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$^1$

November 9, 2018, Crypto Working Group, Utrecht

$^1$Radboud University, Nijmegen, The Netherlands
matthias@kannwischer.eu, joost@joostrijneveld.nl, peter@cryptojedi.org
Motivation

- Lattice-crypto is fast . . . and there is also Shor
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?
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?
- Expensive in (ideal) lattice crypto: Polynomial multiplication
- If \( \mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1) \) with \( n = 2^m \), \( q \) prime, and \( q \equiv 1 \mod 2n \)
  - Use NTT-based multiplication
  - Also studied for microprocessors
- . . . but what about other rings \( \mathcal{R}_q \)?
This work: Optimise multiplication in $\mathbb{Z}_q[X]$

- $R_q = \mathbb{Z}_{2^m}[x]/(f(x))$
  - For arbitrary $f(x)$; do reduction outside of optimised code
  - Applies to Saber, Kindi, NTRU-HRSS, NTRUEncrypt, and RLizard
This work: Optimise multiplication in $\mathbb{Z}_q[X]$

- $R_q = \mathbb{Z}_{2^m}[x]/(f(x))$
  - For arbitrary $f(x)$; do reduction outside of optimised code
  - Applies to Saber, Kindi, NTRU-HRSS, NTRUEncrypt, and RLizard
- Platform: Cortex-M4 (STM32F4DISCOVERY)
  - Ca. 12 Euros
  - 32-bit; more advanced instructions than M0, M1, and M3
  - 192 KiB RAM, 1 MiB ROM, 168 MHz
This work: Optimise multiplication in $\mathbb{Z}_q[X]$

- $R_q = \mathbb{Z}_{2^m}[x]/(f(x))$
  - For arbitrary $f(x)$; do reduction outside of optimised code
  - Applies to Saber, Kindi, NTRU-HRSS, NTRUEncrypt, and RLizard
- Platform: Cortex-M4 (STM32F4DISCOVERY)
  - Ca. 12 Euros
  - 32-bit; more advanced instructions than M0, M1, and M3
  - 192 KiB RAM, 1 MiB ROM, 168 MHz
- Generate optimised assembly for $N \leq 1024$, $q = 2^m \leq 2^{16}$
  - Be adaptable to different parameter sets and schemes
- Focus on speed; not RAM usage or code size
  - See https://downloadmoreram.com/
This work: Optimise multiplication in $\mathbb{Z}_q[X]$

- $R_q = \mathbb{Z}_{2^m}[x]/(f(x))$
  - For arbitrary $f(x)$; do reduction outside of optimised code
  - Applies to Saber, Kindi, NTRU-HRSS, NTRUEncrypt, and RLizard
- Platform: Cortex-M4 (STM32F4DISCOVERY)
  - Ca. 12 Euros
  - 32-bit; more advanced instructions than M0, M1, and M3
  - 192 KiB RAM, 1 MiB ROM, 168 MHz
- Generate optimised assembly for $N \leq 1024$, $q = 2^m \leq 2^{16}$
  - Be adaptable to different parameter sets and schemes
- Focus on speed; not RAM usage or code size
  - See https://downloadmoreram.com/
- Building on top of PQM4
Deliverable of the PQCRYPTO project, mostly public domain
Deliverable of the PQCRYPTO project, mostly public domain

Benchmarking and testing framework for NISTPQC

- Automated functional tests
- Automated comparison of testvectors (ref–opt, ref–host)
- Automated benchmarking (speed and stack usage)
Deliverable of the PQCRYPTO project, mostly public domain

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)
Deliverable of the PQCRYPTO project, mostly public domain

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!
1. Saber, RLizard, NTRU-HRSS, NTRUEncrypt, and Kindi

2. Multiplying polynomials
   - Karatsuba’s method
   - Toom-Cook method
   - Fast schoolbook multiplication

3. Results
   - Polynomial multiplications
   - Optimised schemes
Saber, RLizard, NTRU-HRSS, NTRUEncrypt, and Kindi
Saber

Saber.KeyGen()

1: $\rho \leftarrow \text{Sample}_{\{0,1\}^{256}}$
2: $A \leftarrow \text{Sample}_{\mathcal{R}_q^{\ell} \times \ell}(\rho)$
3: $s \leftarrow \text{Sample}_{\mathcal{R}_q^{\ell}}$
4: $b \leftarrow [A \otimes s + h] \in \mathcal{R}_p^{\ell}$
5: return $(pk = (\rho, b), sk = s)$

Saber.Enc($m, (\rho, b)$)

