Radboud University

Compact Dilithium on Cortex M3 and Cortex M4

Denisa Greconici[a] Matthias Kannwischer[a,b] Daan Sprenkels[a]

17 September 2020

[a] Institute for Computing and Information Sciences – Digital Security
Radboud University Nijmegen

[b] Academia Sinica, Taiwan

Table of contents

Introduction

Dilithium

Dilithium

- Signature scheme
- Part of CRYSTALS (with Kyber)
- One of the 3rd round finalists

Dilithium

- Signature scheme
- Part of CRYSTALS (with Kyber)
- One of the 3rd round finalists
- Fiat-Shamir with aborts
- Module-LWE and Module-SIS

Dilithium

- Signature scheme
- Part of CRYSTALS (with Kyber)
- One of the 3rd round finalists
- Fiat-Shamir with aborts
- Module-LWE and Module-SIS
- Small keys and signatures
- Operates in the polynomial ring $\mathbb{R}_q = \mathbb{Z}_q[X]/(X^{256} + 1)$, with $q = 8380417$
  $\Rightarrow$ Allows efficient polynomial multiplication with NTT

The Number-Theoretic Transform (NTT)

The Number-Theoretic Transform (NTT)

- Fast Fourier Transform (FFT) in finite field
- Let $g = g_0 + g_1 X + ... + g_{n-1} X^{n-1}$, polynomial in $\mathbb{R}_q$
- Representation of polynomial g:
  - By its coefficients: $g_0, g_1 ... g_{n-1}$
  - By evaluating g at the powers of the n'th primitive root of unity: $g(\omega^0), g(\omega^1) ... g(\omega^{n-1})$

The Number-Theoretic Transform (NTT)

- Fast Fourier Transform (FFT) in finite field
- Let $g = g_0 + g_1 X + ... + g_{n-1} X^{n-1}$, polynomial in $\mathbb{R}_q$
- Representation of polynomial g:
  - By its coefficients: $g_0, g_1 ... g_{n-1}$
  - By evaluating g at the powers of the n'th primitive root of unity: $g(\omega^0), g(\omega^1) ... g(\omega^{n-1})$
- Formal definition of the NTT in Dilithium
  - $\hat{g} = NTT(g) = \sum_{i=0}^{n-1} \hat{g}_i X^i, \quad with \quad \hat{g}_i = \sum_{j=0}^{n-1} \psi^j g_j \omega^{ij}; \quad and$
  - $g = INTT(\hat{g}) = \sum_{i=0}^{n-1} g_i X^i, \quad with \quad g_i = n^{-1} \psi^{-i} \sum_{j=0}^{n-1} \hat{g}_j \omega^{-ij}.$
- Polynomial Multiplication in $\mathbb{R}_q$
  $\mathbf{a} \cdot \mathbf{b} = \mathbf{INTT(NTT(a) \circ NTT(b))}$

Dilithium simplified

Dilithium simplified

<u>Gen</u>

$\mathbf{A} \leftarrow R_q^{k \times \ell}$
$(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^\ell \times S_\eta^k$
$\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
return $(pk = (\mathbf{A}, \mathbf{t}), sk = (\mathbf{A}, \mathbf{t}, \mathbf{s}_1, \mathbf{s}_2))$

<u>Sign(sk, M)</u>

$\mathbf{z} := \perp$
while $\mathbf{z} = \perp$ do
  $\mathbf{y} \leftarrow S_{\gamma_1 - 1}^\ell$
  $\mathbf{w}_1 := \mathsf{HighBits}(\mathbf{A}\mathbf{y}, 2\gamma_2)$
  $c \in B_{60} := \mathsf{H}(M \parallel \mathbf{w}_1)$
  $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
  if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathsf{LowBits}(\mathbf{A}\mathbf{y} - c\mathbf{s}_2, 2\gamma_2)\|_\infty \geq \gamma_2 - \beta$, then $\mathbf{z} := \perp$
return $\sigma = (\mathbf{z}, c)$

