, e \leftarrow \chi )</td>
<td>( s', e' \leftarrow \chi )</td>
</tr>
<tr>
<td>( b \leftarrow as + e )</td>
<td>( b \rightarrow u )</td>
</tr>
<tr>
<td>( u \leftarrow as' + e' )</td>
<td>( u \leftarrow as' + e' )</td>
</tr>
</tbody>
</table>

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

Bob has \( v' = bs' = ass' + es' \)

- Secret and noise \( s, s', e, e' \) are small
- \( t \) and \( t' \) are *approximately* the same
Lattice-based KEMs submitted to NIST

- 22 NIST submissions are lattice-based KEMs
- Large design space with many tradeoffs:
• 22 NIST submissions are lattice-based KEMs
• Large design space with many tradeoffs:
  • LWE vs. LWR
  • LWE vs. Ring-LWE vs. Module-LWE
• 22 NIST submissions are lattice-based KEMs
• Large design space with many tradeoffs:
  • LWE vs. LWR
  • LWE vs. Ring-LWE vs. Module-LWE
  • Prime $q$ vs. power-of-two $q$
• 22 NIST submissions are lattice-based KEMs
• Large design space with many tradeoffs:
  • LWE vs. LWR
  • LWE vs. Ring-LWE vs. Module-LWE
  • Prime $q$ vs. power-of-two $q$
  • Prime $n$ vs. power-of-two $n$
Lattice-based KEMs submitted to NIST

- 22 NIST submissions are lattice-based KEMs
- Large design space with many tradeoffs:
  - LWE vs. LWR
  - LWE vs. Ring-LWE vs. Module-LWE
  - Prime $q$ vs. power-of-two $q$
  - Prime $n$ vs. power-of-two $n$
  - NTRU vs. LPR ("quotient" vs. "product")
• 22 NIST submissions are lattice-based KEMs
• Large design space with many tradeoffs:
  • LWE vs. LWR
  • LWE vs. Ring-LWE vs. Module-LWE
  • Prime $q$ vs. power-of-two $q$
  • Prime $n$ vs. power-of-two $n$
  • NTRU vs. LPR ("quotient" vs. "product")
  • “Encryption-based” vs. “Reconciliation based”
Lattice-based KEMs submitted to NIST

- 22 NIST submissions are lattice-based KEMs
- Large design space with many tradeoffs:
  - LWE vs. LWR
  - LWE vs. Ring-LWE vs. Module-LWE
  - Prime $q$ vs. power-of-two $q$
  - Prime $n$ vs. power-of-two $n$
  - NTRU vs. LPR ("quotient" vs. "product")
  - "Encryption-based" vs. "Reconciliation based"
  - Decryption failures vs. no failures
Lattice-based KEMs submitted to NIST

- 22 NIST submissions are lattice-based KEMs
- Large design space with many tradeoffs:
  - LWE vs. LWR
  - LWE vs. Ring-LWE vs. Module-LWE
  - Prime $q$ vs. power-of-two $q$
  - Prime $n$ vs. power-of-two $n$
  - NTRU vs. LPR ("quotient" vs. "product")
  - "Encryption-based" vs. "Reconciliation based"
  - Decryption failures vs. no failures
  - Passive vs. active security
  - ...
Lattice-based KEMs submitted to NIST

- 22 NIST submissions are lattice-based KEMs
- Large design space with many tradeoffs:
  - LWE vs. LWR
  - LWE vs. Ring-LWE vs. Module-LWE
  - Prime $q$ vs. power-of-two $q$
  - Prime $n$ vs. power-of-two $n$
  - NTRU vs. LPR ("quotient" vs. "product")
  - "Encryption-based" vs. "Reconciliation based"
  - Decryption failures vs. no failures
  - Passive vs. active security
  - ...
5 lattice-based KEMs

- RLizard, Saber, NTRU-HRSS, NTRUEncrypt, and Kindi
- All rely on arithmetic in $\mathbb{Z}_{2^m}[x]/f$
  - $11 \leq m \leq 14$
  - $256 \leq n = \deg(f) \leq 1024$
5 lattice-based KEMs

