



Implementing the NIST PQC standards on microcontrollers

Matthias J. Kannwischer
Academia Sinica, Taipei, Taiwan
matthias@kannwischer.eu

05 December 2022

PQC Standardization & Migration Workshop, Asiacrypt 2022, Taipei, Taiwan

The new NIST PQC standards

- NIST announced new NIST PQC standards on July 5, 2022 (March 127, 2022 in Nistian calendar)
 - KEM: Kyber (level 1: 800-byte public key, 768-byte ciphertext)
 - Sign: Dilithium (1312-byte public key, 2420-byte signature) – default
 - Sign: Falcon (897-byte public key, 666-byte signatures)
 - Sign: SPHINCS+ (32-byte public key, 7856-byte (small) or 17088-byte (fast) signature)
- Drafts standards to come out very soon



The new NIST PQC standards

- NIST announced new NIST PQC standards on July 5, 2022 (March 127, 2022 in Nistian calendar)
 - KEM: Kyber (level 1: 800-byte public key, 768-byte ciphertext)
 - Sign: Dilithium (1312-byte public key, 2420-byte signature) – default
 - Sign: Falcon (897-byte public key, 666-byte signatures)
 - Sign: SPHINCS+ (32-byte public key, 7856-byte (small) or 17088-byte (fast) signature)
- Drafts standards to come out very soon



The new NIST PQC standards

- NIST announced new NIST PQC standards on July 5, 2022 (March 127, 2022 in Nistian calendar)
 - KEM: Kyber (level 1: 800-byte public key, 768-byte ciphertext)
 - Sign: Dilithium (1312-byte public key, 2420-byte signature) – default
 - Sign: Falcon (897-byte public key, 666-byte signatures)
 - Sign: SPHINCS+ (32-byte public key, 7856-byte (small) or 17088-byte (fast) signature)
- Drafts standards to come out very soon



NIST PQC on microcontrollers

- NIST designated the Arm Cortex-M4 as the primary microcontroller optimization target – alongside Haswell (AVX2) and Artix-7 FPGAs
- Arm Cortex-M4 is an interesting target
 - Widely used for huge number of use-cases
 - Cheap discovery boards have sufficient RAM for most PQC
 - Instruction set is powerful enough to allow variety of implementation tricks
 - Pipeline is very simple (mostly 1 cycle exec); possible to understand every cycle



NIST PQC on microcontrollers

- NIST designated the Arm Cortex-M4 as the primary microcontroller optimization target – alongside Haswell (AVX2) and Artix-7 FPGAs
- Arm Cortex-M4 is an interesting target
 - Widely used for huge number of use-cases
 - Cheap discovery boards have sufficient RAM for most PQC
 - Instruction set is powerful enough to allow variety of implementation tricks
 - Pipeline is very simple (mostly 1 cycle exec); possible to understand every cycle



NIST PQC on microcontrollers

- NIST designated the Arm Cortex-M4 as the primary microcontroller optimization target – alongside Haswell (AVX2) and Artix-7 FPGAs
- Arm Cortex-M4 is an interesting target
 - Widely used for huge number of use-cases
 - Cheap discovery boards have sufficient RAM for most PQC
 - Instruction set is powerful enough to allow variety of implementation tricks
 - Pipeline is very simple (mostly 1 cycle exec); possible to understand every cycle



NIST PQC on microcontrollers: pqm4

- pqm4 is a benchmarking and testing framework for PQC on M4
- Created and maintained by Matthias Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, Ko Stoffelen
- Initial motivation: Get NIST PQC first round candidates to work on the Cortex-M4
 - Submissions contained C reference code
 - Some libraries not available for the M4
 - Many implementations used too much RAM
 - **Testing** automation proved to be very useful
- Later: Collection of optimized (assembly) implementations
 - We try collect all open-source implementations
 - We welcome pull requests or pointers to faster implementations
 - **Benchmarking** automation ensures consistent and fair benchmarking
 - **Testing** helps to catch some bugs
 - We publish benchmarking results