<u>Verify$(pk, M, \sigma = (\mathbf{z}, c))$</u>

$\mathbf{w}_1' := \mathsf{HighBits}(\mathbf{A}\mathbf{z} - c\mathbf{t}, 2\gamma_2)$
if return $[\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!]$ **and** $[\![c = \mathsf{H}(M \parallel \mathbf{w}_1')]\!]$

Target platforms

Target platforms

- **Arm Cortex M4**(STM32F407-DISCOVERY)

- **Arm Cortex M3** (AtmelSAM3X8E)

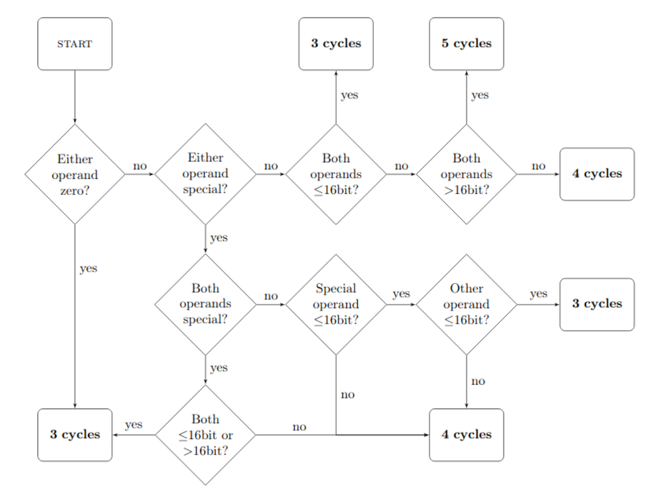
Target platforms

- **Arm Cortex M4**(STM32F407-DISCOVERY)
  - NIST choice for PQC
  - 32-bit, ARMv7e-M
  - 1 MiB ROM, 196 KB RAM, 168 MHz
  - 32-bit multiplications in **1 cycle** (UMULL, SMULL, UMLAL, SMLAL)
- **Arm Cortex M3** (AtmelSAM3X8E)

Target platforms

- **Arm Cortex M4**(STM32F407-DISCOVERY)
  - NIST choice for PQC
  - 32-bit, ARMv7e-M
  - 1 MiB ROM, 196 KB RAM, 168 MHz
  - 32-bit multiplications in **1 cycle** (UMULL, SMULL, UMLAL, SMLAL)
- **Arm Cortex M3** (AtmelSAM3X8E)
  - Arduino Due
  - 32-bit, ARMv7-M
  - 512 KiB Flash, 96 KB RAM, 84 MHz
  - **Variable time 32-bit multiplications!**

UMULL on M3



[1] Based on the Master thesis of [dG15].

Constant time multiplications on Cortex-M3

Overcoming the non-constant time multiplications

Overcoming the non-constant time multiplications

- Variable time 64-bit multiplications
  - But, 16-bit multipliers are constant time
    MUL – 1 cycle; MLA, MLS – 2 cycles

Overcoming the non-constant time multiplications

- Variable time 64-bit multiplications
  - But, 16-bit multipliers are constant time
    MUL – 1 cycle; MLA, MLS – 2 cycles
- Our solution: use 32-bit multipliers
  $\Rightarrow$ represent the 64-bit values in radix 2^{16}

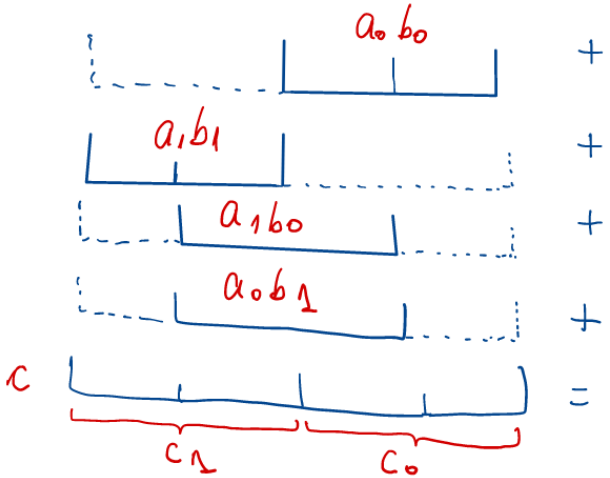
Overcoming the non-constant time multiplications

