Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates

Matthias J. Kannwischer$^1$, Joost Rijneveld$^1$, and Peter Schwabe$^1$
January 07, 2019, Cryptography Research Inc., San Francisco

$^1$Radboud University, Nijmegen, The Netherlands
matthias@kannwischer.eu, joost@joostrijneveld.nl, peter@cryptojedi.org
Lattice-crypto is fast . . . and there is also Shor
Lattice-crypto is fast . . . and there is also Shor
NIST called for PQC – many submissions are lattice-based
Motivation

- Lattice-crypto is fast . . . and there is also Shor
- NIST called for PQC – many submissions are lattice-based
  - They all come with a C reference implementation
Lattice-crypto is fast... and there is also Shor

NIST called for PQC – many submissions are lattice-based

- They all come with a C reference implementation
- And many with AVX2 code
Motivation

- Lattice-crypto is fast ... and there is also Shor
- NIST called for PQC – many submissions are lattice-based
  - They all come with a C reference implementation
  - And many with AVX2 code
  - ... but what about small microprocessors?
Lattice-crypto is fast . . . and there is also Shor

NIST called for PQC – many submissions are lattice-based
  • They all come with a C reference implementation
  • And many with AVX2 code
  • . . . but what about small microprocessors?

PQCRYPTO – WP1: post-quantum crypto for small devices
Lattice-crypto is fast . . . and there is also Shor

NIST called for PQC – many submissions are lattice-based
  • They all come with a C reference implementation
  • And many with AVX2 code
  • . . . but what about small microprocessors?

PQCRIPTO – WP1: post-quantum crypto for small devices
  • Until March 2018
Lattice-crypto is fast . . . and there is also Shor

NIST called for PQC – many submissions are lattice-based
  • They all come with a C reference implementation
  • And many with AVX2 code
  • . . . but what about small microprocessors?

PQCRYPTO – WP1: post-quantum crypto for small devices
  • Until March 2018
  • Deliverable: PQM4 – pq library for the Cortex-M4
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)
- Benchmarking and testing framework for NISTPQC

- Currently including
  - 10 post-quantum KEMs (2 added as part of this work)
  - 3 post-quantum signatures
  - 7 optimised KEM implementations (5 added)

- We are actively working on this...
- Benchmarking and testing framework for NISTPQC
  - Automated functional tests

- Currently including
  - 10 post-quantum KEMs (2 added as part of this work)
  - 3 post-quantum signatures
  - 7 optimised KEM implementations (5 added)

- We are actively working on this...
Benchmarking and testing framework for NISTPQC

- Automated functional tests
- Automated comparison of testvectors (ref–opt, ref–host)
Benchmarking and testing framework for NISTPQC

- Automated functional tests
- Automated comparison of testvectors (ref–opt, ref–host)
- Automated benchmarking (speed and stack usage)
Benchmarking and testing framework for NISTPQC

- Automated functional tests
- Automated comparison of testvectors (ref–opt, ref–host)
- Automated benchmarking (speed and stack usage)

Currently including
- Benchmarking and testing framework for NISTPQC
  - Automated functional tests
  - Automated comparison of testvectors (ref–opt, ref–host)
  - Automated benchmarking (speed and stack usage)
- Currently including
  - 10 post-quantum KEMs (2 added as part of this work)
- Benchmarking and testing framework for NISTPQC
  - Automated functional tests
  - Automated comparison of testvectors (ref–opt, ref–host)
  - Automated benchmarking (speed and stack usage)
- Currently including
  - 10 post-quantum KEMs (2 added as part of this work)
  - 3 post-quantum signatures
- Benchmarking and testing framework for NISTPQC
  - Automated functional tests
  - Automated comparison of testvectors (ref–opt, ref–host)
  - Automated benchmarking (speed and stack usage)
- Currently including
  - 10 post-quantum KEMs (2 added as part of this work)
  - 3 post-quantum signatures
  - 7 optimised KEM implementations (5 added)
Benchmarking and testing framework for NISTPQC

- Automated functional tests
- Automated comparison of testvectors (ref–opt, ref–host)
- Automated benchmarking (speed and stack usage)

Currently including

- 10 post-quantum KEMs (2 added as part of this work)
- 3 post-quantum signatures
- 7 optimised KEM implementations (5 added)

We are actively working on this...
Benchmarking and testing framework for NISTPQC

- Automated functional tests
- Automated comparison of testvectors (ref–opt, ref–host)
- Automated benchmarking (speed and stack usage)

Currently including

- 10 post-quantum KEMs (2 added as part of this work)
- 3 post-quantum signatures
- 7 optimised KEM implementations (5 added)

We are actively working on this...

- . . . looking forward to pull requests!
Since October 2018 working on ERC project *Engineering post-quantum cryptography – EPOQUE*

WP1: Secure implementations of post-quantum crypto

Build on results of PQCRIPTO, e.g., extend pqm4:

- Include more optimized implementations
- Include implementations with SCA protection
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 $A \in \mathbb{Z}_q^{k \times l}$
- Given "noise distribution" $\chi$
- Given samples $A_s + e$, 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 $[As]_p$, with $p < q$
Learning with rounding (LWR)

- Given uniform $\mathbf{A} \in \mathbb{Z}_q^{k \times \ell}$
- Given samples $[\mathbf{A}s]_p$, 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

<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>$\rightarrow b$</td>
</tr>
<tr>
<td>$u \leftarrow as' + e'$</td>
<td>$\leftarrow u$</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
- $v$ and $v'$ are approximately the same
22 NIST submissions are lattice-based KEMs