NIST PQC on microcontrollers: pqm4

- pqm4 is a benchmarking and testing framework for PQC on M4
- Created and maintained by Matthias Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, Ko Stoffelen
- Initial motivation: Get NIST PQC first round candidates to work on the Cortex-M4
 - Submissions contained C reference code
 - Some libraries not available for the M4
 - Many implementations used too much RAM
 - **Testing** automation proved to be very useful
- Later: Collection of optimized (assembly) implementations
 - We try collect all open-source implementations
 - We welcome pull requests or pointers to faster implementations
 - **Benchmarking** automation ensures consistent and fair benchmarking
 - **Testing** helps to catch some bugs
 - We publish benchmarking results



NIST PQC on microcontrollers: pqm4

- pqm4 is a benchmarking and testing framework for PQC on M4
- Created and maintained by Matthias Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, Ko Stoffelen
- Initial motivation: Get NIST PQC first round candidates to work on the Cortex-M4
 - Submissions contained C reference code
 - Some libraries not available for the M4
 - Many implementations used too much RAM
 - **Testing** automation proved to be very useful
- Later: Collection of optimized (assembly) implementations
 - We try collect all open-source implementations
 - We welcome pull requests or pointers to faster implementations
 - **Benchmarking** automation ensures consistent and fair benchmarking
 - **Testing** helps to catch some bugs
 - We publish benchmarking results



This talk: Microcontroller implementations

- Overview of performance of Kyber, Dilithium, Falcon, SPHINCS+ on Arm Cortex-M4
 - Speed, memory usage, code size, and hashing
- Recent papers on Kyber and Dilithium
 - **Improved Plantard Arithmetic for Lattice-based Cryptography** by Huang, Zhang, Zhao, Liu, Cheung, Koç, and Chen
ia.cr/2022/956
 - **Faster Kyber and Dilithium on the Cortex-M4** by Abdulrahman, Hwang, Kannwischer, and Sprenkels
ia.cr/2022/112
 - **Dilithium for Memory Constrained Devices** by Bos, Renes, and Sprenkels
ia.cr/2022/323



This talk: Microcontroller implementations

- Overview of performance of Kyber, Dilithium, Falcon, SPHINCS+ on Arm Cortex-M4
 - Speed, memory usage, code size, and hashing
- Recent papers on Kyber and Dilithium
 - **Improved Plantard Arithmetic for Lattice-based Cryptography** by Huang, Zhang, Zhao, Liu, Cheung, Koç, and Chen
ia.cr/2022/956
 - **Faster Kyber and Dilithium on the Cortex-M4** by Abdulrahman, Hwang, Kannwischer, and Sprenkels
ia.cr/2022/112
 - **Dilithium for Memory Constrained Devices** by Bos, Renes, and Sprenkels
ia.cr/2022/323



Implementations of SPHINCS+

- SPHINCS+ is vastly dominated by hashing
- Heavily optimized SHA-3/SHAKE and SHA-2 implementations are available
- Not much more one can do in terms of speed
 - Signing speed: 383 Mcycles (4s@100MHz; sphincs-sha256-128f-simple) to 93 299 Mcycles (15mins; sphincs-shake256-192s-robust)
 - Verification speed: 7 Mcycles (70ms; sphincs-sha256-128s-simple) to 243 Mcycles (2.4s; sphincs-shake256-256f-robust)
- Embedded systems **really** need SPHINCS+ with smaller parameters
Only option: allowing to sign fewer messages



Implementations of SPHINCS+

- SPHINCS+ is vastly dominated by hashing
- Heavily optimized SHA-3/SHAKE and SHA-2 implementations are available
- Not much more one can do in terms of speed
 - Signing speed: 383 Mcycles (4s@100MHz; sphincs-sha256-128f-simple) to 93 299 Mcycles (15mins; sphincs-shake256-192s-robust)
 - Verification speed: 7 Mcycles (70ms; sphincs-sha256-128s-simple) to 243 Mcycles (2.4s; sphincs-shake256-256f-robust)
- Embedded systems **really** need SPHINCS+ with smaller parameters
Only option: allowing to sign fewer messages



Implementations of SPHINCS+

- SPHINCS+ is vastly dominated by hashing
- Heavily optimized SHA-3/SHAKE and SHA-2 implementations are available
- Not much more one can do in terms of speed
 - Signing speed: 383 Mcycles (4s@100MHz; sphincs-sha256-128f-simple) to 93 299 Mcycles (15mins; sphincs-shake256-192s-robust)
 - Verification speed: 7 Mcycles (70ms; sphincs-sha256-128s-simple) to 243 Mcycles (2.4s; sphincs-shake256-256f-robust)
- Embedded systems **really** need SPHINCS+ with smaller parameters
Only option: allowing to sign fewer messages