1: $A \leftarrow \text{Sample}_{\mathcal{R}_q^{\ell} \times \ell}(\rho)$
2: $s' \leftarrow \text{Sample}_{\mathcal{R}_q^{\ell}}$
3: $b' \leftarrow [A \otimes s' + h] \in \mathcal{R}_p^{\ell}$
4: $v' \leftarrow b \otimes [s'] \in \mathcal{R}_p$
5: $c_m \leftarrow [v' + (p/2) \cdot m] \in \mathcal{R}_{2t}$
6: return $(c_m, b')$

Saber.Dec((c_m, b'), s)

1: $v \leftarrow b' \otimes [s] \in \mathcal{R}_p$
2: $m' \leftarrow [v - (p/(2t)) \cdot c_m + h] \in \mathcal{R}_{2t}$
3: return $m'$
Saber

Saber.KeyGen()

1: \( \rho \leftarrow \text{Sample}_{\{0,1\}^{256}} \)
2: \( A \leftarrow \text{Sample}_{\mathcal{R}_q^{\ell \times \ell}}(\rho) \)
3: \( s \leftarrow \text{Sample}_{\mathcal{R}_q^{\ell}} \)
4: \( b \leftarrow [A \otimes s + h] \in \mathcal{R}_p^{\ell} \)
5: \text{return} (pk = (\rho, b), sk = s)

Saber.Enc \((m, (\rho, b))\)

1: \( A \leftarrow \text{Sample}_{\mathcal{R}_q^{\ell \times \ell}}(\rho) \)
2: \( s' \leftarrow \text{Sample}_{\mathcal{R}_q^{\ell}} \)
3: \( b' \leftarrow [A \otimes s' + h] \in \mathcal{R}_p^{\ell} \)
4: \( v' \leftarrow b \otimes [s'] \in \mathcal{R}_p \)
5: \( c_m \leftarrow [v' + (p/2) \cdot m] \in \mathcal{R}_{2t} \)
6: \text{return} \((c_m, b')\)

Saber.Dec \(((c_m, b'), s)\)

1: \( v \leftarrow b' \otimes [s] \in \mathcal{R}_p \)
2: \( m' \leftarrow [v - (p/(2t)) \cdot c_m + h] \in \mathcal{R}_2 \)
3: \text{return} \(m'\)
<table>
<thead>
<tr>
<th>RLizard.KeyGen()</th>
<th>RLizard.Enc((m, (a, b)))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: a, s, e ← Sample_{R_q}</td>
<td>1: r ← Sample_{R_q}</td>
</tr>
<tr>
<td>2: b ← −a ⊗ s + e ∈ R_q</td>
<td>2: c'_1 ← a ⊗ r ∈ R_q</td>
</tr>
<tr>
<td>3: return ((pk = (a, b), sk = s))</td>
<td>3: c'_2 ← b ⊗ r ∈ R_q</td>
</tr>
<tr>
<td></td>
<td>4: c_1 ← ((p/q) \cdot c'_1) ∈ R_p</td>
</tr>
<tr>
<td></td>
<td>5: c_2 ← ((p/q) \cdot ((q/2) \cdot m + c'_2))</td>
</tr>
<tr>
<td></td>
<td>6: return ((c_1, c_2))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RLizard.Dec((c_1, c_2), s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: m' ← ([(2/p) \cdot (c_2 + c_1 ⊗ s)])</td>
</tr>
<tr>
<td>2: return m'</td>
</tr>
<tr>
<td>RLizard.KeyGen ()</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>1: a, s, e ← Sample(_{R_q})</td>
</tr>
<tr>
<td>2: b ← −a ⊗ s + e ∈ (R_q)</td>
</tr>
<tr>
<td>3: return ((pk = (a, b), sk = s))</td>
</tr>
<tr>
<td>RLizard.Dec ((c(_1), c(_2)), s)</td>
</tr>
<tr>
<td>1: (m' \leftarrow [(2/p) \cdot (c_2 + c_1 \otimes s)])</td>
</tr>
<tr>
<td>2: return (m')</td>
</tr>
</tbody>
</table>
NTRU-HRSS.KeyGen ()
1: \( f, g \leftarrow \text{Sample}_{\mathcal{R}_p} \)
2: \( f_p^{-1} \leftarrow f^{-1} \in \mathcal{R}_p \)
3: \( f_q^{-1} \leftarrow f^{-1} \in \mathcal{R}_q' \)
4: \( h \leftarrow \Phi_1 \circ g \otimes f_q^{-1} \in \mathcal{R}_q \)
5: return \((pk = p \cdot h, sk = (f, f_p^{-1})\)