Large design space with many tradeoffs:
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
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 $q$
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 $n$
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. LWE/LWR (“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. LWE/LWR (“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. LWE/LWR ("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. LWE/LWR ("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. LWE/LWR ("quotient" vs. "product")
  - ...
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 Peter 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 Peter 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 $\mathbb{Z}_{2^m}[x]$
Multiplying polynomials
Schoolbook multiplication \([N=1]\)

\[
a = a_0 \\
b = b_0
\]

1 multiplication, 0 additions
Schoolbook multiplication $[N=2]$

$$a = a_1x + a_0$$

$$b = b_1x + b_0$$

| $a_5b_0$ | $a_4b_0$ | $a_3b_0$ | $a_2b_0$ |
| $a_5b_1$ | $a_4b_1$ | $a_3b_1$ | $a_2b_1$ |
| $a_5b_2$ | $a_4b_2$ | $a_3b_2$ | $a_2b_2$ |
| $a_5b_3$ | $a_4b_3$ | $a_3b_3$ | $a_2b_3$ |
| $a_5b_4$ | $a_4b_4$ | $a_3b_4$ | $a_2b_4$ |
| $a_5b_5$ | $a_4b_5$ | $a_3b_5$ | $a_2b_5$ |

4 multiplications, 1 addition
Schoolbook multiplication \([N=3]\)

\[a = a_2 x^2 + a_1 x + a_0\]
\[b = b_2 x^2 + b_1 x + b_0\]

9 multiplications, 4 additions
Schoolbook multiplication \( [N=4] \)

\[
a = a_3 x^3 + 2 x^2 + a_1 x + a_0
\]

\[
b = b_3 x^3 + b_2 x^2 + b_1 x + b_0
\]

<table>
<thead>
<tr>
<th>(a_5 b_0)</th>
<th>(a_4 b_0)</th>
<th>(a_3 b_0)</th>
<th>(a_2 b_0)</th>
<th>(a_1 b_0)</th>
<th>(a_0 b_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_5 b_1)</td>
<td>(a_4 b_1)</td>
<td>(a_3 b_1)</td>
<td>(a_2 b_1)</td>
<td>(a_1 b_1)</td>
<td>(a_0 b_1)</td>
</tr>
<tr>
<td>(a_5 b_2)</td>
<td>(a_4 b_2)</td>
<td>(a_3 b_2)</td>
<td>(a_2 b_2)</td>
<td>(a_1 b_2)</td>
<td>(a_0 b_2)</td>
</tr>
<tr>
<td>(a_5 b_3)</td>
<td>(a_4 b_3)</td>
<td>(a_3 b_3)</td>
<td>(a_2 b_3)</td>
<td>(a_1 b_3)</td>
<td>(a_0 b_3)</td>
</tr>
</tbody>
</table>

16 multiplications, 9 additions
Schoolbook multiplication [N=6]

\[ a = a_5 x^5 + a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0 \]

\[ b = b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x + b_0 \]

<table>
<thead>
<tr>
<th>(a_5 b_0)</th>
<th>(a_4 b_0)</th>
<th>(a_3 b_0)</th>
<th>(a_2 b_0)</th>
<th>(a_1 b_0)</th>
<th>(a_0 b_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_5 b_1)</td>
<td>(a_4 b_1)</td>
<td>(a_3 b_1)</td>
<td>(a_2 b_1)</td>
<td>(a_1 b_1)</td>
<td>(a_0 b_1)</td>
</tr>
<tr>
<td>(a_5 b_2)</td>
<td>(a_4 b_2)</td>
<td>(a_3 b_2)</td>
<td>(a_2 b_2)</td>
<td>(a_1 b_2)</td>
<td>(a_0 b_2)</td>
</tr>
<tr>
<td>(a_5 b_3)</td>
<td>(a_4 b_3)</td>
<td>(a_3 b_3)</td>
<td>(a_2 b_3)</td>
<td>(a_1 b_3)</td>
<td>(a_0 b_3)</td>
</tr>
<tr>
<td>(a_5 b_4)</td>
<td>(a_4 b_4)</td>
<td>(a_3 b_4)</td>
<td>(a_2 b_4)</td>
<td>(a_1 b_4)</td>
<td>(a_0 b_4)</td>
</tr>
<tr>
<td>(a_5 b_5)</td>
<td>(a_4 b_5)</td>
<td>(a_3 b_5)</td>
<td>(a_2 b_5)</td>
<td>(a_1 b_5)</td>
<td>(a_0 b_5)</td>
</tr>
</tbody>
</table>

36 multiplications, 25 additions
Schoolbook multiplication has quadratic complexity
Can we do better?
Karatsuba’s method

- Split inputs in half: \( a = a_1 x^{n/2} + a_0, b = b_1 x^{n/2} + b_0 \)
Karatsuba’s method

- Split inputs in half: $a = a_1x^{n/2} + a_0, b = b_1x^{n/2} + b_0$

$$c = a \cdot b = (a_1 \cdot x^{n/2} + a_0)(b_1 \cdot x^{n/2} + b_0)$$
Karatsuba’s method

- Split inputs in half: $a = a_1 x^{n/2} + a_0$, $b = b_1 x^{n/2} + b_0$

\[ c = a \cdot b = (a_1 \cdot x^{n/2} + a_0)(b_1 \cdot x^{n/2} + b_0) \]
\[ = a_1 b_1 \cdot x^n + (a_1 b_0 + a_0 b_1) \cdot x^{n/2} + a_0 b_0 \]
Karatsuba’s method