Implementations of Falcon

- Fastest implementation: **New Efficient, Constant-Time Implementations of Falcon**
by Pornin
ia.cr/2019/893
- No progress since then
- Signing: 17.7 Mcycles (170ms; falcon-512-tree)
- Verification: 0.5 Mcycles (50ms; falcon-512-tree)



Implementations of Falcon

- Fastest implementation: **New Efficient, Constant-Time Implementations of Falcon**
by Pornin
ia.cr/2019/893
- No progress since then
- Signing: 17.7 Mcycles (170ms; falcon-512-tree)
- Verification: 0.5 Mcycles (50ms; falcon-512-tree)



Implementations of Kyber

- Long series of papers (not only me!)
 - 2019: **Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4** by Botros, Kannwischer, and Schwabe
ia.cr/2019/489
 - 2020: **Cortex-M4 Optimizations for {R,M}LWE Schemes** by Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard
ia.cr/2020/012
 - 2022: **Faster Kyber and Dilithium on the Cortex-M4** by Abdulrahman, Hwang, Kannwischer, and Sprenkels
ia.cr/2022/112
 - 2022: **Improved Plantard Arithmetic for Lattice-based Cryptography** by Huang, Zhang, Zhao, Liu, Cheung, Koç, and Chen
ia.cr/2022/956



Implementations of Kyber: ia.cr/2022/956

- **Improved Plantard Arithmetic for Lattice-based Cryptography**
- Presented at CHES 2022
- Main contribution: Improved modular multiplication for moduli < 16 bit
 - Heavily utilized within the number-theoretic transforms of Kyber
 - Also useful for other lattice-based schemes with small primes, e.g., NTRU (ia.cr/2019/040)



Implementations of Kyber: ia.cr/2022/956

- **Improved Plantard Arithmetic for Lattice-based Cryptography**
- Presented at CHES 2022
- Main contribution: Improved modular multiplication for moduli < 16 bit
 - Heavily utilized within the number-theoretic transforms of Kyber
 - Also useful for other lattice-based schemes with small primes, e.g., NTRU (ia.cr/2019/040)



Plantard multiplication

Old: Montgomery Multiplication

- $(ab + tq)/R$ with $t = -q^{-1}ab \bmod R$
- b pre-computed in Montgomery domain ($b = b'R \bmod q$)
- Implemented on the M4 in 3 cycles
smulbb t, a, b
smulbb t2, t, $-q^{-1}$
smlabb t2, t2, q, t
(result in top half of t2)

New: Plantard Multiplication

- Find $(tq - ab)/2^{2\ell}$ with $t = abq^{-1} \bmod 2^{2\ell}$
- $\frac{tq - ab}{2^{2\ell}} = \left[\frac{q \left\lfloor \frac{abq^{-1} \bmod \pm 2^{2\ell}}{2^\ell} \right\rfloor + q2^\alpha}{2^\ell} \right]$
- $\alpha = 3$ for Kyber
- Precompute $b' = (-b2^{2\ell} \bmod q)q^{-1} \bmod^\pm 2^{2\ell}$
- Implemented on the M4 in 2 cycles
smulwb r, b', a
smlabb r, r, q, $q2^\alpha$
(Result in top half of r)



Plantard multiplication

Old: Montgomery Multiplication

- $(ab + tq)/R$ with $t = -q^{-1}ab \bmod R$
- b pre-computed in Montgomery domain ($b = b'R \bmod q$)
- Implemented on the M4 in 3 cycles
smulbb t, a, b
smulbb t2, t, $-q^{-1}$
smlabb t2, t2, q, t
(result in top half of t2)

New: Plantard Multiplication

- Find $(tq - ab)/2^{2\ell}$ with $t = abq^{-1} \bmod 2^{2\ell}$
- $\frac{tq - ab}{2^{2\ell}} = \left[\frac{q \left\lfloor \frac{abq^{-1} \bmod \pm 2^{2\ell}}{2^\ell} \right\rfloor + q2^\alpha}{2^\ell} \right]$
- $\alpha = 3$ for Kyber
- Precompute $b' = (-b2^{2\ell} \bmod q)q^{-1} \bmod^\pm 2^{2\ell}$
- Implemented on the M4 in 2 cycles
smulwb r, b', a
smlabb r, r, q, $q2^\alpha$
(Result in top half of r)



