Radbound University

Post-quantum crypto on ARM Cortex-M

Peter Schwabe
peter@cryptojedi.org
https://cryptojedi.org

January 23, 2019

PQCRYPTO

- 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

PQCRYPTO – WP1

- 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.

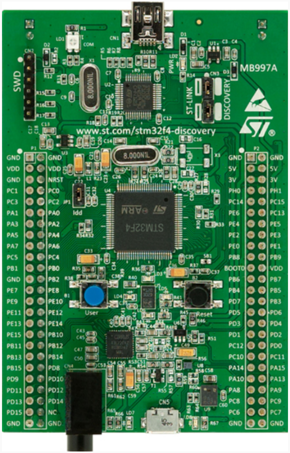
PQCRYPTO – WP1

- 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

PQCRYPTO – WP1

- 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:

```
python3 test.py
```

- Generate testvectors, compare for consistency (also with host):

```
python3 testvectors.py
```

- Run speed and stack benchmarks:

```
python3 benchmarks.py
```

- Easy to evaluate only subset of schemes, e.g.:

```
python3 test.py newhope1024cca sphincs-shake256-128s
```

Initial pqm4 results KEM/PKE

BIG QUAKE	?
BIKE	?
Classic McEliece	✗
CRYSTALS-Kyber	✓
DAGS	?
FrodoKEM	✓
KINDI	✓
NewHope	✓
NTRU-HRSS-KEM	✓
NTRU Prime	✓
Post-quantum RSA-Encryption	✗
Ramstake	✗(?)
SABER	✓
SIKE	✓

Initial `pqm4` results signatures

CRYSTALS-Dilithium	✓
GUI	✗
LUOV	?
MQDSS	✗(?)
Picnic	✗
Post-quantum RSA-Signature	✗
qTESLA	✓
Rainbow	? (probably no)
SPHINCS+	✓

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

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 $\mathbf{A} \in \mathbb{Z}_q^{k \times \ell}$
- Given "noise distribution" χ
- Given samples $\mathbf{As} + \mathbf{e}$, with $\mathbf{e} \leftarrow \chi$

Learning with errors (LWE)

- Given uniform $\mathbf{A} \in \mathbb{Z}_q^{k \times \ell}$
- Given "noise distribution" χ
- Given samples $\mathbf{As} + \mathbf{e}$, with $\mathbf{e} \leftarrow \chi$
- Search version: find $\mathbf{s}$
- Decision version: distinguish from uniform random

Learning with errors (LWE)

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

Learning with rounding (LWR)

- Given uniform $\mathbf{A} \in \mathbb{Z}_q^{k \times \ell}$
- Given samples $\lceil \mathbf{A}\mathbf{s} \rfloor_p$, with $p < q$

Learning with rounding (LWR)

- Given uniform $\mathbf{A} \in \mathbb{Z}_q^{k \times \ell}$
- Given samples $\lceil \mathbf{As} \rfloor_p$, with $p < q$
- Search version: find $\mathbf{s}$
- Decision version: distinguish from uniform random
- Structured lattices: work in $\mathbb{Z}_q[x]/f$

Lattice-based KEMs – the basic idea

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

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

- Secret and noise $\mathbf{s}, \mathbf{s}', \mathbf{e}, \mathbf{e}'$ are small
- $\mathbf{t}$ and $\mathbf{t}'$ are *approximately* the same

Lattice-based KEMs submitted to NIST

- 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

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

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. LPR (“quotient” vs. “product”)

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"

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:

$$
\begin{aligned}
& (a_\ell + X^k a_h) \cdot (b_\ell + X^k b_h) \\
= \; & a_\ell b_\ell + X^k (a_\ell b_h + a_h b_\ell) + X^n a_h b_h \\
= \; & a_\ell b_\ell + X^k ((a_\ell + a_h)(b_\ell + b_h) - a_\ell b_\ell - a_h b_h) + X^n a_h b_h
\end{aligned}
$$

- Recursive application yields complexity $\Theta(n^{\log_2 3})$

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:

$$\begin{aligned}
&(a_\ell + X^k a_h) \cdot (b_\ell + X^k b_h) \\
&= a_\ell b_\ell + X^k(a_\ell b_h + a_h b_\ell) + X^n a_h b_h \\
&= a_\ell b_\ell + X^k((a_\ell + a_h)(b_\ell + b_h) - a_\ell b_\ell - a_h b_h) + X^n a_h b_h
\end{aligned}$$