NTRU-HRSS.Enc \((m, (p \cdot h))\)
1: \( r \leftarrow \text{Sample}_{\mathcal{R}_q} \)
2: \( c \leftarrow h' \otimes r + m \in \mathcal{R}_q \)
3: return \(c\)

NTRU-HRSS.Dec \((c, (f, f_p^{-1}))\)
1: \( v \leftarrow c \otimes f \in \mathcal{R}_q \)
2: \( m' \leftarrow v \otimes f_p^{-1} \in \mathcal{R}_p \)
3: return \(m\)
### NTRU-HRSS.KeyGen ()

1. $f, g \leftarrow \text{Sample}_{\mathcal{R}_p}$
2. $f_p^{-1} \leftarrow f^{-1} \in \mathcal{R}_p$
3. $f_q^{-1} \leftarrow f^{-1} \in \mathcal{R}'_q$
4. $h \leftarrow \Phi_1 \ast g \otimes f_q^{-1} \in \mathcal{R}_q$
5. return $(pk = p \cdot h, sk = (f, f_p^{-1})$

### NTRU-HRSS.Enc ($m, (p \cdot h)$)

1. $r \leftarrow \text{Sample}_{\mathcal{R}_q}$
2. $c \leftarrow h' \otimes r + m \in \mathcal{R}_q$
3. return $c$

### NTRU-HRSS.Dec ($c, (f, f_p^{-1})$)

1. $v \leftarrow c \otimes f \in \mathcal{R}_q$
2. $m' \leftarrow v \otimes f_p^{-1} \in \mathcal{R}_p$
3. return $m$
NTRUEncrypt

NTRUEncrypt.KeyGen():
1: \( f, g \leftarrow \text{Sample}_{\mathcal{R}_q} \)
2: \( h \leftarrow (p \cdot g)/(p \cdot f + 1) \mod q \)
3: \( \text{return} \ (pk = h, sk = (f, h)) \)

NTRUEncrypt.Enc\( m, h \):
1: \( r \leftarrow \text{Sample}_{\mathcal{R}_q}(m, h) \)
2: \( t \leftarrow r \otimes h \)
3: \( m_{\text{mask}} \leftarrow \text{Sample}_{\mathcal{R}_q}(t) \)
4: \( m' \leftarrow m - m_{\text{mask}} \mod p \)
5: \( c \leftarrow t + m' \)
6: \( \text{return} \ c \)

NTRUEncrypt.Dec\( c, (f, h) \):
1: \( m' \leftarrow f \otimes c \mod p \)
2: \( t \leftarrow c - m \)
3: \( m_{\text{mask}} \leftarrow \text{Sample}_{\mathcal{R}_q}(t) \)
4: \( m \leftarrow m' + m_{\text{mask}} \mod p \)
5: \( r \leftarrow \text{Sample}_{\mathcal{R}_q}(m, h) \)
6: \( \text{if } p \cdot r \otimes h = t \text{ then} \)
7: \( \text{return } m \)
8: \( \text{else} \)
9: \( \text{return } \perp \)
10: \( \text{end if} \)
# NTRUEncrypt

## NTRUEncrypt.KeyGen()

1. \( f, g \leftarrow \text{Sample}_{R_q} \)
2. \( h \leftarrow (p \cdot g)/(p \cdot f + 1) \mod q \)
3. **return** \((pk = h, sk = (f, h))\)

## NTRUEncrypt.Enc \((m, h)\)

1. \( r \leftarrow \text{Sample}_{R_q}(m, h) \)
2. \( t \leftarrow r \otimes h \)
3. \( m_{\text{mask}} \leftarrow \text{Sample}_{R_q}(t) \)
4. \( m' \leftarrow m - m_{\text{mask}} \mod p \)
5. \( c \leftarrow t + m' \)
6. **return** \(c\)

## NTRUEncrypt.Dec \((c, (f, h))\)

1. \( m' \leftarrow f \otimes c \mod p \)
2. \( t \leftarrow c - m \)
3. \( m_{\text{mask}} \leftarrow \text{Sample}_{R_q}(t) \)
4. \( m \leftarrow m' + m_{\text{mask}} \mod p \)
5. \( r \leftarrow \text{Sample}_{R_q}(m, h) \)
6. **if** \( p \cdot r \otimes h = t \) **then**
7. **return** \(m\)
8. **else**
9. **return** \(⊥\)
10. **end if**
Kindi.KeyGen() 

1: $\mu \leftarrow \text{Sample}_{\{0,1\}^{256}}$
2: $A \leftarrow \text{Sample}_{\mathcal{R}_q^{\ell \times \ell}}(\mu)$
3: $r, r' \leftarrow \text{Sample}_{\mathcal{R}_q^\ell}$
4: $b \leftarrow A \otimes r + r'$
5: return $(pk = (b, \mu), sk = (r, b, \mu))$
Kindi.KeyGen()