Implementations of Dilithium: ia.cr/2022/112 (speed)

- **Faster Kyber and Dilithium on the Cortex-M4**
- Presented at ACNS 2022
- Contributions
 - Cooley–Tukey butterfly for Dilithium and Kyber iNTT
 - Port many known tricks from Saber/NTRU/NTRU Prime to Dilithium and Kyber
 - **Modulus switching for computation of cs_1 and cs_2**



Implementations of Dilithium: ia.cr/2022/112 (speed)

- **Faster Kyber and Dilithium on the Cortex-M4**
- Presented at ACNS 2022
- Contributions
 - Cooley–Tukey butterfly for Dilithium and Kyber iNTT
 - Port many known tricks from Saber/NTRU/NTRU Prime to Dilithium and Kyber
 - **Modulus switching for computation of cs_1 and cs_2**



Implementations of Dilithium: ia.cr/2022/112 (speed)

- Observation: Both cs_1 and cs_2 have small coefficients
 - c has τ coefficients in $\{-1, +1\}$
 $\tau = 39/49/60$ for dilithium2/3/5
 - s_1 and s_2 have coefficients in $\{-\eta, \eta\}$
 $\eta = 2/4/2$ dilithium2/3/5
 - Coefficients of cs_1 and cs_2 are at most $\tau\eta$ (signed)
i.e., 78/196/120 (signed!!)
 - We don't need arithmetic modulo $q = 8380417$, can switch to a smaller $q' > 156/392/240$
 - Can use Fermat number transform with $q' = 257$ for dilithium 2 and 5
 - Can use an NTT with $q' = 769$ for all parameter sets
 - Alternative: Re-use Kyber NTT with $q' = 3329$
 - Another trick: Can use incomplete NTT as in Kyber



Implementations of Dilithium: ia.cr/2022/112 (speed)

- Observation: Both cs_1 and cs_2 have small coefficients
 - c has τ coefficients in $\{-1, +1\}$
 $\tau = 39/49/60$ for dilithium2/3/5
 - s_1 and s_2 have coefficients in $\{-\eta, \eta\}$
 $\eta = 2/4/2$ dilithium2/3/5
 - Coefficients of cs_1 and cs_2 are at most $\tau\eta$ (signed)
i.e., 78/196/120 (signed!!)
 - We don't need arithmetic modulo $q = 8380417$, can switch to a smaller $q' > 156/392/240$
 - Can use Fermat number transform with $q' = 257$ for dilithium 2 and 5
 - Can use an NTT with $q' = 769$ for all parameter sets
 - Alternative: Re-use Kyber NTT with $q' = 3329$
 - Another trick: Can use incomplete NTT as in Kyber



Implementations of Dilithium: ia.cr/2022/112 (speed)

- Observation: Both cs_1 and cs_2 have small coefficients
 - c has τ coefficients in $\{-1, +1\}$
 $\tau = 39/49/60$ for dilithium2/3/5
 - s_1 and s_2 have coefficients in $\{-\eta, \eta\}$
 $\eta = 2/4/2$ dilithium2/3/5
 - Coefficients of cs_1 and cs_2 are at most $\tau\eta$ (signed)
i.e., 78/196/120 (signed!!)
 - We don't need arithmetic modulo $q = 8380417$, can switch to a smaller $q' > 156/392/240$
 - Can use Fermat number transform with $q' = 257$ for dilithium 2 and 5
 - Can use an NTT with $q' = 769$ for all parameter sets
 - Alternative: Re-use Kyber NTT with $q' = 3329$
 - Another trick: Can use incomplete NTT as in Kyber



Implementations of Dilithium: ia.cr/2022/112 (speed)