- Recursive application yields complexity $\Theta(n^{\log_2 3})$
- Generalization: Toom-Cook
  - Toom-3: split into 5 multiplications of 1/3 size
  - Toom-4: split into 7 multiplications of 1/4 size
- Approach: Evaluate, multiply, interpolate

Initial observations

- Karatsuba/Toom is asymptotically faster, but isn't for "small" polynomials

Initial observations

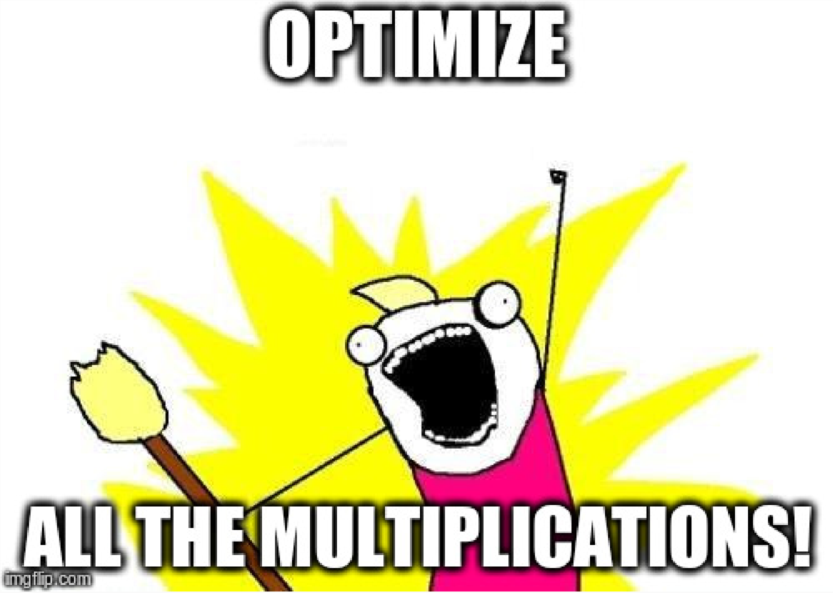
- Karatsuba/Toom is asymptotically faster, but isn't for "small" polynomials
- Toom-3 needs division by 2, loses 1 bit of precision
- Toom-4 needs division by 8, loses 3 bits of precision
- This limits recursive application when using 16-bit integers
- Can use Toom-4 only for $q \leq 2^{13}$

Initial observations

- Karatsuba/Toom is asymptotically faster, but isn't for "small" polynomials
- Toom-3 needs division by 2, loses 1 bit of precision
- Toom-4 needs division by 8, loses 3 bits of precision
- This limits recursive application when using 16-bit integers
- Can use Toom-4 only for $q \leq 2^{13}$
- Karmakar, Bermudo Mera, Sinha Roy, Verbauwhede (CHES 2018):
  - Optimize Saber, $q = 2^{13}$, $n = 256$
  - Use Toom-4 + two levels of Karatsuba
  - Optimized 16-coefficient schoolbook multiplication

Initial observations

- Karatsuba/Toom is asymptotically faster, but isn't for "small" polynomials
- Toom-3 needs division by 2, loses 1 bit of precision
- Toom-4 needs division by 8, loses 3 bits of precision
- This limits recursive application when using 16-bit integers
- Can use Toom-4 only for $q \leq 2^{13}$
- Karmakar, Bermudo Mera, Sinha Roy, Verbauwhede (CHES 2018):
  - Optimize Saber, $q = 2^{13}, n = 256$
  - Use Toom-4 + two levels of Karatsuba
  - Optimized 16-coefficient schoolbook multiplication
- **Is this the best approach? How about other values of q and n?**