- Split inputs in half: $a = a_1 x^{n/2} + a_0$, $b = b_1 x^{n/2} + b_0$

\[
c = a \cdot b = (a_1 \cdot x^{n/2} + a_0)(b_1 \cdot x^{n/2} + b_0)
= a_1 b_1 \cdot x^n + (a_1 b_0 + a_0 b_1) \cdot x^{n/2} + a_0 b_0
= a_1 b_1 \cdot x^n + a_0 b_0 +
\]
\[
((a_1 + a_0)(b_0 + b_1) - a_1 b_1 - a_0 b_0) \cdot x^{n/2}
\]
Karatsuba’s method

- Split inputs in half: $a = a_1 x^{n/2} + a_0, b = b_1 x^{n/2} + b_0$

$$c = a \cdot b = (a_1 \cdot x^{n/2} + a_0)(b_1 \cdot x^{n/2} + b_0)$$

$$= a_1 b_1 \cdot x^n + (a_1 b_0 + a_0 b_1) \cdot x^{n/2} + a_0 b_0$$

$$= a_1 b_1 \cdot x^n + a_0 b_0 +$$

$$(((a_1 + a_0)(b_0 + b_1) - a_1 b_1 - a_0 b_0) \cdot x^{n/2}$$

- Only need 3 multiplications with $n/2$ coefficients, instead of 4
- Need additional additions and subtractions
Karatsuba’s method

- Split inputs in half: \( a = a_1 x^{n/2} + a_0, b = b_1 x^{n/2} + b_0 \)

\[
c = a \cdot b = (a_1 \cdot x^{n/2} + a_0)(b_1 \cdot x^{n/2} + b_0) \\
= a_1 b_1 \cdot x^n + (a_1 b_0 + a_0 b_1) \cdot x^{n/2} + a_0 b_0 \\
= a_1 b_1 \cdot x^n + a_0 b_0 + \\
((a_1 + a_0)(b_0 + b_1) - a_1 b_1 - a_0 b_0) \cdot x^{n/2}
\]

- Only need 3 multiplications with \( n/2 \) coefficients, instead of 4
- Need additional additions and subtractions
- Can be applied recursively
  - But at some threshold you want to use schoolbooks
Generalisation of Karatsuba: Split in k parts
Toom-Cook method

- Generalisation of Karatsuba: Split in k parts
- Toom-3: $a = a_2 Y^2 + a_1 Y + a_0$, $b = b_2 Y^2 + b_1 Y + b_0$
  - for $Y = x^{n/3}$
Toom-Cook method

- Generalisation of Karatsuba: Split in k parts
- Toom-3: $a = a_2 Y^2 + a_1 Y + a_0$, $b = b_2 Y^2 + b_1 Y + b_0$
  - for $Y = x^{n/3}$
  - Evaluate $a$ and $b$ for $Y = \{0, 1, -1, 2, \infty\}$
Toom-Cook method

- Generalisation of Karatsuba: Split in k parts
- Toom-3: \( a = a_2 Y^2 + a_1 Y + a_0, \ b = b_2 Y^2 + b_1 Y + b_0 \)
  - for \( Y = x^{n/3} \)
  - Evaluate \( a \) and \( b \) for \( Y = \{0, 1, -1, 2, \infty\} \)
  - Perform 5 multiplications with \( n/3 \) coefficients
  - Interpolate \( c = a \cdot b = c_4 Y^4 + c_3 Y^3 + c_2 Y^2 + c_1 Y + c_0 \)
Toom-Cook method

- Generalisation of Karatsuba: Split in $k$ parts
- Toom-3: $a = a_2 Y^2 + a_1 Y + a_0$, $b = b_2 Y^2 + b_1 Y + b_0$
  - for $Y = x^{n/3}$
  - Evaluate $a$ and $b$ for $Y = \{0, 1, -1, 2, \infty\}$
  - Perform 5 multiplications with $n/3$ coefficients
  - Interpolate $c = a \cdot b = c_4 Y^4 + c_3 Y^3 + c_2 Y^2 + c_1 Y + c_0$
- Similar for Toom-4
  - 7 multiplications with $n/4$ coefficients
Problem in practice: Toom needs multiplication in $\mathbb{Z}$

- We only have 16-bit coefficients, i.e. $\mathbb{Z}_{2^{16}}$
Toom-Cook method

- Problem in practice: Toom needs multiplication in $\mathbb{Z}$
  - We only have 16-bit coefficients, i.e. $\mathbb{Z}_{2^{16}}$
  - Leads to loss of precision due to divisions in interpolation
    - Toom-3: Division by 2, loses 1 bit $\rightarrow \mathbb{Z}_{2^{15}}$
    - Toom-4: Division by 8, loses 3 bit $\rightarrow \mathbb{Z}_{2^{13}}$
Problem in practice: Toom needs multiplication in $\mathbb{Z}$

- We only have 16-bit coefficients, i.e. $\mathbb{Z}_{2^{16}}$
- Leads to loss of precision due to divisions in interpolation
  - Toom-3: Division by 2, loses 1 bit → $\mathbb{Z}_{2^{15}}$
  - Toom-4: Division by 8, loses 3 bit → $\mathbb{Z}_{2^{13}}$

We limit our work to Toom-3 and Toom-4, since higher order Toom would lose even more bits
Problem in practice: Toom needs multiplication in $\mathbb{Z}$

- We only have 16-bit coefficients, i.e. $\mathbb{Z}_{2^{16}}$
- Leads to loss of precision due to divisions in interpolation
  - Toom-3: Division by 2, loses 1 bit $\rightarrow \mathbb{Z}_{2^{15}}$
  - Toom-4: Division by 8, loses 3 bit $\rightarrow \mathbb{Z}_{2^{13}}$

- We limit our work to Toom-3 and Toom-4, since higher order Toom would lose even more bits

- Can be used in combination or recursively, but losses add up
  - Toom-4 $+$ Toom-3 $\rightarrow \mathbb{Z}_{2^{12}}$
What’s the best method?