1: \( \mu \leftarrow \text{Sample}_{\{0,1\}^{256}} \)
2: \( A \leftarrow \text{Sample}_{\mathbb{R}_q^{\ell \times \ell}}(\mu) \)
3: \( r, r' \leftarrow \text{Sample}_{\mathbb{R}_q^{\ell}} \)
4: \( b \leftarrow A \odot r + r' \)
5: \textbf{return} \ (pk = (b, \mu), sk = (r, b, \mu))
Kindi.Enc\left(m, (b, \mu)\right)

1. \(s_1 \leftarrow \text{Sample}_{\mathcal{R}_2}\)
2. \(A \leftarrow \text{Sample}_{\mathcal{R}_{q \times \ell}}(\mu)\)
3. \(p \leftarrow b + g\)
4. \(\tilde{s}_1 \leftarrow \text{Sample}_{\mathcal{R}_p}(s_1)\)
5. \((s_2, \ldots, s_{\ell}) \leftarrow \text{Sample}_{\mathcal{R}_{q}^{\ell-1}}(s_1)\)
6. \(s \leftarrow (s_1 + 2 \cdot \tilde{s}_1 - \lfloor p \rfloor, s_2 - \lfloor p \rfloor, \ldots, s_{\ell} - \lfloor p \rfloor) \in \mathcal{R}_{q}^{\ell}\)
7. \(\bar{u} \leftarrow \text{Sample}_{\{0,1\}^{n(\ell+1) \log 2p}}(s_1)\)
8. \(u \leftarrow \bar{u} \oplus m\)
9. \(e \leftarrow (u_1 - \lfloor p \rfloor, \ldots, u_{\ell} - \lfloor p \rfloor) \in \mathcal{R}_{q}^{\ell}\)
10. \(e_{\ell+1} \leftarrow u_{\ell+1} - \lfloor p \rfloor\)
11. \((c, c_{\ell+1}) \leftarrow (A \otimes s + e, p \otimes s + g \cdot \lfloor p \rfloor + e) \in \mathcal{R}_{q}^{\ell+1}\)
12. \textbf{return} \((c, c_{\ell+1})\)
Kindi.\text{Enc}(m, (b, \mu))$

1: \ s_1 \leftarrow \text{Sample}_{\mathcal{R}_2}
2: \ A \leftarrow \text{Sample}_{\mathcal{R}_{\ell \times \ell}}(\mu)
3: \ p \leftarrow b + g
4: \ \tilde{s}_1 \leftarrow \text{Sample}_{\mathcal{R}_p}(s_1)
5: \ (s_2, \ldots, s_\ell) \leftarrow \text{Sample}_{\mathcal{R}_{\ell-1}}(s_1)
6: \ s \leftarrow (s_1 + 2 \cdot \tilde{s}_1 - [p], s_2 - [p], \ldots, s_\ell - [p]) \in \mathcal{R}_\ell
7: \ \bar{u} \leftarrow \text{Sample}_{\{0,1\}^{n(\ell+1)\log 2p}}(s_1)
8: \ u \leftarrow \bar{u} \oplus m
9: \ e \leftarrow (u_1 - [p], \ldots, u_\ell - [p]) \in \mathcal{R}_q
10: \ e_{\ell+1} \leftarrow u_{\ell+1} - [p]
11: \ (c, c_{\ell+1}) \leftarrow (A \otimes s + e, p \otimes s + g \cdot [p] + e) \in \mathcal{R}_{\ell+1}
12: \ \text{return} \ (c, c_{\ell+1})
Kindi.Dec \((r, b, \mu, (c, c_{\ell+1}))\)

1: \(A \leftarrow \text{Sample}_{\mathcal{R}_q^{\ell \times \ell}}(\mu)\)
2: \(p \leftarrow b + g\)
3: \(v \leftarrow c_{\ell+1} - c \otimes r\)
4: \(s_1 \leftarrow ([v_1/2^{\log q-1}], \ldots, [v_n/2^{\log q-1}]) \in \mathcal{R}_2\)
5: \(\tilde{s}_1 \leftarrow \text{Sample}_{\mathcal{R}_p}(s_1)\)
6: \((s_2, \ldots, s_\ell) \leftarrow \text{Sample}_{\mathcal{R}_p^{\ell-1}}(s_1)\)
7: \(s \leftarrow (s_1 + 2 \cdot \tilde{s}_1 - [p], s_2 - [p], \ldots, s_\ell - [p])\)
8: \(\tilde{u} \leftarrow \text{Sample}_{\{0,1\}^{n(\ell+1) \log 2p}}(s_1)\)
9: \((e, e_{\ell+1}) \leftarrow (c - A \otimes s, c_{\ell+1} - p \otimes s) \in \mathcal{R}_q^{\ell+1}\)
10: \(u \leftarrow (e_1 + [p], \ldots, e_\ell + [p])\)
11: \(u_{\ell+1} \leftarrow e_{\ell+1} + [p]\)
12: \(m \leftarrow u \oplus \tilde{u}\)
13: \text{return } m
Kindi.Dec \((r, b, \mu, (c, c_{\ell+1}))\)