OPTIMIZE
ALL THE MULTIPLICATIONS!
imgflip.com

Our approach

- Generate optimized assembly for Karatsuba/Toom
- Use Python scripts, receive as input n and q
- Hand-optimize "small" schoolbook multiplications
- 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

instruction	semantics
`smuad Ra, Rb, Rc`	$\mathtt{Ra} \leftarrow \mathtt{Rb}_\mathtt{L} \cdot \mathtt{Rc}_\mathtt{L} + \mathtt{Rb}_\mathtt{H} \cdot \mathtt{Rc}_\mathtt{H}$
`smuadx Ra, Rb, Rc`	$\mathtt{Ra} \leftarrow \mathtt{Rb}_\mathtt{L} \cdot \mathtt{Rc}_\mathtt{H} + \mathtt{Rb}_\mathtt{H} \cdot \mathtt{Rc}_\mathtt{L}$
`smlad Ra, Rb, Rc, Rd`	$\mathtt{Ra} \leftarrow \mathtt{Rb}_\mathtt{L} \cdot \mathtt{Rc}_\mathtt{L} + \mathtt{Rb}_\mathtt{H} \cdot \mathtt{Rc}_\mathtt{H} + \mathtt{Rd}$
`smladx Ra, Rb, Rc, Rd`	$\mathtt{Ra} \leftarrow \mathtt{Rb}_\mathtt{L} \cdot \mathtt{Rc}_\mathtt{H} + \mathtt{Rb}_\mathtt{H} \cdot \mathtt{Rc}_\mathtt{L} + \mathtt{Rd}$

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

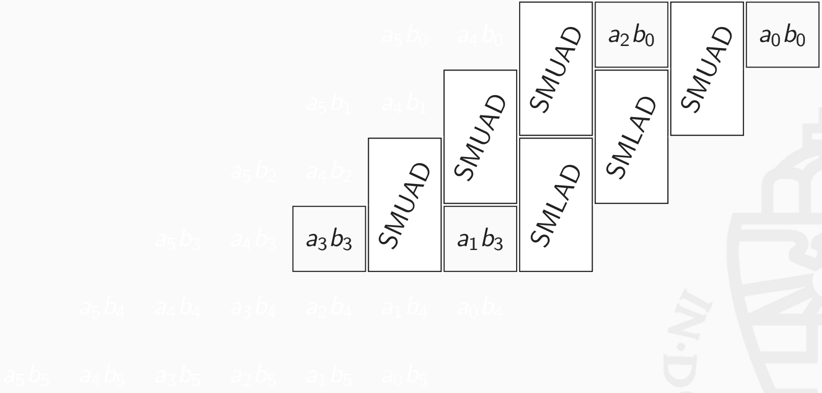
Slide credit to Matthias Kannwischer

Fast schoolbook multiplication [N=4]

					$a_5 b_0$	$a_4 b_0$	$a_3 b_0$	$a_2 b_0$	$a_1 b_0$	$a_0 b_0$
				$a_5 b_1$	$a_4 b_1$	$a_3 b_1$	$a_2 b_1$	$a_1 b_1$	$a_0 b_1$	
			$a_5 b_2$	$a_4 b_2$	$a_3 b_2$	$a_2 b_2$	$a_1 b_2$	$a_0 b_2$		
		$a_5 b_3$	$a_4 b_3$	$a_3 b_3$	$a_2 b_3$	$a_1 b_3$	$a_0 b_3$			
	$a_5 b_4$	$a_4 b_4$	$a_3 b_4$	$a_2 b_4$	$a_1 b_4$	$a_0 b_4$				
$a_5 b_5$	$a_4 b_5$	$a_3 b_5$	$a_2 b_5$	$a_1 b_5$	$a_0 b_5$					

Slide credit to Matthias Kannwischer

Fast schoolbook multiplication [N=4]



- 10 multiplications instead of 16

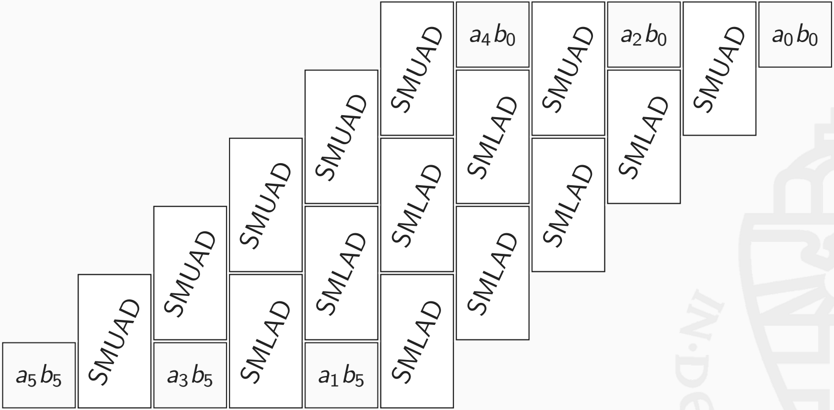
Slide credit to Matthias Kannwischer

Fast schoolbook multiplication [N=6]