- Asymptotic: Toom-4 wins
- ... what about $n = 701$?
What’s the best method?

- Asymptotic: Toom-4 wins
- ... what about \( n = 701 \)?
- Our approach: Try all
- We need
  - Fast Karatsuba for all \( n \)
  - Fast Toom-4 for all \( n \)
  - Fast Toom-3 for all \( n \)
  - Fast schoolbooks for small \( n \)
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>
Fast schoolbook multiplication [N=2]

\[
\begin{array}{cccc}
  a_5 b_0 & a_4 b_0 & a_3 b_0 & a_2 b_0 \\
  a_5 b_1 & a_4 b_1 & a_3 b_1 & a_2 b_1 \\
  a_5 b_2 & a_4 b_2 & a_3 b_2 & a_2 b_2 & a_1 b_2 & a_0 b_2 \\
  a_5 b_3 & a_4 b_3 & a_3 b_3 & a_2 b_3 & a_1 b_3 & a_0 b_3 \\
  a_5 b_4 & a_4 b_4 & a_3 b_4 & a_2 b_4 & a_1 b_4 & a_0 b_4 \\
  a_5 b_5 & a_4 b_5 & a_3 b_5 & a_2 b_5 & a_1 b_5 & a_0 b_5 \\
\end{array}
\]
Fast schoolbook multiplication \([N=2]\)

- 3 multiplications instead of 4
Fast schoolbook multiplication \([N=4]\)

\[
\begin{array}{cccc}
  a_5 b_0 & a_4 b_0 & a_3 b_0 & a_2 b_0 \\
  a_5 b_1 & a_4 b_1 & a_3 b_1 & a_2 b_1 \\
  a_5 b_2 & a_4 b_2 & a_3 b_2 & a_2 b_2 \\
  a_5 b_3 & a_4 b_3 & a_3 b_3 & a_2 b_3 \\
  a_5 b_4 & a_4 b_4 & a_3 b_4 & a_2 b_4 \\
  a_5 b_5 & a_4 b_5 & a_3 b_5 & a_2 b_5 \\
\end{array}
\]
Fast schoolbook multiplication \([N=4]\)

- 10 multiplications instead of 16
Fast schoolbook multiplication \([N=6]\)
Fast schoolbook multiplication [N=6]

- 21 multiplications instead of 36
Fast schoolbook multiplication \[N=12\]

- How many can we fit in registers?
- 16 registers minus SP and PC → we fit 24 coefficients
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
Fast schoolbook multiplication: Reduce repacks

<table>
<thead>
<tr>
<th></th>
<th>$a_5b_0$</th>
<th>$a_4b_0$</th>
<th>$a_3b_0$</th>
<th>$a_2b_0$</th>
<th>$a_1b_0$</th>
<th>$a_0b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_5b_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_4b_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_3b_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_2b_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_1b_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_0b_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_5b_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_4b_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_3b_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_2b_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_1b_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_0b_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_5b_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_4b_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_3b_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_2b_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_1b_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_0b_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_5b_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_4b_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_3b_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_2b_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_1b_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_0b_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_5b_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_4b_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_3b_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_2b_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_1b_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_0b_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $R_0 = a_1 | a_0$, $R_1 = a_3 | a_2$, $R_2 = a_5 | a_4$
- $R_3 = b_1 | b_0$, $R_4 = b_3 | b_2$, $R_5 = b_5 | b_4$
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
Fast schoolbook multiplication: Reduce repacks

First do odd columns

- $R0 = a_1|a_0$, $R1 = a_3|a_2$, $R2 = a_5|a_4$
- $R3 = b_1|b_0$, $R4 = b_3|b_2$, $R5 = b_5|b_4$
Fast schoolbook multiplication: Reduce repacks

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a_5b_0$</td>
<td>$a_4b_0$</td>
<td>$a_3b_0$</td>
<td>$a_2b_0$</td>
<td>$a_1b_0$</td>
</tr>
<tr>
<td>$a_5b_1$</td>
<td></td>
<td>$a_4b_1$</td>
<td>$a_3b_1$</td>
<td>$a_2b_1$</td>
<td>$a_1b_1$</td>
</tr>
<tr>
<td>$a_5b_2$</td>
<td>$a_4b_2$</td>
<td></td>
<td>$a_3b_2$</td>
<td>$a_2b_2$</td>
<td>$a_1b_2$</td>
</tr>
<tr>
<td>$a_5b_3$</td>
<td>$a_4b_3$</td>
<td>$a_3b_3$</td>
<td></td>
<td>$a_2b_3$</td>
<td>$a_1b_3$</td>
</tr>
<tr>
<td>$a_5b_4$</td>
<td>$a_4b_4$</td>
<td>$a_3b_4$</td>
<td>$a_2b_4$</td>
<td></td>
<td>$a_1b_4$</td>
</tr>
<tr>
<td>$a_5b_5$</td>
<td>$a_4b_5$</td>
<td>$a_3b_5$</td>
<td>$a_2b_5$</td>
<td>$a_1b_5$</td>
<td></td>
</tr>
</tbody>
</table>

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

- Schoolbook is faster for $n \leq 16$
Schoolbook vs. Karatsuba

- Schoolbook is faster for $n \leq 16$
- Karatsuba is faster for $n > 36$
Schoolbook vs. Karatsuba