1: \(A \leftarrow \text{Sample}_{\mathcal{R}_q^{\ell \times \ell}}(\mu)\)
2: \(p \leftarrow b + g\)
3: \(v \leftarrow c_{\ell+1} - c \otimes r\)
4: \(s_1 \leftarrow ([v_1/2^{\log q-1}], \ldots, [v_n/2^{\log q-1}]) \in \mathcal{R}_2\)
5: \(\tilde{s}_1 \leftarrow \text{Sample}_{\mathcal{R}_p}(s_1)\)
6: \((s_2, \ldots, s_\ell) \leftarrow \text{Sample}_{\mathcal{R}_p^{\ell-1}}(s_1)\)
7: \(s \leftarrow (s_1 + 2 \cdot \tilde{s}_1 - [p], s_2 - [p], \ldots, s_\ell - [p])\)
8: \(\bar{u} \leftarrow \text{Sample}_{\{0,1\}^{n(\ell+1) \log 2p}}(s_1)\)
9: \((e, e_{\ell+1}) \leftarrow (c - A \otimes s, c_{\ell+1} - p \otimes s) \in \mathcal{R}_q^{\ell+1}\)
10: \(u \leftarrow (e_1 + [p], \ldots e_\ell + [p])\)
11: \(u_{\ell+1} \leftarrow e_{\ell+1} + [p]\)
12: \(m \leftarrow u \oplus \bar{u}\)
13: \text{return } m
- All 5 schemes heavily rely on polynomial multiplication
- We focus mainly on NIST level 3 parameter sets
All 5 schemes heavily rely on polynomial multiplication.

We focus mainly on NIST level 3 parameter sets.

This limits us to

- $q \in \{2^{11}, 2^{13}, 2^{14}\}$
- $n \in \{256, 701, 743, 1024\}$
Multiplying polynomials
Schoolbook multiplication \([N=1]\)

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

\[
\begin{array}{cccccccc}
a_5 & b_0 & a_4 & b_0 & a_3 & b_0 & a_2 & b_0 & a_1 & b_0 \\
a_5 & b_1 & a_4 & b_1 & a_3 & b_1 & a_2 & b_1 & a_1 & b_1 & a_0 & b_1 \\
a_5 & b_2 & a_4 & b_2 & a_3 & b_2 & a_2 & b_2 & a_1 & b_2 & a_0 & b_2 \\
a_5 & b_3 & a_4 & b_3 & a_3 & b_3 & a_2 & b_3 & a_1 & b_3 & a_0 & b_3 \\
a_5 & b_4 & a_4 & b_4 & a_3 & b_4 & a_2 & b_4 & a_1 & b_4 & a_0 & b_4 \\
a_5 & b_5 & a_4 & b_5 & a_3 & b_5 & a_2 & b_5 & a_1 & b_5 & a_0 & b_5 \\
\end{array}
\]

1 multiplication, 0 additions
Schoolbook multiplication \([N=2]\)

\[a = a_1x + a_0\]
\[b = b_1x + b_0\]

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

\[a = a_2x^2 + a_1x + a_0\]
\[b = b_2x^2 + b_1x + b_0\]

\[
\begin{array}{ccc}
    a_5b_0 & a_4b_0 & a_3b_0 \\
    a_5b_1 & a_4b_1 & a_3b_1 \\
    a_5b_2 & a_4b_2 & a_3b_2 \\
    a_5b_3 & a_4b_3 & a_3b_3 \\
    a_5b_4 & a_4b_4 & a_3b_4 \\
    a_5b_5 & a_4b_5 & a_3b_5 \\
\end{array}
\]

\[
\begin{array}{ccc}
    a_2b_0 & a_1b_0 & a_0b_0 \\
    a_2b_1 & a_1b_1 & a_0b_1 \\
    a_2b_2 & a_1b_2 & a_0b_2 \\
\end{array}
\]