- Variable time 64-bit multiplications
  - But, 16-bit multipliers are constant time
    MUL – 1 cycle; MLA, MLS – 2 cycles
- Our solution: use 32-bit multipliers
  $\Rightarrow$ represent the 64-bit values in radix 2^{16}
  - Let $a = 2^{16}a_1 + a_0$ and $b = 2^{16}b_1 + b_0$
    with $0 \leq a_0, b_0 < 2^{16}$ and $-2^{15} \leq a_1, b_1 < 2^{15}$

Overcoming the non-constant time multiplications

- Variable time 64-bit multiplications
  - But, 16-bit multipliers are constant time
    MUL – 1 cycle; MLA, MLS – 2 cycles
- Our solution: use 32-bit multipliers
  $\Rightarrow$ represent the 64-bit values in radix 2^{16}
  - Let $a = 2^{16}a_1 + a_0$ and $b = 2^{16}b_1 + b_0$
    with $0 \leq a_0, b_0 < 2^{16}$ and $-2^{15} \leq a_1, b_1 < 2^{15}$
  - Then $ab = 2^{32}a_1b_1 + 2^{16}(a_0b_1 + a_1b_0) + a_0b_0$,
    with $-2^{31} \leq a_ib_j < 2^{31}$

Schoolbook multiplication



Optimizing performance

Optimizing performance

(1) Applying the CRT

(2) {Unsigned => Signed} representation

(3) Merging layer

$$c := a \cdot b$$

$$\hat{a} := NTT(a)$$

$$\hat{b} := NTT(b)$$

$$\hat{c} := \hat{a} \cdot \hat{b}$$

$$c := NTT^{-1}(\hat{c})$$

[1] Based on [BCLv19].

Applying the CRT[1]

$$c := a \cdot b$$

$$a_i := a \bmod q_i$$

$$b_i := b \bmod q_i$$

$$c_i := NTT^{-1}(NTT(a_i) \circ NTT(b_i))$$

$$c := CRT(c_1, c_2 \ldots c_k)$$

[1] Based on [BCLv19].

Applying the CRT

Applying the CRT

- NTT has to work in $\mathbb{Z}_{q_i}/(X^{256}+1)$
  $\Rightarrow$ choose q_i NTT primes

Applying the CRT

- NTT has to work in $\mathbb{Z}_{q_i}/(X^{256}+1)$
  $\Rightarrow$ choose q_i NTT primes

- $\prod_i q_i$ must be larger than coefficients in c!
- For Dilithium, need to split into 4 polynomials mod q_i

Applying the CRT

- NTT has to work in $\mathbb{Z}_{q_i}/(X^{256}+1)$
  $\Rightarrow$ choose q_i NTT primes

- $\prod_i q_i$ must be larger than coefficients in c!
- For Dilithium, need to split into 4 polynomials mod q_i

- Unfortunately, this is slower than doing schoolbook
- But it might be useful for other platforms :)

{Unsigned => Signed} representation

- Unsigned subtraction $a - b$ overflows if $a < b$

{Unsigned => Signed} representation

- Unsigned subtraction $a - b$ overflows if $a < b$
- All subtractions are $a - b \equiv (a + Nq) - b$ to mitigate this

{Unsigned => Signed} representation

- Unsigned subtraction $a - b$ overflows if $a < b$
- All subtractions are $a - b \equiv (a + Nq) - b$ to mitigate this
  - Extra addition
  - Numbers grow faster $\Rightarrow$ more reductions needed