- Schoolbook is faster for $n \leq 16$
- Karatsuba is faster for $n > 36$
- We are mainly interested in $n = \{10, 11, 12, 16\}$
Schoolbook vs. Karatsuba

- Schoolbook is faster for $n \leq 16$
- Karatsuba is faster for $n > 36$
- We are mainly interested in $n = \{10, 11, 12, 16\}$
  - or multiples $\{20, 22, 24, 32\}$
Schoolbook vs. Karatsuba

- Schoolbook is faster for $n \leq 16$
- Karatsuba is faster for $n > 36$
- We are mainly interested in $n = \{10, 11, 12, 16\}$
  - or multiples $\{20, 22, 24, 32\}$
- For $\{20, 22, 24, 32\}$ Karatsuba is faster
Karatsuba vs. Toom-4 vs. Toom-3

- Toom and then multiple layers of Karatsuba
- Should never increase since you can always pad
- Some schoolbooks are just not that optimized
We are mainly interested in $n = \{256, 701, 743, 1024\}$
- at some point we gave up making this plot look nice
Saber \([n = 256, q = 2^{13}]\)

- No Toom-4 + Toom-3 possible, because \(q\) is too large
No Toom-4+Toom-3 possible, because $q$ is too large

Toom-4 is optimal; but Karatsuba only and Toom-3 are close
No Toom-4 possible, because $q$ is too large
No Toom-4 possible, because $q$ is too large

Karatsuba is faster than Toom-3 for $n = 256$ (but it’s close)
NTRU-HRSS \([n = 701, q = 2^{13}]\)

<table>
<thead>
<tr>
<th>sb</th>
<th>cycles</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>K only</td>
<td>11</td>
<td>230 132</td>
</tr>
<tr>
<td>T3</td>
<td>15</td>
<td>217 436</td>
</tr>
<tr>
<td>T4</td>
<td>11</td>
<td>182 129</td>
</tr>
<tr>
<td>T4 + T3</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
NTRU-HRSS \([n = 701, q = 2^{13}]\)

<table>
<thead>
<tr>
<th></th>
<th>sb</th>
<th>cycles</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>K only</td>
<td>11</td>
<td>230 132</td>
<td>5 676</td>
</tr>
<tr>
<td>T3</td>
<td>15</td>
<td>217 436</td>
<td>9 384</td>
</tr>
<tr>
<td>T4</td>
<td>11</td>
<td>182 129</td>
<td>10 596</td>
</tr>
<tr>
<td>T4 + T3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

\[ T_4 \]
NTRUEncrypt \( [n = 743, \, q = 2^{11}] \)

<table>
<thead>
<tr>
<th></th>
<th>sb</th>
<th>cycles</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>K only</td>
<td>12</td>
<td>247 489</td>
<td>6 012</td>
</tr>
<tr>
<td>T3</td>
<td>16</td>
<td>219 061</td>
<td>9 920</td>
</tr>
<tr>
<td><strong>T4</strong></td>
<td>12</td>
<td><strong>196 940</strong></td>
<td><strong>11 208</strong></td>
</tr>
<tr>
<td>T4+T3</td>
<td>16</td>
<td>197 227</td>
<td>12 152</td>
</tr>
</tbody>
</table>

- We could use Toom-4+Toom-3 here
NTRUEncrypt \([n = 743, q = 2^{11}]\)

<table>
<thead>
<tr>
<th></th>
<th>sb</th>
<th>cycles</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>K only</td>
<td>12</td>
<td>247 489</td>
<td>6 012</td>
</tr>
<tr>
<td>T3</td>
<td>16</td>
<td>219 061</td>
<td>9 920</td>
</tr>
<tr>
<td>T4</td>
<td>12</td>
<td>196 940</td>
<td>11 208</td>
</tr>
<tr>
<td>T4+T3</td>
<td>16</td>
<td>197 227</td>
<td>12 152</td>
</tr>
</tbody>
</table>

- We could use Toom-4+Toom-3 here
- but Toom-4 is faster (again quite close)
**RLizard** \( n = 1024, q = 2^{11} \)

<table>
<thead>
<tr>
<th></th>
<th>sb</th>
<th>cycles</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>K only</td>
<td>16</td>
<td>400 810</td>
<td>8 188</td>
</tr>
<tr>
<td>T3</td>
<td>11</td>
<td>360 589</td>
<td>13 756</td>
</tr>
<tr>
<td><strong>T4</strong></td>
<td>16</td>
<td><strong>313 744</strong></td>
<td><strong>15 344</strong></td>
</tr>
<tr>
<td>T4+T3</td>
<td>11</td>
<td>315 788</td>
<td>16 816</td>
</tr>
</tbody>
</table>
RLizard \( [n = 1024, \ q = 2^{11}] \)

\[
\begin{array}{c}
1026 \\
\downarrow_{T3} \\
342 \\
\downarrow_{K} \\
171 \\
\downarrow_{K} \\
85 \quad 86 \\
\downarrow_{K} \quad \downarrow_{K} \\
42 \quad 43 \\
\downarrow_{K} \quad \downarrow_{K} \\
21 \quad 22 \\
\downarrow_{K} \quad \downarrow_{K} \\
10 \quad 11
\end{array}
\]

\[
\begin{array}{c}
1024 \\
\downarrow_{K} \\
512 \\
\downarrow_{K} \\
256 \\
\downarrow_{K} \\
128 \\
\downarrow_{K} \\
64 \\
\downarrow_{K} \\
32 \\
\downarrow_{K} \\
16
\end{array}
\]