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

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

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

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

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

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

\[
\begin{array}{ccccccc}
  & a_5 b_0 & & & & & \\
  & & a_5 b_1 & a_4 b_0 & a_3 b_0 & a_2 b_0 & a_1 b_0 & a_0 b_0 \\
  & & a_5 b_2 & a_4 b_1 & a_3 b_1 & a_2 b_1 & a_1 b_1 & a_0 b_1 \\
  & & a_5 b_3 & a_4 b_2 & a_3 b_2 & a_2 b_2 & a_1 b_2 & a_0 b_2 \\
  & & a_5 b_4 & a_4 b_3 & a_3 b_3 & a_2 b_3 & a_1 b_3 & a_0 b_3 \\
  & & a_5 b_5 & a_4 b_4 & a_3 b_4 & a_2 b_4 & a_1 b_4 & a_0 b_4 \\
\end{array}
\]

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

- Split inputs in half: \( a = a_1x^{n/2} + a_0, \quad b = b_1x^{n/2} + b_0 \)
Karatsuba’s method

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

$$c = a \cdot b = (a_1x^{n/2} + a_0) \cdot (b_1x^{n/2} + b_0)$$
Karatsuba’s method

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

\[
c = a \cdot b = (a_1x^{n/2} + a_0) \cdot (b_1x^{n/2} + b_0)
= a_1 \cdot b_1x^n + (a_1b_0 + a_0b_1)x^{n/2} + a_0 \cdot b_0
\]

- 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
Karatsuba’s method

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

\[
c = a \cdot b = (a_1x^{n/2} + a_0) \cdot (b_1x^{n/2} + b_0)
\]
\[
= a_1 \cdot b_1x^n + (a_1b_0 + a_0b_1)x^{n/2} + a_0 \cdot b_0
\]
\[
= a_1 \cdot b_1 + a_0 \cdot b_0 + ((a_1 + a_0) \cdot (b_0 + b_1) - a_1 \cdot b_1 - a_0 \cdot b_0)
\]
Karatsuba’s method

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

$$c = a \cdot b = (a_1x^{n/2} + a_0) \cdot (b_1x^{n/2} + b_0)$$
$$= a_1 \cdot b_1x^n + (a_1b_0 + a_0b_1)x^{n/2} + a_0 \cdot b_0$$
$$= a_1 \cdot b_1 + a_0 \cdot b_0 + ((a_1 + a_0) \cdot (b_0 + b_1) - a_1 \cdot b_1 - a_0 \cdot b_0)$$

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

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

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

- 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. $\mathbb{Z}_{2^{16}}$
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 $\rightarrow \mathbb{Z}_{2^{15}}$
  - 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
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 $n$
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 ← Rb_L · Rc_L + Rb_H · Rc_H</td>
</tr>
<tr>
<td>smuadx Ra, Rb, Rc</td>
<td>Ra ← Rb_L · Rc_H + Rb_H · Rc_L</td>
</tr>
<tr>
<td>smlad Ra, Rb, Rc, Rd</td>
<td>Ra ← Rb_L · Rc_L + Rb_H · Rc_H + Rd</td>
</tr>
<tr>
<td>smladx Ra, Rb, Rc, Rd</td>
<td>Ra ← Rb_L · Rc_H + Rb_H · Rc_L + Rd</td>
</tr>
</tbody>
</table>
Fast schoolbook multiplication \([N=2]\)
Fast schoolbook multiplication \([N=2]\)

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

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

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a_5 b_0)</td>
<td>(a_4 b_0)</td>
<td>(a_3 b_0)</td>
<td>(a_2 b_0)</td>
<td>(a_1 b_0)</td>
<td>(a_0 b_0)</td>
<td></td>
</tr>
<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>
<td></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>
<td></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>
<td></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>
<td></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>
<td></td>
</tr>
</tbody>
</table>

\(21\) multiplications instead of \(36\)
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 \(\rightarrow\) we fit 24 coefficients
- 78 multiplications instead of 144
We want to merge all.' But we don't have enough registers.
- We want to merge all
  - But we don’t have enough registers
Instead we perform 4 times 12x12
Fast schoolbook multiplication [N=36]