{Unsigned => Signed} representation

- Unsigned subtraction $a - b$ overflows if $a < b$
- All subtractions are $a - b \equiv (a + Nq) - b$ to mitigate this
  - Extra addition
  - Numbers grow faster $\Rightarrow$ more reductions needed

- Signed representation is better! :)

{Unsigned => Signed} representation

- Unsigned subtraction $a - b$ overflows if $a < b$
- All subtractions are $a - b \equiv (a + Nq) - b$ to mitigate this
  - Extra addition
  - Numbers grow faster $\Rightarrow$ more reductions needed

- Signed representation is better! :)
  - No extra addition
  - Numbers grow less $\Rightarrow$ less reductions

Merging layers

- NTT (= FFT) recurses a binary tree

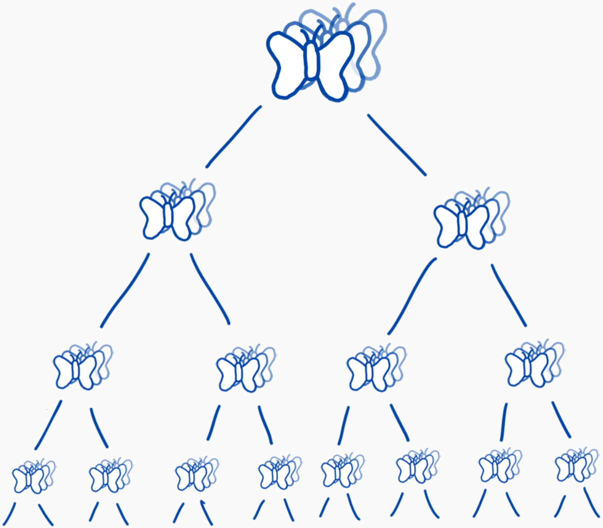
Merging layers

- NTT (= FFT) recurses a binary tree
- Depth first: Many reloads of twiddle factors
- Breadth first: Many loads/spills of coefficients