- Observation: Both cs_1 and cs_2 have small coefficients
 - c has τ coefficients in $\{-1, +1\}$
 $\tau = 39/49/60$ for dilithium2/3/5
 - s_1 and s_2 have coefficients in $\{-\eta, \eta\}$
 $\eta = 2/4/2$ dilithium2/3/5
 - Coefficients of cs_1 and cs_2 are at most $\tau\eta$ (signed)
i.e., 78/196/120 (signed!!)
 - We don't need arithmetic modulo $q = 8380417$, can switch to a smaller $q' > 156/392/240$
 - Can use Fermat number transform with $q' = 257$ for dilithium 2 and 5
 - Can use an NTT with $q' = 769$ for all parameter sets
 - Alternative: Re-use Kyber NTT with $q' = 3329$
 - Another trick: Can use incomplete NTT as in Kyber



Implementations of Dilithium: ia.cr/2022/112 (speed)

- Observation: Both cs_1 and cs_2 have small coefficients
 - c has τ coefficients in $\{-1, +1\}$
 $\tau = 39/49/60$ for dilithium2/3/5
 - s_1 and s_2 have coefficients in $\{-\eta, \eta\}$
 $\eta = 2/4/2$ dilithium2/3/5
 - Coefficients of cs_1 and cs_2 are at most $\tau\eta$ (signed)
i.e., 78/196/120 (signed!!)
 - We don't need arithmetic modulo $q = 8380417$, can switch to a smaller $q' > 156/392/240$
 - Can use Fermat number transform with $q' = 257$ for dilithium 2 and 5
 - Can use an NTT with $q' = 769$ for all parameter sets
 - Alternative: Re-use Kyber NTT with $q' = 3329$
 - Another trick: Can use incomplete NTT as in Kyber



Implementations of Dilithium: ia.cr/2022/112 (speed)

- Observation: Both cs_1 and cs_2 have small coefficients
 - c has τ coefficients in $\{-1, +1\}$
 $\tau = 39/49/60$ for dilithium2/3/5
 - s_1 and s_2 have coefficients in $\{-\eta, \eta\}$
 $\eta = 2/4/2$ dilithium2/3/5
 - Coefficients of cs_1 and cs_2 are at most $\tau\eta$ (signed)
i.e., 78/196/120 (signed!!)
 - We don't need arithmetic modulo $q = 8380417$, can switch to a smaller $q' > 156/392/240$
 - Can use Fermat number transform with $q' = 257$ for dilithium 2 and 5
 - Can use an NTT with $q' = 769$ for all parameter sets
 - Alternative: Re-use Kyber NTT with $q' = 3329$
 - Another trick: Can use incomplete NTT as in Kyber



Implementations of Dilithium: ia.cr/2022/112 (speed)

- Observation: Both cs_1 and cs_2 have small coefficients
 - c has τ coefficients in $\{-1, +1\}$
 $\tau = 39/49/60$ for dilithium2/3/5
 - s_1 and s_2 have coefficients in $\{-\eta, \eta\}$
 $\eta = 2/4/2$ dilithium2/3/5
 - Coefficients of cs_1 and cs_2 are at most $\tau\eta$ (signed)
i.e., 78/196/120 (signed!!)
 - We don't need arithmetic modulo $q = 8380417$, can switch to a smaller $q' > 156/392/240$
 - Can use Fermat number transform with $q' = 257$ for dilithium 2 and 5
 - Can use an NTT with $q' = 769$ for all parameter sets
 - Alternative: Re-use Kyber NTT with $q' = 3329$
- Another trick: Can use incomplete NTT as in Kyber



Implementations of Dilithium: ia.cr/2022/112 (speed)

- Observation: Both cs_1 and cs_2 have small coefficients
 - c has τ coefficients in $\{-1, +1\}$
 $\tau = 39/49/60$ for dilithium2/3/5
 - s_1 and s_2 have coefficients in $\{-\eta, \eta\}$
 $\eta = 2/4/2$ dilithium2/3/5
 - Coefficients of cs_1 and cs_2 are at most $\tau\eta$ (signed)
i.e., 78/196/120 (signed!!)
 - We don't need arithmetic modulo $q = 8380417$, can switch to a smaller $q' > 156/392/240$
 - Can use Fermat number transform with $q' = 257$ for dilithium 2 and 5
 - Can use an NTT with $q' = 769$ for all parameter sets
 - Alternative: Re-use Kyber NTT with $q' = 3329$
 - Another trick: Can use incomplete NTT as in Kyber



Dilithium: implementations: ia.cr/2022/323 (stack)