<table>
<thead>
<tr>
<th>stack</th>
<th>sb</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>K only</td>
<td>16</td>
<td>400 810</td>
</tr>
<tr>
<td>T3</td>
<td>11</td>
<td>360 589</td>
</tr>
<tr>
<td>T4</td>
<td>16</td>
<td>313 744</td>
</tr>
<tr>
<td>T4+T3</td>
<td>11</td>
<td>315 788</td>
</tr>
<tr>
<td>Implementation</td>
<td>Clock Cycles</td>
<td>Stack Usage</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>K: 6530k</td>
<td>K: 12616</td>
</tr>
<tr>
<td></td>
<td>E: 8684k</td>
<td>E: 14896</td>
</tr>
<tr>
<td></td>
<td>D: 10581k</td>
<td>D: 15992</td>
</tr>
<tr>
<td>Saber Reference</td>
<td>K: 1147k</td>
<td>K: 13883</td>
</tr>
<tr>
<td>[KMRV18]</td>
<td>E: 1444k</td>
<td>E: 16667</td>
</tr>
<tr>
<td></td>
<td>D: 1543k</td>
<td>D: 17763</td>
</tr>
<tr>
<td>This work</td>
<td>K: 949k</td>
<td>K: 13248</td>
</tr>
<tr>
<td></td>
<td>E: 1232k</td>
<td>E: 15528</td>
</tr>
<tr>
<td></td>
<td>D: 1260k</td>
<td>D: 16624</td>
</tr>
<tr>
<td>Kindi-256-3-4-2 Reference</td>
<td>K: 21794k</td>
<td>K: 59864</td>
</tr>
<tr>
<td></td>
<td>E: 28176k</td>
<td>E: 71000</td>
</tr>
<tr>
<td></td>
<td>D: 37129k</td>
<td>D: 84096</td>
</tr>
<tr>
<td>This work</td>
<td>K: 1010k</td>
<td>K: 44264</td>
</tr>
<tr>
<td></td>
<td>E: 1365k</td>
<td>E: 55392</td>
</tr>
<tr>
<td></td>
<td>D: 1563k</td>
<td>D: 64376</td>
</tr>
<tr>
<td>NTRU-HRSS Reference</td>
<td>K: 205156k</td>
<td>K: 10020</td>
</tr>
<tr>
<td></td>
<td>E: 5166k</td>
<td>E: 8956</td>
</tr>
<tr>
<td></td>
<td>D: 15067k</td>
<td>D: 10204</td>
</tr>
<tr>
<td>This work</td>
<td>K: 161790k</td>
<td>K: 23396</td>
</tr>
<tr>
<td></td>
<td>E: 432k</td>
<td>E: 19492</td>
</tr>
<tr>
<td></td>
<td>D: 863k</td>
<td>D: 22140</td>
</tr>
<tr>
<td>NTRU-KEM-743 Reference</td>
<td>K: 59815k</td>
<td>K: 14148</td>
</tr>
<tr>
<td></td>
<td>E: 7540k</td>
<td>E: 13372</td>
</tr>
<tr>
<td></td>
<td>D: 14229k</td>
<td>D: 18036</td>
</tr>
<tr>
<td>This work</td>
<td>K: 5663k</td>
<td>K: 25320</td>
</tr>
<tr>
<td></td>
<td>E: 1655k</td>
<td>E: 23808</td>
</tr>
<tr>
<td></td>
<td>D: 1904k</td>
<td>D: 28472</td>
</tr>
<tr>
<td>RLizard-1024 Reference</td>
<td>K: 26423k</td>
<td>K: 4272</td>
</tr>
<tr>
<td></td>
<td>E: 32156k</td>
<td>E: 10532</td>
</tr>
<tr>
<td></td>
<td>D: 53181k</td>
<td>D: 12636</td>
</tr>
<tr>
<td>This work</td>
<td>K: 537k</td>
<td>K: 27720</td>
</tr>
<tr>
<td></td>
<td>E: 1358k</td>
<td>E: 33328</td>
</tr>
<tr>
<td></td>
<td>D: 1740k</td>
<td>D: 35448</td>
</tr>
</tbody>
</table>
Results

- Large speed-ups vs. reference C implementations
Results

- Large speed-ups vs. reference C implementations
- Speed-up of up to 49x
Results

- Large speed-ups vs. reference C implementations
- Speed-up of up to 49x
- Higher stack usage because of intermediate limbs
  - Exception: Kindi already uses “optimised” multiplication in C
<table>
<thead>
<tr>
<th>implementation</th>
<th>clock cycles</th>
<th>stack usage</th>
</tr>
</thead>
<tbody>
<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>[KMRV18]</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>
</tbody>
</table>

- Speed-ups of 17%, 15%, and 18% compared to CHES18 paper
## Results: Saber

<table>
<thead>
<tr>
<th>implementation</th>
<th>clock cycles</th>
<th>stack usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K:</td>
<td>6530k</td>
<td>K: 12616</td>
</tr>
<tr>
<td>E:</td>
<td>8684k</td>
<td>E: 14896</td>
</tr>
<tr>
<td>D:</td>
<td>10581k</td>
<td>D: 15992</td>
</tr>
<tr>
<td>[KMRV18]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K:</td>
<td>1147k</td>
<td>K: 13883</td>
</tr>
<tr>
<td>E:</td>
<td>1444k</td>
<td>E: 16667</td>
</tr>
<tr>
<td>D:</td>
<td>1543k</td>
<td>D: 17763</td>
</tr>
<tr>
<td>This work</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K:</td>
<td>949k</td>
<td>K: 13248</td>
</tr>
<tr>
<td>E:</td>
<td>1232k</td>
<td>E: 15528</td>
</tr>
<tr>
<td>D:</td>
<td>1260k</td>
<td>D: 16624</td>
</tr>
</tbody>
</table>