- RLizard, Saber, NTRU-HRSS, NTRUEncrypt, and Kindi
- All rely on arithmetic in $\mathbb{Z}_{2^m}[x]/f$
  - $11 \leq m \leq 14$
  - $256 \leq n = \deg(f) \leq 1024$
- Why optimize those 5 KEMs?
  - Have to start somewhere... 
  - Joost and I are co-submitters of NTRU-HRSS 
  - It seemed like NTRU-HRSS could be faster than Round5 
  - Only Saber has been optimized on Cortex-M4 before (CHES 2018)
5 lattice-based KEMs

- RLizard, Saber, NTRU-HRSS, NTRUEncrypt, and Kindi
- All rely on arithmetic in $\mathbb{Z}_{2^m}[x]/f$
  - $11 \leq m \leq 14$
  - $256 \leq n = \deg(f) \leq 1024$
- Why optimize those 5 KEMs?
  - Have to start somewhere...
  - Joost and I are co-submitters of NTRU-HRSS
  - It seemed like NTRU-HRSS could be faster than Round5
  - Only Saber has been optimized on Cortex-M4 before (CHES 2018)
- How to optimize those 5 KEMs?
  - Faster multiplication of polynomials with $n$ coefficients over $\mathbb{Z}_{2^m}[x]$
Polynomial multiplication

- Represent coefficients as 16-bit integers
- No modular reductions required, $2^{16}$ is a multiple of $q = 2^m$
Polynomial multiplication

- 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
Polynomial multiplication

- 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:

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

- Recursive application yields complexity \(\Theta(n^{\log_3 3})\)
Polynomial multiplication

- 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
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 $n$?
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
- Benchmark different options, pick fastest
Fast schoolbook multiplication

- ARMv7E-M supports SMUAD(X) and SMLAD(X)
- All in one clock cycle
- Perfect for polynomial multiplication

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>smuad Ra, Rb, Rc</td>
<td>$Ra \leftarrow Rb_L \cdot Rc_L + Rb_H \cdot Rc_H$</td>
</tr>
<tr>
<td>smuadx Ra, Rb, Rc</td>
<td>$Ra \leftarrow Rb_L \cdot Rc_H + Rb_H \cdot Rc_L$</td>
</tr>
<tr>
<td>smlad Ra, Rb, Rc, Rd</td>
<td>$Ra \leftarrow Rb_L \cdot Rc_L + Rb_H \cdot Rc_H + Rd$</td>
</tr>
<tr>
<td>smladx Ra, Rb, Rc, Rd</td>
<td>$Ra \leftarrow Rb_L \cdot Rc_H + Rb_H \cdot Rc_L + Rd$</td>
</tr>
</tbody>
</table>

Slide credit to Matthias Kannwischer
Fast schoolbook multiplication [N=2]

Slide credit to Matthias Kannwischer
Fast schoolbook multiplication $[N=2]$

- 3 multiplications instead of 4

Slide credit to Matthias Kannwischer
Fast schoolbook multiplication [N=4]

Slide credit to Matthias Kannwischer
Fast schoolbook multiplication \([N=4]\)

- 10 multiplications instead of 16

Slide credit to Matthias Kannwischer
Fast schoolbook multiplication \([N=6]\)

Slide credit to Matthias Kannwischer
Fast schoolbook multiplication \( [N=6] \)

- 21 multiplications instead of 36

Slide credit to Matthias Kannwischer
Fast schoolbook multiplication \([N=12]\)

- How many can we fit in registers?
- 16 registers minus SP and PC \(\rightarrow\) we fit 24 coefficients

Slide credit to Matthias Kannwischer
Fast schoolbook multiplication [N=12]

- How many can we fit in registers?
- 16 registers minus SP and PC → we fit 24 coefficients
- 78 multiplications instead of 144

Slide credit to Matthias Kannwischer
Fast schoolbook multiplication \([N=24]\)

Slide credit to Matthias Kannwischer
• We want to merge all, but not enough registers

Slide credit to Matthias Kannwischer
• Instead we perform 4 times 12x12

Slide credit to Matthias Kannwischer
Fast schoolbook multiplication \([N=36]\)

- Or 9 times 12x12

Slide credit to Matthias Kannwischer
Fast schoolbook multiplication: Reduce repacks

- $R0 = a_1|a_0$, $R1 = a_3|a_2$, $R2 = a_5|a_4$
- $R3 = b_1|b_0$, $R4 = b_3|b_2$, $R5 = b_5|b_4$

Slide credit to Matthias Kannwischer
Fast schoolbook multiplication: Reduce repacks