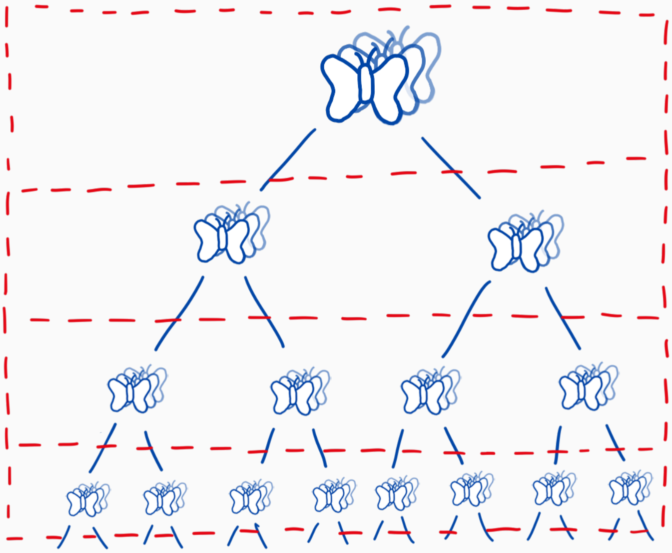
Merging layers

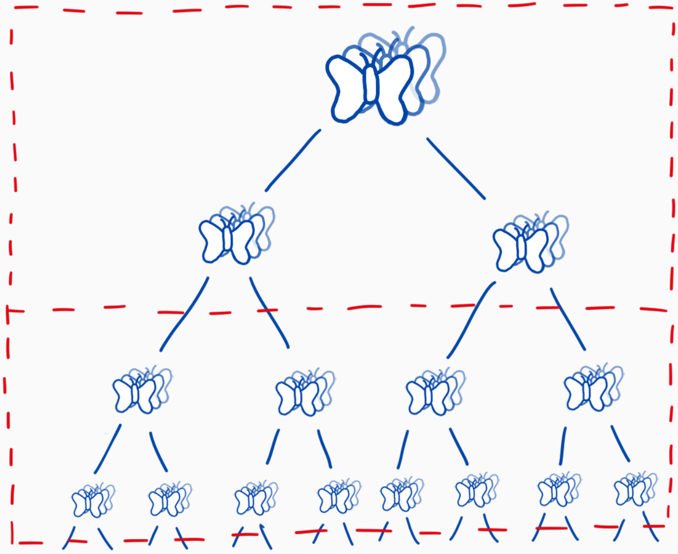
- NTT (= FFT) recurses a binary tree
- Depth first: Many reloads of twiddle factors
- Breadth first: Many loads/spills of coefficients
- Go for hybrid approach, i.e., *merging layers*

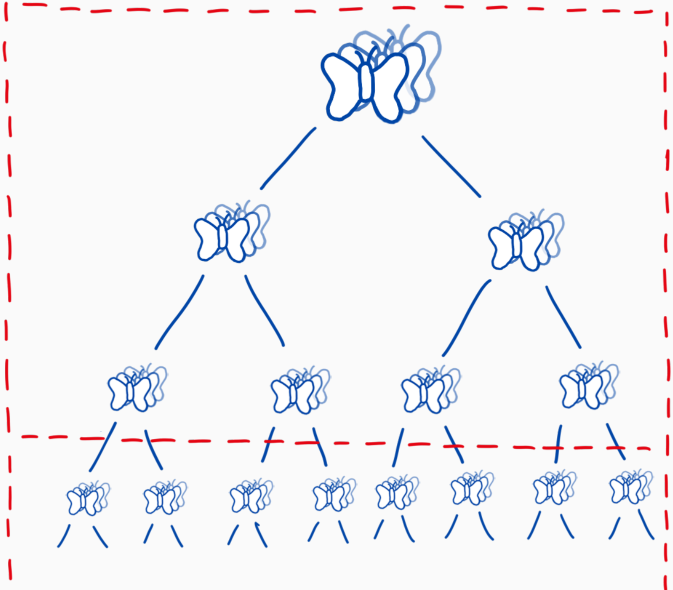
Merging layers (visualisation)



Merging layers (visualisation)







Merging layers (impl)

- M4: Merge 2 layers
- M3 (constant-time): No merged layers
- M3 (leaktime): Merge 2 layers

Optimization memory

Three strategies

(1) Storing A in flash (realistic setting)

(2) Storing A in SRAM ("vanilla" setting)

(3) Streaming A and y (how small can we go?)

Three strategies

(1) Storing A in flash (realistic setting)

- Can read A from flash during signing
- Needs extra flash space

(2) Storing A in SRAM (“vanilla” setting)

(3) Streaming A and y (how small can we go?)

Three strategies

(1) Storing A in flash (realistic setting)

- Can read A from flash during signing
- Needs extra flash space

(2) Storing A in SRAM (“vanilla” setting)

- Generate A once during signing
- Needs extra SRAM space

(3) Streaming A and y (how small can we go?)

Three strategies

(1) Storing A in flash (realistic setting)

- Can read A from flash during signing
- Needs extra flash space

(2) Storing A in SRAM ("vanilla" setting)

- Generate A once during signing
- Needs extra SRAM space

(3) Streaming A and y (how small can we go?)