- **Dilithium for Memory Constrained Devices**
- Presented at Africacrypt 2022
- Contribution
 - Fit (most parameter sets of) Dilithium in 8 KiB of RAM



Dilithium: implementations: ia.cr/2022/323 (stack)

- **Dilithium for Memory Constrained Devices**
- Presented at Africacrypt 2022
- Contribution
 - Fit (most parameter sets of) Dilithium in 8 KiB of RAM



Dilithium: implementations: ia.cr/2022/323 (stack)

- Known from previous work: Re-compute everything within the loop
- Trick #1: $\mathbf{w} = \mathbf{A}\mathbf{y}$
 - Used two times in signing
 - Idea: Compress polynomial as 24-bit coefficients rather than 32 bit
- Trick #2: $c\mathbf{s}_1/c\mathbf{s}_2$
 - Use $q' = 257/769/257$ NTTs
 - Results in 16-bit coefficients rather than 32-bit coefficients
- Trick #3: $c\mathbf{t}_0$
 - \mathbf{t}_0 is not small, so we cannot switch to a smaller q'
 - Use schoolbook multiplication
 - \implies Can unpack \mathbf{t}_0 lazily from secret key
 - \implies Can store c compressed
- Trick #4: Careful variable allocation



Dilithium: implementations: ia.cr/2022/323 (stack)

- Known from previous work: Re-compute everything within the loop
- Trick #1: $\mathbf{w} = \mathbf{A}\mathbf{y}$
 - Used two times in signing
 - Idea: Compress polynomial as 24-bit coefficients rather than 32 bit
- Trick #2: $\mathbf{cS}_1/\mathbf{cS}_2$
 - Use $q' = 257/769/257$ NTTs
 - Results in 16-bit coefficients rather than 32-bit coefficients
- Trick #3: \mathbf{ct}_0
 - \mathbf{t}_0 is not small, so we cannot switch to a smaller q'
 - Use schoolbook multiplication
 - \implies Can unpack \mathbf{t}_0 lazily from secret key
 - \implies Can store c compressed
- Trick #4: Careful variable allocation



Dilithium: implementations: ia.cr/2022/323 (stack)

- Known from previous work: Re-compute everything within the loop
- Trick #1: $\mathbf{w} = \mathbf{A}\mathbf{y}$
 - Used two times in signing
 - Idea: Compress polynomial as 24-bit coefficients rather than 32 bit
- Trick #2: $c\mathbf{s}_1/c\mathbf{s}_2$
 - Use $q' = 257/769/257$ NTTs
 - Results in 16-bit coefficients rather than 32-bit coefficients
- Trick #3: $c\mathbf{t}_0$
 - \mathbf{t}_0 is not small, so we cannot switch to a smaller q'
 - Use schoolbook multiplication
 - \implies Can unpack \mathbf{t}_0 lazily from secret key
 - \implies Can store c compressed
- Trick #4: Careful variable allocation



Dilithium: implementations: ia.cr/2022/323 (stack)

- Known from previous work: Re-compute everything within the loop
- Trick #1: $\mathbf{w} = \mathbf{A}\mathbf{y}$
 - Used two times in signing
 - Idea: Compress polynomial as 24-bit coefficients rather than 32 bit
- Trick #2: $c\mathbf{s}_1/c\mathbf{s}_2$
 - Use $q' = 257/769/257$ NTTs
 - Results in 16-bit coefficients rather than 32-bit coefficients
- Trick #3: $c\mathbf{t}_0$
 - \mathbf{t}_0 is not small, so we cannot switch to a smaller q'
 - Use schoolbook multiplication
 - \implies Can unpack \mathbf{t}_0 lazily from secret key
 - \implies Can store c compressed
- Trick #4: Careful variable allocation



Dilithium: implementations: ia.cr/2022/323 (stack)

- Known from previous work: Re-compute everything within the loop
- Trick #1: $\mathbf{w} = \mathbf{A}\mathbf{y}$
 - Used two times in signing
 - Idea: Compress polynomial as 24-bit coefficients rather than 32 bit
- Trick #2: $c\mathbf{s}_1/c\mathbf{s}_2$
 - Use $q' = 257/769/257$ NTTs
 - Results in 16-bit coefficients rather than 32-bit coefficients
- Trick #3: $c\mathbf{t}_0$
 - \mathbf{t}_0 is not small, so we cannot switch to a smaller q'
 - Use schoolbook multiplication
 - \implies Can unpack \mathbf{t}_0 lazily from secret key
 - \implies Can store c compressed
- Trick #4: Careful variable allocation