- $R0 = a_1 | a_0$, $R1 = a_3 | a_2$, $R2 = a_5 | a_4$
- $R3 = b_1 | b_0$, $R4 = b_3 | b_2$, $R5 = b_5 | b_4$
- For even columns we need to repack $b$

Slide credit to Matthias Kannwischer
Fast schoolbook multiplication: Reduce repacks

- \( R0 = a_1|a_0, R1 = a_3|a_2, R2 = a_5|a_4 \)
- \( R3 = b_1|b_0, R4 = b_3|b_2, R5 = b_5|b_4 \)
- First do odd columns

Slide credit to Matthias Kannwischer
Fast schoolbook multiplication: Reduce repacks

- $R_0 = a_1|a_0$, $R_1 = a_3|a_2$, $R_2 = a_5|a_4$
- Then repack to $R_3 = b_2|b_1$, $R_4 = b_4|b_3$ and do even columns

Slide credit to Matthias Kannwischer
## Multiplication results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Approach</th>
<th>“small”</th>
<th>Cycles</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saber</td>
<td>Karatsuba only</td>
<td>16</td>
<td>41 121</td>
<td>2 020</td>
</tr>
<tr>
<td>($n = 256, q = 2^{13}$)</td>
<td>Toom-3</td>
<td>11</td>
<td>41 225</td>
<td>3 480</td>
</tr>
<tr>
<td></td>
<td><strong>Toom-4</strong></td>
<td>16</td>
<td>39 124</td>
<td>3 800</td>
</tr>
<tr>
<td></td>
<td>Toom-4 + Toom-3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Kindi-256-3-4-2</td>
<td>Karatsuba only</td>
<td>16</td>
<td>41 121</td>
<td>2 020</td>
</tr>
<tr>
<td>($n = 256, q = 2^{14}$)</td>
<td>Toom-3</td>
<td>11</td>
<td>41 225</td>
<td>3 480</td>
</tr>
<tr>
<td></td>
<td>Toom-4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Toom-4 + Toom-3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NTRU-HRSS</td>
<td>Karatsuba only</td>
<td>11</td>
<td>230 132</td>
<td>5 676</td>
</tr>
<tr>
<td>($n = 701, q = 2^{13}$)</td>
<td>Toom-3</td>
<td>15</td>
<td>217 436</td>
<td>9 384</td>
</tr>
<tr>
<td></td>
<td><strong>Toom-4</strong></td>
<td>11</td>
<td>182 129</td>
<td>10 596</td>
</tr>
<tr>
<td></td>
<td>Toom-4 + Toom-3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NTRU-KEM-743</td>
<td>Karatsuba only</td>
<td>12</td>
<td>247 489</td>
<td>6 012</td>
</tr>
<tr>
<td>($n = 743, q = 2^{11}$)</td>
<td>Toom-3</td>
<td>16</td>
<td>219 061</td>
<td>9 920</td>
</tr>
<tr>
<td></td>
<td><strong>Toom-4</strong></td>
<td>12</td>
<td>196 940</td>
<td>11 208</td>
</tr>
<tr>
<td></td>
<td>Toom-4 + Toom-3</td>
<td>16</td>
<td>197 227</td>
<td>12 152</td>
</tr>
<tr>
<td>RLiizard-1024</td>
<td>Karatsuba only</td>
<td>16</td>
<td>400 810</td>
<td>8 188</td>
</tr>
<tr>
<td>($n = 1024, q = 2^{11}$)</td>
<td>Toom-3</td>
<td>11</td>
<td>360 589</td>
<td>13 756</td>
</tr>
<tr>
<td></td>
<td><strong>Toom-4</strong></td>
<td>16</td>
<td>313 744</td>
<td>15 344</td>
</tr>
<tr>
<td></td>
<td>Toom-4 + Toom-3</td>
<td>11</td>
<td>315 788</td>
<td>16 816</td>
</tr>
</tbody>
</table>
Anything else to do?

- Integrate with fast SHA-3/SHAKE implementation
- Add fast SHA-512 implementation (C as fast as asm!)
- Between 69% and 92% of cycles spent in mul+hash
Anything else to do?