					$a_5 b_0$	$a_4 b_0$	$a_3 b_0$	$a_2 b_0$	$a_1 b_0$	$a_0 b_0$
				$a_5 b_1$	$a_4 b_1$	$a_3 b_1$	$a_2 b_1$	$a_1 b_1$	$a_0 b_1$	
			$a_5 b_2$	$a_4 b_2$	$a_3 b_2$	$a_2 b_2$	$a_1 b_2$	$a_0 b_2$		
		$a_5 b_3$	$a_4 b_3$	$a_3 b_3$	$a_2 b_3$	$a_1 b_3$	$a_0 b_3$			
	$a_5 b_4$	$a_4 b_4$	$a_3 b_4$	$a_2 b_4$	$a_1 b_4$	$a_0 b_4$				
$a_5 b_5$	$a_4 b_5$	$a_3 b_5$	$a_2 b_5$	$a_1 b_5$	$a_0 b_5$					

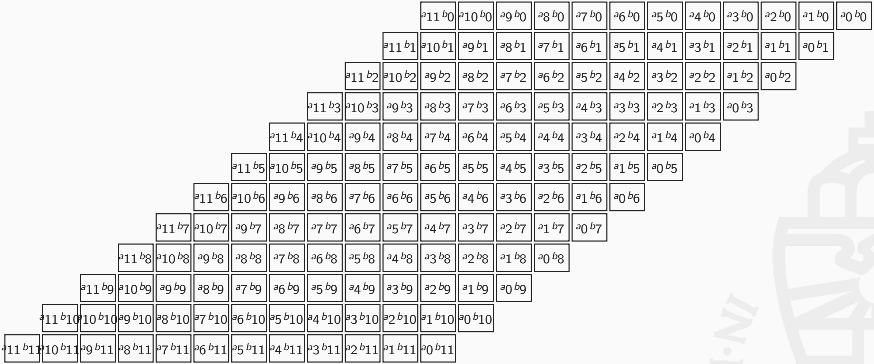
Slide credit to Matthias Kannwischer

Fast schoolbook multiplication [N=6]



- 21 multiplications instead of 36

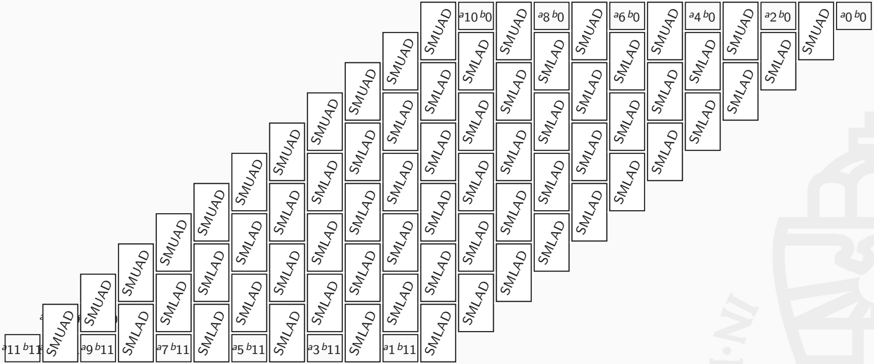
Slide credit to Matthias Kannwischer



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

Slide credit to Matthias Kannwischer

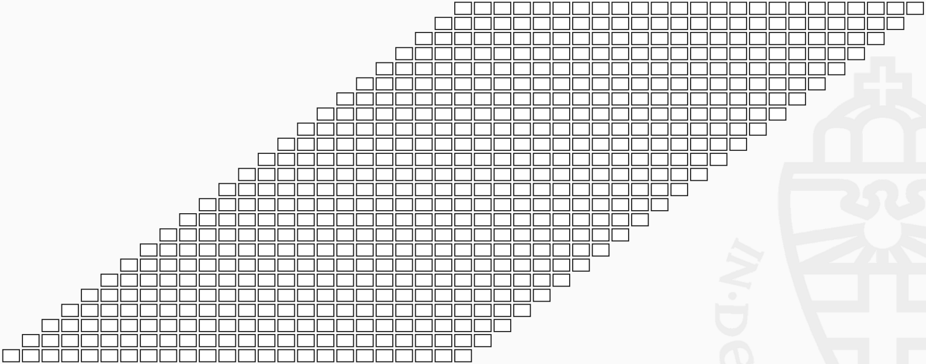
Fast schoolbook multiplication [N=12]