- No extra space needed
- Likely to be very slow

Stack optimization

$\underline{\mathsf{Sign}(sk, M)}$

$\mathbf{A} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$ $\quad \triangleright$ $\mathbf{A}$ is generated and stored in NTT Representation as $\hat{\mathbf{A}}$

$\mu \in \{0,1\}^{384} := \mathsf{CRH}(tr \parallel M)$

$\kappa := 0,\ (\mathbf{z}, \mathbf{h}) := \perp$

$\rho' \in \{0,1\}^{384} := \mathsf{CRH}(K \parallel \mu)$ (or $\rho' \leftarrow \{0,1\}^{384}$ for randomized signing)

while $(\mathbf{z}, \mathbf{h}) = \perp$ do $\quad \triangleright$ Pre-compute $\hat{\mathbf{s}}_1 := \mathtt{NTT}(\mathbf{s}_1)$, $\hat{\mathbf{s}}_2 := \mathtt{NTT}(\mathbf{s}_2)$, and $\hat{\mathbf{t}}_0 := \mathtt{NTT}(\mathbf{t}_0)$

  $\mathbf{y} \in S_{\gamma_1 - 1}^{\ell} := \mathsf{ExpandMask}(\rho', \kappa)$

  $\mathbf{w} := \mathbf{A}\mathbf{y}$ $\quad \triangleright$ $\mathbf{w} := \mathtt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \mathtt{NTT}(\mathbf{y}))$

  $\mathbf{w}_1 := \mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2)$

  $c \in B_{60} := \mathsf{H}(\mu \parallel \mathbf{w}_1)$ $\quad \triangleright$ Store c in NTT representation as $\hat{c} = \mathtt{NTT}(c)$

  $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$ $\quad \triangleright$ Compute $c\mathbf{s}_1$ as $\mathtt{NTT}^{-1}(\hat{c} \cdot \hat{\mathbf{s}}_1)$

  $(\mathbf{r}_1, \mathbf{r}_0) := \mathsf{Decompose}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)$ $\quad \triangleright$ Compute $c\mathbf{s}_2$ as $\mathtt{NTT}^{-1}(\hat{c} \cdot \hat{\mathbf{s}}_2)$

  if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ or $\mathbf{r}_1 \neq \mathbf{w}_1$, then $(\mathbf{z}, \mathbf{h}) := \perp$

  else

    $\mathbf{h} := \mathsf{MakeHint}_q(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0, 2\gamma_2)$ $\quad \triangleright$ Compute $c\mathbf{t}_0$ as $\mathtt{NTT}^{-1}(\hat{c} \cdot \hat{\mathbf{t}}_0)$

    if $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ **or** the # of 1's in $\mathbf{h}$ is greater than ω, then $(\mathbf{z}, \mathbf{h}) := \perp$

  $\kappa := \kappa + 1$

return $\sigma = (\mathbf{z}, \mathbf{h}, c)$

Results

How we measured

Measuring performance

- M4: Use systick timer
- M3: Use the `DWT` cycle counter (`CYCCNT`)

How we measured

Measuring performance

- M4: Use systick timer
- M3: Use the `DWT` cycle counter (`CYCCNT`)

Measuring stack usage

(1) Fill the stack with dummy values
(2) Run the algorithm
(3) Count how many dummy bytes were overwritten

Results: NTT performance

				NTT	$\mathtt{NTT}^{-1}$	$\circ$
Dilithium	[GKOS18]	constant-time	M4	10 701	11 662	–
	This work	constant-time	M4	8 540	8 923	1 955
	This work	variable-time	M3	19 347	21 006	4 899
	This work	constant-time	M3	33 025	36 609	8 479

Results: NTT performance

				NTT	$\mathtt{NTT}^{-1}$	$\circ$
Dilithium	[GKOS18]	constant-time	M4	10 701	11 662	–
	This work	constant-time	M4	8 540	8 923	1 955
	This work	variable-time	M3	19 347	21 006	4 899
	This work	constant-time	M3	33 025	36 609	8 479

- On Cortex M4 we have a 25% improvement
- (Leaktime) operations on M3 are $2.3\times$ – $2.5\times$ slower
- Constant-time NTT $1.7\times$ slower than leaktime