- Speed-ups of 17%, 15%, and 18% compared to CHES18 paper
- Stack usage slightly better – but they also have a stack-optimised variant
Results vs. Round5 (CARDIS’18)

- HILA5 recently merged with Round2 → Round5
- Optimised for the M4 at CARDIS’18 [SBGM⁺18]
### NIST Security level 1

<table>
<thead>
<tr>
<th></th>
<th>clock cycles</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K</td>
<td>E</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>NTRU-HRSS</td>
<td>161 790k</td>
<td>432k</td>
<td>863k</td>
<td></td>
</tr>
<tr>
<td>R5ND_1PKEb</td>
<td>658k</td>
<td>984k</td>
<td>1 265k</td>
<td></td>
</tr>
</tbody>
</table>

- Encapsulation and decapsulation of NTRU-HRSS are faster than R5ND_1PKEb
  - > factor 2 for encapsulation
### Results vs. Round5 (CARDIS’18)

#### NIST Security level 1

<table>
<thead>
<tr>
<th></th>
<th>clock cycles</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>K</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>NTRU-HRSS</td>
<td>This work</td>
<td>161 790k</td>
<td>432k</td>
<td>863k</td>
</tr>
<tr>
<td>R5ND_1PKEb</td>
<td>[SBGM$^+18$]</td>
<td>658k</td>
<td>984k</td>
<td>1 265k</td>
</tr>
</tbody>
</table>

#### NIST Security level 3

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K</td>
<td>E</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>Saber</td>
<td>This work</td>
<td>949k</td>
<td>1 232k</td>
<td>1 260k</td>
</tr>
<tr>
<td>R5ND_3PKEb</td>
<td>[SBGM$^+18$]</td>
<td>1 032k</td>
<td>1 510k</td>
<td>1 913k</td>
</tr>
</tbody>
</table>

- Encapsulation and decapsulation of NTRU-HRSS are faster than R5ND_1PKEb
  - > factor 2 for encapsulation
- Saber is faster than R5ND_3PKEb for all operations
Is there anything else we can optimise?
<table>
<thead>
<tr>
<th>scheme</th>
<th>polymul</th>
<th>hashing</th>
<th>randombytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>Saber</td>
<td>37%</td>
<td>38%</td>
<td>46%</td>
</tr>
<tr>
<td>Kindi</td>
<td>37%</td>
<td>36%</td>
<td>40%</td>
</tr>
<tr>
<td>NTRU-HRSS</td>
<td>1%</td>
<td>42%</td>
<td>63%</td>
</tr>
<tr>
<td>NTRUEncrypt</td>
<td>31%</td>
<td>12%</td>
<td>21%</td>
</tr>
<tr>
<td>RLizard</td>
<td>59%</td>
<td>46%</td>
<td>54%</td>
</tr>
</tbody>
</table>

- Still quite some time spent in polymul (12% - 63%)
### Profiling optimised implementations

<table>
<thead>
<tr>
<th>scheme</th>
<th>polymul</th>
<th>hashing</th>
<th>randombytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>Saber</td>
<td>37%</td>
<td>38%</td>
<td>46%</td>
</tr>
<tr>
<td>Kindi</td>
<td>37%</td>
<td>36%</td>
<td>40%</td>
</tr>
<tr>
<td>NTRU-HRSS</td>
<td>1%</td>
<td>42%</td>
<td>63%</td>
</tr>
<tr>
<td>NTRUEncrypt</td>
<td>31%</td>
<td>12%</td>
<td>21%</td>
</tr>
<tr>
<td>RLizard</td>
<td>59%</td>
<td>46%</td>
<td>54%</td>
</tr>
</tbody>
</table>

- Still quite some time spent in polymul (12% - 63%)
- Additionally, a lot of time is spent in hashing
  - SHA3: Optimised M4 assembly from the Keccak team
  - SHA2: Fast C impl from SUPERCOP (we tried assembly as well)
Profiling optimised implementations

<table>
<thead>
<tr>
<th>scheme</th>
<th>polymul K</th>
<th>polymul E</th>
<th>polymul D</th>
<th>hashing K</th>
<th>hashing E</th>
<th>hashing D</th>
<th>randombytes K</th>
<th>randombytes E</th>
<th>randombytes D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saber</td>
<td>37%</td>
<td>38%</td>
<td>46%</td>
<td>54%</td>
<td>54%</td>
<td>43%</td>
<td>&lt;1%</td>
<td>&lt;1%</td>
<td>0%</td>
</tr>
<tr>
<td>Kindi</td>
<td>37%</td>
<td>36%</td>
<td>40%</td>
<td>41%</td>
<td>44%</td>
<td>39%</td>
<td>&lt;1%</td>
<td>&lt;1%</td>
<td>0%</td>
</tr>
<tr>
<td>NTRU-HRSS</td>
<td>1%</td>
<td>42%</td>
<td>63%</td>
<td>&lt;1%</td>
<td>27%</td>
<td>9%</td>
<td>&lt;1%</td>
<td>&lt;1%</td>
<td>0%</td>
</tr>
<tr>
<td>NTRUEncrypt</td>
<td>31%</td>
<td>12%</td>
<td>21%</td>
<td>0%</td>
<td>73%</td>
<td>62%</td>
<td>2%</td>
<td>3%</td>
<td>0%</td>
</tr>
<tr>
<td>RLizard</td>
<td>59%</td>
<td>46%</td>
<td>54%</td>
<td>0%</td>
<td>46%</td>
<td>36%</td>
<td>22%</td>
<td>&lt;1%</td>
<td>0%</td>
</tr>
</tbody>
</table>