- Integrate with fast SHA-3/SHAKE implementation
- Add fast SHA-512 implementation (C as fast as asm!)
- Between 69% and 92% of cycles spent in mul+hash

**NISTPQC code quality...**

- Fix misunderstandings of NIST API
- Remove all dynamic memory allocations
- Fix some obvious timing leakages
- **More work required, for many NIST submissions!**
<table>
<thead>
<tr>
<th>Implementation</th>
<th>Clock Cycles</th>
<th>Stack Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Saber</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>K: 6 530k</td>
<td>K: 12 616k</td>
</tr>
<tr>
<td></td>
<td>E: 8 684k</td>
<td>E: 14 896k</td>
</tr>
<tr>
<td></td>
<td>D: 10 581k</td>
<td>D: 15 992k</td>
</tr>
<tr>
<td>[KBSV18]</td>
<td>K: 1 147k</td>
<td>K: 13 883k</td>
</tr>
<tr>
<td></td>
<td>E: 1 444k</td>
<td>E: 16 667k</td>
</tr>
<tr>
<td></td>
<td>D: 1 543k</td>
<td>D: 17 763k</td>
</tr>
<tr>
<td>This work</td>
<td>K: 949k</td>
<td>K: 13 248k</td>
</tr>
<tr>
<td></td>
<td>E: 1 232k</td>
<td>E: 15 528k</td>
</tr>
<tr>
<td></td>
<td>D: 1 260k</td>
<td>D: 16 624k</td>
</tr>
<tr>
<td><strong>Kindi-256-3-4-2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>K: 21 794k</td>
<td>K: 59 864k</td>
</tr>
<tr>
<td></td>
<td>E: 28 176k</td>
<td>E: 71 000k</td>
</tr>
<tr>
<td></td>
<td>D: 37 129k</td>
<td>D: 84 096k</td>
</tr>
<tr>
<td>This work</td>
<td>K: 1 010k</td>
<td>K: 44 264k</td>
</tr>
<tr>
<td></td>
<td>E: 1 365k</td>
<td>E: 55 392k</td>
</tr>
<tr>
<td></td>
<td>D: 1 563k</td>
<td>D: 64 376k</td>
</tr>
</tbody>
</table>
## KEM results

<table>
<thead>
<tr>
<th></th>
<th>implementation</th>
<th>clock cycles</th>
<th>stack usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NTRU-HRSS</strong></td>
<td>Reference</td>
<td>K: 205 156k</td>
<td>K: 10 020</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E: 5 166k</td>
<td>E: 8 956</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D: 15 067k</td>
<td>D: 10 204</td>
</tr>
<tr>
<td></td>
<td>This work</td>
<td>K: 161 790k</td>
<td>K: 23 396</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E: 432k</td>
<td>E: 19 492</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D: 863k</td>
<td>D: 22 140</td>
</tr>
<tr>
<td><strong>NTRU-KEM-743</strong></td>
<td>Reference</td>
<td>K: 59 815k</td>
<td>K: 14 148</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E: 7 540k</td>
<td>E: 13 372</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D: 14 229k</td>
<td>D: 18 036</td>
</tr>
<tr>
<td></td>
<td>This work</td>
<td>K: 5 663k</td>
<td>K: 25 320</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E: 1 655k</td>
<td>E: 23 808</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D: 1 904k</td>
<td>D: 28 472</td>
</tr>
<tr>
<td><strong>RLizard-1024</strong></td>
<td>Reference</td>
<td>K: 26 423k</td>
<td>K: 4 272</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E: 32 156k</td>
<td>E: 10 532</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D: 53 181k</td>
<td>D: 12 636</td>
</tr>
<tr>
<td></td>
<td>This work</td>
<td>K: 537k</td>
<td>K: 27 720</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E: 1 358k</td>
<td>E: 33 328</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D: 1 740k</td>
<td>D: 35 448</td>
</tr>
</tbody>
</table>
• Great about NISTPQC: we actually have implementations!
Coming back to NISTPQC code quality

- Great about NISTPQC: we actually have implementations!
- Bad about NISTPQC: we have lots of terrible implementations...
Coming back to NISTPQC code quality

- Great about NISTPQC: we actually have implementations!
- Bad about NISTPQC: we have lots of terrible implementations...
- Typical effort for *any* project working with NISTPQC code:
  1. Clean up existing implementation
  2. Do what you actually want to do
Coming back to NISTPQC code quality