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

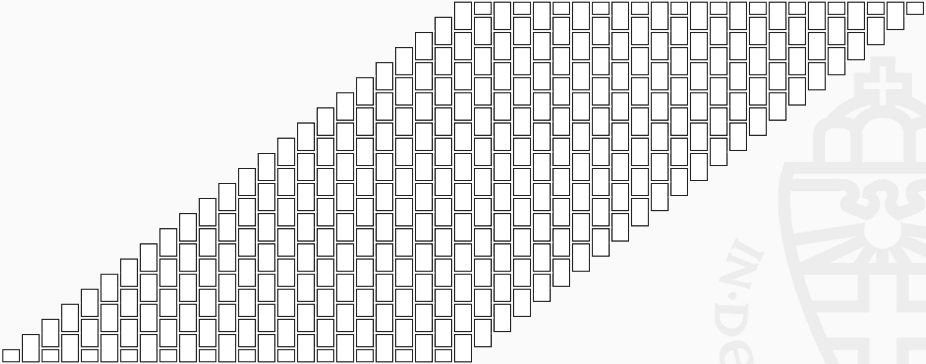
Slide credit to Matthias Kannwischer

Fast schoolbook multiplication [N=24]



Slide credit to Matthias Kannwischer

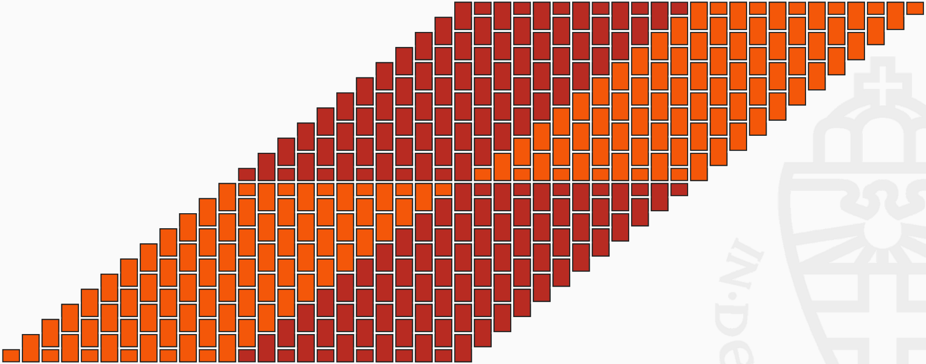
Fast schoolbook multiplication [N=24]



- We want to merge all, but not enough registers

Slide credit to Matthias Kannwischer

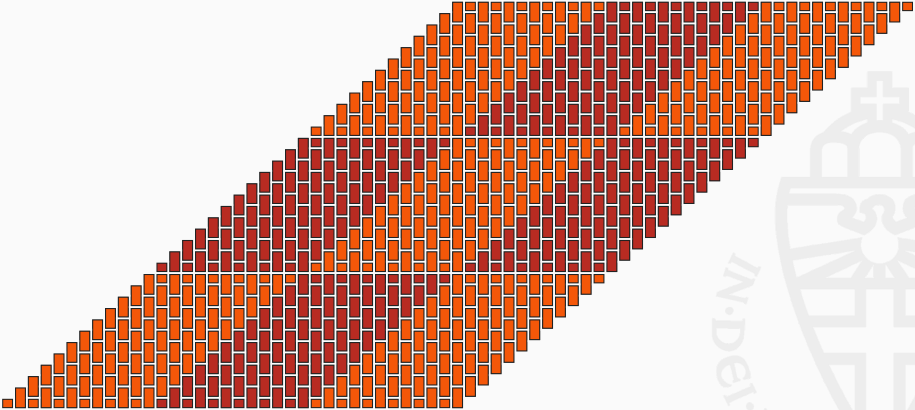
Fast schoolbook multiplication [N=24]