Results M4 strategy 1

Algorithm/strategy	Params	Work	Speed [kcc]	Stack [B]
KeyGen (1)	`Dilithium2`	**This work**	2 267	7 916
	`Dilithium3`	**This work**	3 545	8 940
	`Dilithium4`	**This work**	5 086	9 964
Sign (1)	`Dilithium2`	[RGCB19, scen. 2]	3 640	–
	`Dilithium2`	**This work**	3 097	14 428
	`Dilithium3`	[RGCB19, scen. 2]	5 495	–
	`Dilithium3`	**This work**	4 578	17 628
	`Dilithium4`	[RGCB19, scen. 2]	4 733	–
	`Dilithium4`	**This work**	3 768	20 828
Verify	`Dilithium2`	**This work**	1 259	9 004
	`Dilithium3`	[GKOS18]	2 342	54 800
	`Dilithium3`	**This work**	1 917	10 028
	`Dilithium4`	**This work**	2 720	11 052

Results M4 strategy 2

Algorithm/strategy	Params	Work	Speed [kcc]	Stack [B]
KeyGen (2 & 3)	`Dilithium2`	**This work**	1 315	7 916
	`Dilithium3`	[GKOS18]	2 320	50 488
	`Dilithium3`	**This work**	2 013	8 940
	`Dilithium4`	**This work**	2 837	9 964
Sign (2)	`Dilithium2`	[RGCB19, scen. 1]	4 632	–
	`Dilithium2`	**This work**	3 987	38 300
	`Dilithium3`	[GKOS18]	8 348	86 568
	`Dilithium3`	[RGCB19, scen. 1]	7 085	–
	`Dilithium3`	**This work**	6 053	52 756
	`Dilithium4`	[RGCB19, scen. 1]	7 061	–
	`Dilithium4`	**This work**	6 001	69 276
Verify	`Dilithium2`	**This work**	1 259	9 004
	`Dilithium3`	[GKOS18]	2 342	54 800
	`Dilithium3`	**This work**	1 917	10 028
	`Dilithium4`	**This work**	2 720	11 052

Results M4 strategy 3

Algorithm/strategy	Params	Work	Speed [kcc]	Stack [B]
KeyGen (2 & 3)	`Dilithium2`	**This work**	1 315	7 916
	`Dilithium3`	[GKOS18]	2 320	50 488
	`Dilithium3`	**This work**	2 013	8 940
	`Dilithium4`	**This work**	2 837	9 964
Sign (3)	`Dilithium2`	**This work**	13 332	8 924
	`Dilithium3`	**This work**	23 550	9 948
	`Dilithium4`	**This work**	22 658	10 972
Verify	`Dilithium2`	**This work**	1 259	9 004
	`Dilithium3`	[GKOS18]	2 342	54 800
	`Dilithium3`	**This work**	1 917	10 028
	`Dilithium4`	**This work**	2 720	11 052

Performance results

Cortex M4

- At the time, fastest implementation for M4
- 13%, 27%, and 18% speedup compared to [GKOS18]
- 14% – 20% speedup compared to [RGCB19]

Results M3 strategy 1

Algorithm/ strategy	Params	Speed [kcc]	Stack [B]
KeyGen (1)	`Dilithium2`	2 945	12 631
	`Dilithium3`	4 503	15 703
	`Dilithium4`	6 380	18 783
Sign (1)	`Dilithium2`	5 822	14 869[a]
	`Dilithium3`	8 730	18 083[b]
	`Dilithium4`	7 398	18 083[c]
Verify	`Dilithium2`	1 541	8 944
	`Dilithium3`	2 321	9 967
	`Dilithium4`	3 260	10 999

[a] Uses additional 23 632 bytes of flash space.
[b] Uses additional 34 896 bytes of flash space.
[c] Uses additional 48 208 bytes of flash space.