- Great about NISTPQC: we actually have implementations!
- Bad about NISTPQC: we have lots of terrible implementations...
- Typical effort for *any* project working with NISTPQC code:
  1. Clean up existing implementation
  2. Do what you actually want to do
- Examples of what you actually want to do:
  - Use in libraries (e.g., liboqs or libpqcrypto)
  - Benchmark (e.g., SUPERCOP)
  - Evaluate on embedded platforms (e.g., pqm4)
  - Use in higher-level protocols (e.g., OQS)
• Great about NISTPQC: we actually have implementations!
• Bad about NISTPQC: we have lots of terrible implementations. . .
• Typical effort for any project working with NISTPQC code:
  1. Clean up existing implementation
  2. Do what you actually want to do
• Examples of what you actually want to do:
  • Use in libraries (e.g., liboqs or libpqcrypto)
  • Benchmark (e.g., SUPERCOP)
  • Evaluate on embedded platforms (e.g., pqm4)
  • Use in higher-level protocols (e.g., OQS)
• Idea: collect “clean” implementations once
● Joint work with
  Matthias Kannwischer, Joost Rijneveld, Douglas Stebila, Thom Wiggers
● GitHub repo with extensive CI to ensure “clean” implementations
Joint work with

Matthias Kannwischer, Joost Rijneveld, Douglas Stebila, Thom Wiggers

GitHub repo with extensive CI to ensure “clean” implementations

Goal: eventually have all round-2 candidates in there

Start with clean C implementations
PQClean

• Joint work with
  Matthias Kannwischer, Joost Rijneveld, Douglas Stebila, Thom Wiggers
• GitHub repo with extensive CI to ensure “clean” implementations
• Goal: eventually have all round-2 candidates in there
• Start with clean C implementations
• Longer-term, if there is interest:
  • implementations with architecture-specific optimizations?
  • implementations in other languages?

At the moment still setting up CI
• Hope to be done soon, then PRs very welcome!
• Joint work with
  Matthias Kannwischer, Joost Rijneveld, Douglas Stebila, Thom Wiggers
• GitHub repo with extensive CI to ensure “clean” implementations
• Goal: eventually have all round-2 candidates in there
• Start with clean C implementations
• Longer-term, if there is interest:
  • implementations with architecture-specific optimizations?
  • implementations in other languages?
• At the moment still setting up CI
• Hope to be done soon, then PRs very welcome!
The definition of “clean”

Automatically checked by CI

- Code is valid C99
- Passes functional tests
- API functions do not write outside provided buffers
- Compiles with -Wall -Wextra -Wpedantic -Werror with gcc and clang
- 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

- No errors/warnings reported by valgrind
- No errors/warnings reported by address sanitizer
- Only dependencies:
  - fips202.c
  - sha2.c
  - aes.c
  - randombytes.c
The definition of “clean”

Automatically checked by CI

- Code is valid C99
- Passes functional tests
- API functions do not write outside provided buffers
- Compiles with -Wall -Wextra -Wpedantic -Werror with gcc and clang
- 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
- No errors/warnings reported by valgrind
- No errors/warnings reported by address sanitizer
- Only dependencies:
  - fips202.c
  - sha2.c
  - aes.c
  - randombytes.c
Automatically checked by CI

- API functions return 0 on success, negative on failure (WIP!)
  - 0 on success
  - Negative on failure (currently: partially)
- No dynamic memory allocations
- No branching on secret data (dynamically checked using valgrind)
- No access to secret memory locations (dynamically checked using valgrind)
The definition of “clean” ctd.

Automatically checked by CI

- API functions return 0 on success, negative on failure (WIP!)
  - 0 on success
  - Negative on failure (currently: partially)
- No dynamic memory allocations
- No branching on secret data (dynamically checked using valgrind)
- No access to secret memory locations (dynamically checked using valgrind)
- Separate subdirectories (without symlinks) for each parameter set of each scheme
- Builds under Linux, MacOS, and Windows
- All exported symbols are namedpaced with PQCLEAN_SCHEMENAME_
- Each implementation comes with license and meta information in META.yml
The definition of “clean”

Manually checked

- `#ifdefs` only for header encapsulation
- No stringification macros
- Output-parameter pointers in functions are on the left
- `const` arguments are labeled as `const`
- All exported symbols are namespaced in place
- All integer types are of fixed size, using `stdint.h` types (including `uint8_t` instead of unsigned char)
- Integers used for indexing are of type `size_t`
- Variable declarations at the beginning (except in `for (size_t i=...))`
• 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

• PQClean repository:  
  https://github.com/PQClean/PQClean