- Or 9 times 12x12
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>$a_4b_1$</td>
<td>$a_3b_1$</td>
<td>$a_2b_1$</td>
<td>$a_1b_1$</td>
<td>$a_0b_1$</td>
<td></td>
</tr>
<tr>
<td>$a_5b_2$</td>
<td>$a_4b_2$</td>
<td>$a_3b_2$</td>
<td>$a_2b_2$</td>
<td>$a_1b_2$</td>
<td>$a_0b_2$</td>
<td></td>
</tr>
<tr>
<td>$a_5b_3$</td>
<td>$a_4b_3$</td>
<td>$a_3b_3$</td>
<td>$a_2b_3$</td>
<td>$a_1b_3$</td>
<td>$a_0b_3$</td>
<td></td>
</tr>
<tr>
<td>$a_5b_4$</td>
<td>$a_4b_4$</td>
<td>$a_3b_4$</td>
<td>$a_2b_4$</td>
<td>$a_1b_4$</td>
<td>$a_0b_4$</td>
<td></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>$a_0b_5$</td>
<td></td>
</tr>
</tbody>
</table>

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

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

\[
\begin{array}{c}
a_5 b_0 & a_4 b_0 & a_3 b_0 & a_2 b_0 & a_1 b_0 & a_0 b_0 \\
| & | & | & | & | & | \\
\hline \\
\end{array}
\]

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

- $R0 = a_1|a_0$, $R1 = a_3|a_2$, $R2 = a_5|a_4$
- Then repack to $R3 = b_2|b_1$, $R4 = b_4|b_3$ 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 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

<table>
<thead>
<tr>
<th></th>
<th>(s_b)</th>
<th>cycles</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>K only</td>
<td>16</td>
<td>41 121</td>
<td>2 020</td>
</tr>
<tr>
<td>T3</td>
<td>11</td>
<td>41 225</td>
<td>3 480</td>
</tr>
<tr>
<td>T4</td>
<td>16</td>
<td>39 124</td>
<td>3 800</td>
</tr>
<tr>
<td>T4+T3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
No Toom-4+Toom-3 possible, because $q$ is too large

Toom-4 is optimal; but Karatsuba only and Toom-3 are close
Kindi \[ n = 256, \ q = 2^{14} \]

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

<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>41 121</td>
<td>2 020</td>
</tr>
<tr>
<td>T3</td>
<td>11</td>
<td>41 225</td>
<td>3 480</td>
</tr>
<tr>
<td>T4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>T4+T3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
No Toom-4 possible, because $q$ is too large

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

\[
\begin{array}{cccc}
701 & 704 & 702 \\
350 & 351 & T_4 \\
175 & 176 & T_3 \\
87 & 88 & 117 \\
43 & 44 & 58 \\
21 & 22 & 29 \\
10 & 11 & 14 \\
\end{array}
\]

<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>
NTRU-HRSS \( [n = 701, q = 2^{13}] \)

\[
\begin{array}{c}
701 \quad 704 \\
\uparrow \odot \uparrow \quad \uparrow \odot \uparrow \\
350 \quad 351 \quad T_4 \\
\downarrow \odot \downarrow \quad \downarrow \odot \downarrow \\
175 \quad 176 \\
\downarrow \odot \downarrow \quad \downarrow \odot \downarrow \\
87 \quad 88 \\
\downarrow \odot \downarrow \quad \downarrow \odot \downarrow \\
43 \quad 44 \\
\downarrow \odot \downarrow \quad \downarrow \odot \downarrow \\
21 \quad 22 \\
\downarrow \odot \downarrow \quad \downarrow \odot \downarrow \\
10 \quad 11
\end{array}
\]

\[
\begin{array}{c}
702 \\
\uparrow \\
351 \\
\downarrow \\
176 \\
\downarrow \\
175 \\
\downarrow \\
87 \\
\downarrow \\
43 \\
\downarrow \\
21 \\
\downarrow \\
10
\end{array}
\]

\[
\begin{array}{c}
234 \\
\downarrow \odot \\
117 \\
\downarrow \odot \\
58 \\
\downarrow \odot \\
29 \\
\downarrow \odot \\
14 \\
\downarrow \odot
\end{array}
\]

\[
\begin{array}{c}
T_3 \\
\downarrow \odot \\
T_4
\end{array}
\]

\[
\begin{array}{c|c|c|c}
& sb & cycles & stack \\
K only & 11 & 230 132 & 5 676 \\
T3 & 15 & 217 436 & 9 384 \\
T4 & 11 & 182 129 & 10 596 \\
T4 + T3 & - & - & - \\
\end{array}
\]
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><strong>12</strong></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
We could use Toom-4+Toom-3 here

but Toom-4 is faster (again quite close)
RLizard \[n = 1024, q = 2^{11}\]

\[
\begin{array}{c}
1026 \\
\downarrow_{T3} \\
342 \\
\downarrow_{K} \\
171 \\
\downarrow_{K} \\
85 \\
\downarrow_{K} \\
42 \\
\downarrow_{K} \\
21 \\
\downarrow_{K} \\
10
\end{array}
\quad
\begin{array}{c}
1024 \\
\downarrow_{K} \\
512 \\
\downarrow_{K} \\
256 \\
\downarrow_{K} \\
128 \\
\downarrow_{K} \\
64 \\
\downarrow_{K} \\
32 \\
\downarrow_{K} \\
16
\end{array}
\]