Results M3 strategy 2

Algorithm/ strategy	Params	Speed [kcc]	Stack [B]
KeyGen (2 & 3)	Dilithium2	1 699	7 983
	Dilithium3	2 562	9 007
	Dilithium4	3 587	10 031
Sign (2)	Dilithium2	7 115	39 503
	Dilithium3	10 667	53 959
	Dilithium4	10 031	70 463
Verify	Dilithium2	1 541	8 944
	Dilithium3	2 321	9 967
	Dilithium4	3 260	10 999

Results M3 strategy 3

Algorithm/ strategy	Params	Speed [kcc]	Stack [B]
KeyGen (2 & 3)	`Dilithium2`	1 699	7 983
	`Dilithium3`	2 562	9 007
	`Dilithium4`	3 587	10 031
Sign (3)	`Dilithium2`	18 932	9 463
	`Dilithium3`	33 229	10 495
	`Dilithium4`	31 180	11 511
Verify	`Dilithium2`	1 541	8 944
	`Dilithium3`	2 321	9 967
	`Dilithium4`	3 260	10 999

Performance results

Cortex M3

- No previous work to compare with
- Signing: 40% – 100% more cycles than M4
- Verify only 20% slower

Memory results

- Keygen and Verify are always pretty cheap
- Generally need 40, 54, 70 kB of memory
- Strategy 1: 24, 35, 48 kB can be flash instead of SRAM

Memory results

- Keygen and Verify are always pretty cheap
- Generally need 40, 54, 70 kB of memory
- Strategy 1: 24, 35, 48 kB can be flash instead of SRAM

- Also can get signing to around 10 kB
- For a factor $3\times - 4\times$, we save 39, 43, 58 kB

Conclusion

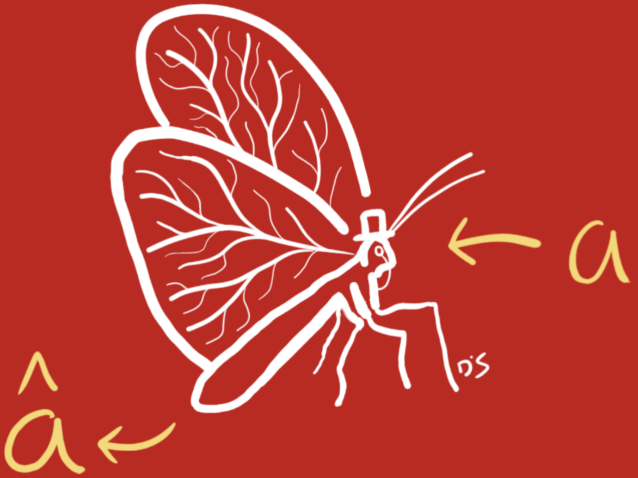
Links

Paper: `https://dsprenkels.com/files/dilithium-m3.pdf`

Code: `https://github.com/dilithium-cortexm/dilithium-cortexm`

Authors:

- Daan: `https://dsprenkels.com`
- Denisa: TBD
- Matthias: `https://kannwischer.eu`


a
â

Backup slides

Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal.
NTRU Prime.
Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2019.
available at
https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

Wouter de Groot.
A performance study of X25519 on Cortex-M3 and M4, 2015.
https://pure.tue.nl/ws/portalfiles/portal/47038543.

Tim Güneysu, Markus Krausz, Tobias Oder, and Julian Speith.
Evaluation of lattice-based signature schemes in embedded systems.
In *ICECS 2018*, pages 385–388, 2018.
https://www.seceng.ruhr-uni-bochum.de/media/seceng/veroeffentlichungen/2018/10/17/paper.pdf.

Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, and Damien Stehlé.
CRYSTALS-DILITHIUM.
Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS16], 2019.
available at
https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

NIST Computer Security Division.
Post-Quantum Cryptography Standardization, 2016.
`https://csrc.nist.gov/Projects/Post-Quantum-Cryptography`.

Prasanna Ravi, Sourav Sen Gupta, Anupam Chattopadhyay, and Shivam Bhasin.
Improving Speed of Dilithium's Signing Procedure.
In *CARDIS 2019*, volume 11833 of *LNCS*, pages 57–73. Springer, 2019.
`https://eprint.iacr.org/2019/420.pdf`.