- Still quite some time spent in polymul (12% - 63%)
- Additionally, a lot of time is spent in hashing
  - SHA3: Optimised M4 assembly from the Keccak team
  - SHA2: Fast C impl from SUPERCOP (we tried assembly as well)
- Some schemes use randombytes to sample a lot randomness
Fastest PQC scheme implementation on the Cortex-M4
Summary

- Fastest PQC scheme implementation on the Cortex-M4
  - More than twice as fast for encapsulation as any previous implementation on the M4
Fastest PQC scheme implementation on the Cortex-M4

- More than twice as fast for encapsulation as any previous implementation on the M4

Our scripts can generate code for any $n \leq 1024$ and $q = 2^m \leq 2^{16}$. 
Summary

- Fastest PQC scheme implementation on the Cortex-M4
  - More than twice as fast for encapsulation as any previous implementation on the M4
- Our scripts can generate code for any $n \leq 1024$ and $q = 2^m \leq 2^{16}$.
  - easy to adopt to other parameter sets

Checkout our code - it’s public domain

https://github.com/mupq/polymul-z2mx-m4
https://github.com/mupq/pqm4

Paper is available at kannwischer.eu
Summary

- Fastest PQC scheme implementation on the Cortex-M4
  - More than twice as fast for encapsulation as any previous implementation on the M4
- Our scripts can generate code for any $n \leq 1024$ and $q = 2^m \leq 2^{16}$.
  - easy to adopt to other parameter sets
  - e.g., in case of parameter changes in the next rounds
Summary

- Fastest PQC scheme implementation on the Cortex-M4
  - More than twice as fast for encapsulation as any previous implementation on the M4
- Our scripts can generate code for any $n \leq 1024$ and $q = 2^m \leq 2^{16}$.
  - easy to adopt to other parameter sets
  - e.g., in case of parameter changes in the next rounds
- Checkout our code - it’s public domain
Fastest PQC scheme implementation on the Cortex-M4
  • More than twice as fast for encapsulation as any previous implementation on the M4

Our scripts can generate code for any $n \leq 1024$ and $q = 2^m \leq 2^{16}$.
  • easy to adopt to other parameter sets
  • e.g., in case of parameter changes in the next rounds

Checkout our code - it’s public domain
  • https://github.com/mupq/polymul-z2mx-m4
Summary

- Fastest PQC scheme implementation on the Cortex-M4
  - More than twice as fast for encapsulation as any previous implementation on the M4
- Our scripts can generate code for any $n \leq 1024$ and $q = 2^m \leq 2^{16}$.
  - easy to adopt to other parameter sets
  - e.g., in case of parameter changes in the next rounds
- Checkout our code - it’s public domain
  - https://github.com/mupq/polymul-z2mx-m4
  - https://github.com/mupq/pqm4

Paper is available at kannwischer.eu
Summary

- Fastest PQC scheme implementation on the Cortex-M4
  - More than twice as fast for encapsulation as any previous implementation on the M4
- Our scripts can generate code for any $n \leq 1024$ and $q = 2^m \leq 2^{16}$.
  - easy to adopt to other parameter sets
  - e.g., in case of parameter changes in the next rounds
- Checkout our code - it’s public domain
  - https://github.com/mupq/polymul-z2mx-m4
  - https://github.com/mupq/pqm4
- Paper is available at kannwischer.eu
Conclusions

Run-time is dominated by poly multiplication and hashing
Conclusions

- Run-time is dominated by poly multiplication and hashing
- Optimising these schemes for other platforms
Conclusions

- Run-time is dominated by poly multiplication and hashing
- Optimising these schemes for other platforms
  1. Optimise polynomial multiplication
Conclusions

- Run-time is dominated by poly multiplication and hashing
- Optimising these schemes for other platforms
  1. Optimise polynomial multiplication
  2. Use fast hash implementation
Conclusions

- Run-time is dominated by poly multiplication and hashing
- Optimising these schemes for other platforms
  1. Optimise polynomial multiplication
  2. Use fast hash implementation
  3. Replace randombytes with an appropriate fast implementation
Some reference implementations we looked at are...
Some reference implementations we looked at are . . .

- Highly non-constant time
Some reference implementations we looked at are . . .

- Highly non-constant time
- Dynamic memory allocations (#free’s != #malloc’s)
Some reference implementations we looked at are...

- Highly non-constant time
- Dynamic memory allocations (\#free’s $\neq$ \#malloc’s)
- A lot of compiler warnings
Some reference implementations we looked at are . . .

- Highly non-contant time
- Dynamic memory allocations (#free’s != #malloc’s)
- A lot of compiler warnings
- Incorrectly implementing the NIST API
Some reference implementations we looked at are . . .

- Highly non-constant time
- Dynamic memory allocations (#free’s != #malloc’s)
- A lot of compiler warnings
- Incorrectly implementing the NIST API

We fixed some of those problems, but not all
Some reference implementations we looked at are . . .

- Highly non-contant time
- Dynamic memory allocations (#free’s != #malloc’s)
- A lot of compiler warnings
- Incorrectly implementing the NIST API

- We fixed some of those problems, but not all
- More work required
PQC code quality

Slide credit: Daniel J. Bernstein and Tanja Lange
(35c3, The year in post-quantum crypto, 28 December 2018):
checkout our code - it’s public domain

https://github.com/mupq/polymul-z2mx-m4
https://github.com/mupq/pqm4

paper is available at kannwischer.eu

Thank you!
Angshuman Karmakar, Jose Maria Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede.

**Saber on ARM.**


**Shorter messages and faster post-quantum encryption with Round5 on Cortex M.**