<table>
<thead>
<tr>
<th></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>T4</td>
<td>16</td>
<td>313 744</td>
<td>15 344</td>
</tr>
<tr>
<td>T4+T3</td>
<td>11</td>
<td>315 788</td>
<td>16 816</td>
</tr>
</tbody>
</table>
**RLizard \[n = 1024, q = 2^{11}\]**

<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>T4</td>
<td>16</td>
<td>313 744</td>
<td>15 344</td>
</tr>
<tr>
<td>T4 + T3</td>
<td>11</td>
<td>315 788</td>
<td>16 816</td>
</tr>
</tbody>
</table>

Diagram:

```
1026
↓_{T3}
342
↓_{K}
171
K\↓_{K}  K\↓_{K}
85  86
↓_{K}  K↓_{K}
42  43
↓_{K}  K↓_{K}
21  22
↓_{K}  K↓_{K}
10  11
```

```
1024
K\↓_{T4}
512
K\↓_{T3}
256
K↓_{K}
128
K↓_{K}
64
K↓_{K}
32
K↓_{K}
16
```
<table>
<thead>
<tr>
<th>Implementation</th>
<th>Clock Cycles</th>
<th>Stack Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K:</td>
<td>E:</td>
</tr>
<tr>
<td>Saber</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>6530k</td>
<td>8684k</td>
</tr>
<tr>
<td>[KMRV18]</td>
<td>1147k</td>
<td>1444k</td>
</tr>
<tr>
<td>This work</td>
<td>949k</td>
<td>1232k</td>
</tr>
<tr>
<td>Kindi-256-3-4-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>21794k</td>
<td>28176k</td>
</tr>
<tr>
<td>This work</td>
<td>1010k</td>
<td>1365k</td>
</tr>
<tr>
<td>NTRU-HRSS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>205156k</td>
<td>5166k</td>
</tr>
<tr>
<td>This work</td>
<td>161790k</td>
<td>432k</td>
</tr>
<tr>
<td>NTRU-KEM-743</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>59815k</td>
<td>7540k</td>
</tr>
<tr>
<td>This work</td>
<td>5663k</td>
<td>1655k</td>
</tr>
<tr>
<td>RLizard-1024</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>26423k</td>
<td>32156k</td>
</tr>
<tr>
<td>This work</td>
<td>537k</td>
<td>1358k</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
### Results: Saber

<table>
<thead>
<tr>
<th>implementation</th>
<th>clock cycles</th>
<th>stack usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K: 6 530k</td>
<td>K: 12 616k</td>
</tr>
<tr>
<td>Reference</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></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></td>
<td></td>
</tr>
<tr>
<td></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></td>
<td></td>
</tr>
<tr>
<td></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
- 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]
HILA5 recently merged with Round2 → Round5
Optimised for the M4 at CARDIS’18 [SBGM^+18]

“... Round5 offers not only the shortest key and ciphertext sizes among Lattice-based candidates, but also has leading performance and implementation size characteristics.”
## NIST Security level 1

<table>
<thead>
<tr>
<th>Method</th>
<th>Platform</th>
<th>K (clock cycles)</th>
<th>E (clock cycles)</th>
<th>D (clock cycles)</th>
</tr>
</thead>
<tbody>
<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>

- 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>K</td>
<td>E</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>NTRU-HRSS</td>
<td>This work</td>
<td>161790k</td>
<td>432k</td>
<td>863k</td>
</tr>
<tr>
<td>R5ND_1PKEb</td>
<td>[SBGM+18]</td>
<td>658k</td>
<td>984k</td>
<td>1265k</td>
</tr>
</tbody>
</table>

### NIST Security level 3

<table>
<thead>
<tr>
<th></th>
<th>K</th>
<th>E</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saber</td>
<td>This work</td>
<td>949k</td>
<td>1232k</td>
</tr>
<tr>
<td>R5ND_3PKEb</td>
<td>[SBGM+18]</td>
<td>1032k</td>
<td>1510k</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?
### Profiling Optimised Implementations

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

- Fastest PQC scheme implementation on the Cortex-M4

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
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 $q = 2^m \leq 2^{16}$. 

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

- Highly non-constant time
- Dynamic memory allocations (#free’s != #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
A note on PQC code quality

- 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
A note on PQC code quality

- 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
- More work required
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.**