- 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

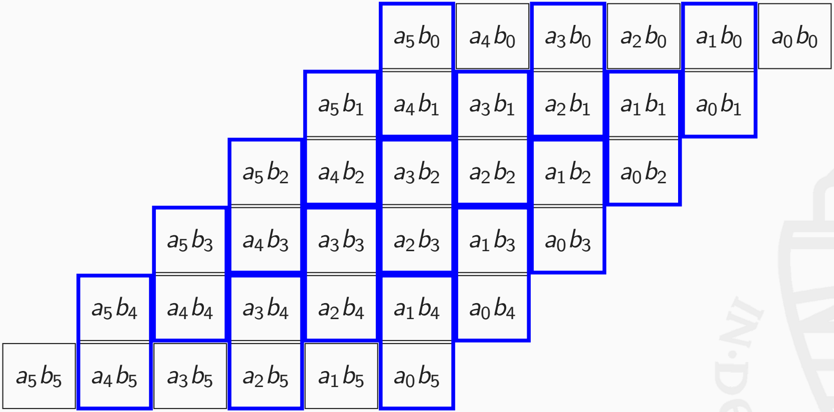
Fast schoolbook multiplication: Reduce repacks

					$a_5 b_0$	$a_4 b_0$	$a_3 b_0$	$a_2 b_0$	$a_1 b_0$	$a_0 b_0$
				$a_5 b_1$	$a_4 b_1$	$a_3 b_1$	$a_2 b_1$	$a_1 b_1$	$a_0 b_1$	
			$a_5 b_2$	$a_4 b_2$	$a_3 b_2$	$a_2 b_2$	$a_1 b_2$	$a_0 b_2$		
		$a_5 b_3$	$a_4 b_3$	$a_3 b_3$	$a_2 b_3$	$a_1 b_3$	$a_0 b_3$			
	$a_5 b_4$	$a_4 b_4$	$a_3 b_4$	$a_2 b_4$	$a_1 b_4$	$a_0 b_4$				
$a_5 b_5$	$a_4 b_5$	$a_3 b_5$	$a_2 b_5$	$a_1 b_5$	$a_0 b_5$					

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

Slide credit to Matthias Kannwischer

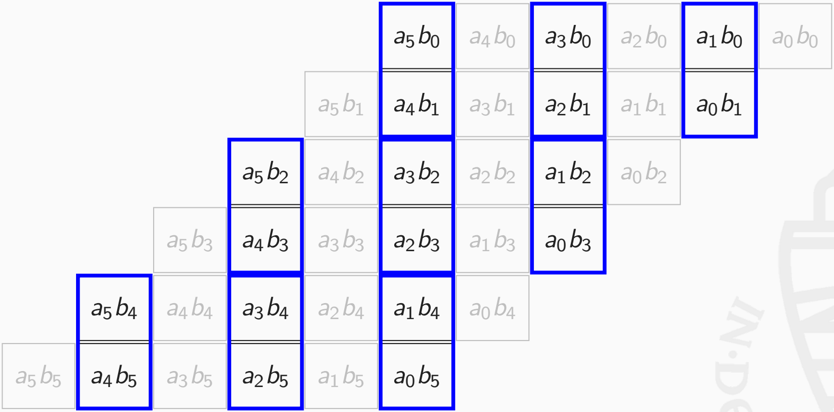
Fast schoolbook multiplication: Reduce repacks



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

Slide credit to Matthias Kannwischer

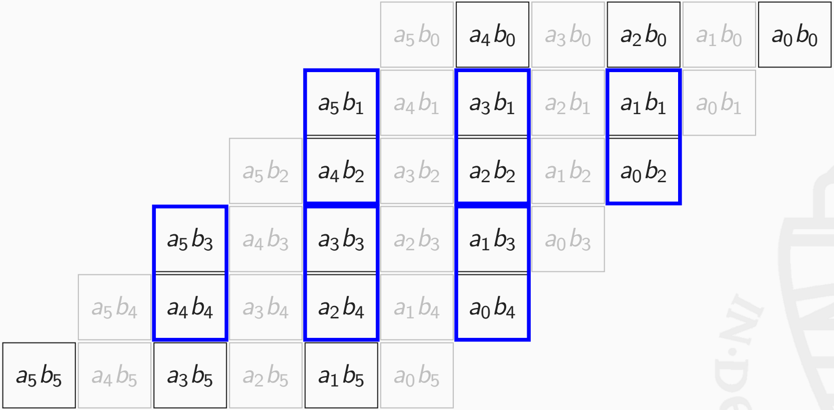
Fast schoolbook multiplication: Reduce repacks



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

Slide credit to Matthias Kannwischer

Fast schoolbook multiplication: Reduce repacks



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

Slide credit to Matthias Kannwischer

Multiplication results

	approach	"small"	cycles	stack
Saber ($n = 256,\ q = 2^{13}$)	Karatsuba only	16	41 121	2 020
	Toom-3	11	41 225	3 480
	Toom-4	**16**	**39 124**	**3 800**
	Toom-4 + Toom-3	-	-	-
Kindi-256-3-4-2 ($n = 256,\ q = 2^{14}$)	**Karatsuba only**	**16**	**41 121**	**2 020**
	Toom-3	11	41 225	3 480
	Toom-4	-	-	-
	Toom-4 + Toom-3	-	-	-
NTRU-HRSS ($n = 701,\ q = 2^{13}$)	Karatsuba only	11	230 132	5 676
	Toom-3	15	217 436	9 384
	Toom-4	**11**	**182 129**	**10 596**
	Toom-4 + Toom-3	-	-	-
NTRU-KEM-743 ($n = 743,\ q = 2^{11}$)	Karatsuba only	12	247 489	6 012
	Toom-3	16	219 061	9 920
	Toom-4	**12**	**196 940**	**11 208**
	Toom-4 + Toom-3	16	197 227	12 152
RLizard-1024 ($n = 1024,\ q = 2^{11}$)	Karatsuba only	16	400 810	8 188
	Toom-3	11	360 589	13 756
	Toom-4	**16**	**313 744**	**15 344**
	Toom-4 + Toom-3	11	315 788	16 816

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!**

KEM results

	implementation	clock cycles	stack usage
Saber	Reference	**K:** 6 530k **E:** 8 684k **D:** 10 581k	**K:** 12 616 **E:** 14 896 **D:** 15 992
	[KBSV18]	**K:** 1 147k **E:** 1 444k **D:** 1 543k	**K:** 13 883 **E:** 16 667 **D:** 17 763
	This work	**K:** 949k **E:** 1 232k **D:** 1 260k	**K:** 13 248 **E:** 15 528 **D:** 16 624
Kindi-256-3-4-2	Reference	**K:** 21 794k **E:** 28 176k **D:** 37 129k	**K:** 59 864 **E:** 71 000 **D:** 84 096
	This work	**K:** 1 010k **E:** 1 365k **D:** 1 563k	**K:** 44 264 **E:** 55 392 **D:** 64 376

KEM results

	implementation	clock cycles		stack usage	
NTRU-HRSS	Reference	K:	205 156k	K:	10 020
		E:	5 166k	E:	8 956
		D:	15 067k	D:	10 204
	This work	K:	161 790k	K:	23 396
		E:	432k	E:	19 492
		D:	863k	D:	22 140
NTRU-KEM-743	Reference	K:	59 815k	K:	14 148
		E:	7 540k	E:	13 372
		D:	14 229k	D:	18 036
	This work	K:	5 663k	K:	25 320
		E:	1 655k	E:	23 808
		D:	1 904k	D:	28 472
RLizard-1024	Reference	K:	26 423k	K:	4 272
		E:	32 156k	E:	10 532
		D:	53 181k	D:	12 636
	This work	K:	537k	K:	27 720
		E:	1 358k	E:	33 328
		D:	1 740k	D:	35 448

Coming back to NISTPQC code quality

- 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)

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)
- Idea: collect "clean" implementations **once**

PQClean

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

PQClean

- Joint work with
  Matthias Kannwischer, Joost Rijneveld, Douglas Stebila, Thom Wiggers
- GitHub repo with extensive CI to ensure “clean” implementations
- Goal: 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?

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!

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

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” 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)

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 namespaced 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 inplace
- 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=...)`)

Resources online

- 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