NIST PQC: Performance

- For this comparison: security level 1, Keccak parameters
⇒ kyber512, dilithium2, falcon-512, sphincs-shake256-128f-simple, and sphincs-shake256-128s-simple
- For the full results, check
<https://github.com/mupq/pqm4/blob/master/benchmarks.md>

scheme	public key	ciphertext
X25519	32 bytes	32 bytes
kyber512	800 bytes	768 bytes

scheme	public key	signature
Ed25519	32 bytes	64 bytes
dilithium2	1 184 bytes	2 044 bytes
falcon512	897 bytes	690 bytes
sphincs-*-128f	32 bytes	17 088 bytes
sphincs-*-128s	32 bytes	7 856 bytes



NIST PQC: KEM Speed

scheme	impl	enc	dec
X25519 ¹		548 kcycles	548 kcycles
kyber512	speed	530 kcycles	477 kcycles
kyber512	stack	532 kcycles	478 kcycles

- Kyber is faster than X25519!
- Stack-efficient implementations come at virtually no cost for Kyber

¹X25519 implementation from <https://github.com/Emill/X25519-Cortex-M4>



NIST PQC: Signature Speed

scheme	impl	sign	open
Ed25519 ²		496 kcycles	1 265 kcycles
dilithium2	speed	4 111 kcycles	1 572 kcycles
dilithium2	stack	18 470 kcycles	4 036 kcycles
falcon-512-tree	speed	17 650 kcycles	481 kcycles
sphincs-shake256-128f-simple	clean	1 483 676 kcycles	83 065 kcycles
sphincs-shake256-128s-simple	clean	29 086 410 kcycles	29 495 kcycles

- Dilithium verification slightly slower than Ed25519; Falcon verification much faster
- Dilithium+Falcon signing vastly slower than Ed25519
- Dilithium stack optimization comes at a huge cost
- SPHINCS+ signing far from practical

²Ed25519 by Hayato Fujii, Diego F. Aranha, **Curve25519 for the Cortex-M4 and Beyond**, LATINCRYPT 2017, <https://github.com/hayatofujii/curve25519-cortex-m4>



NIST PQC: KEM Memory and Code

scheme	impl	enc stack	dec stack	code
X25519		368 bytes	368 bytes	1 892 bytes
kyber512	speed	5 424 bytes	5 432 bytes	15 332 bytes
kyber512	stack	2 336 bytes	2 352 bytes	12 820 bytes

- kyber512 fits in less than 2.5 KiB of RAM (best of all NIST PQC KEMs)
- Still a lot more RAM needed than for pre-quantum crypto
- Code size of current PQC implementations is huge



NIST PQC: Signature Memory and Code

scheme	impl	sign stack	open stack	code ³
Ed25519		?	?	28 240 bytes
dilithium2	speed	49 380 bytes	$\approx 10\,000^4$ bytes	18 480 bytes
dilithium2	stack	5 120 bytes	2 764 bytes	10 091 bytes
falcon-512-tree	speed	42 732 bytes	5 655 bytes	82 821 bytes
sphincs-shake256-128f-simple	clean	2 092 bytes	2 580 bytes	3 884 bytes
sphincs-shake256-128s-simple	clean	2 312 bytes	1 884 bytes	4 152 bytes

- Falcon and Dilithium have a huge memory footprint for signing if speed is a concern
- Falcon and Dilithium verification has a much smaller memory footprint
- Dilithium signing can be implemented in less than 8 KiB, but the speed suffers 5×
- SPHINCS+ has small memory footprint and code-size

³PQC results exclude code for Keccak (8000 bytes for optimized code)

⁴not currently implemented in pqm4; see ia.cr/2020/1278



NIST PQC: Hashing Dominance

scheme	impl	enc/sign	dec/open
kyber512	speed	82%	73%
dilithium2	speed	66%	81%
falcon-512-tree	speed	1%	36%
sphincs-shake256-128f-simple	clean	96%	96%
sphincs-shake256-128s-simple	clean	96%	96%

- Kyber, Dilithium, and SPHINCS+ are dominated by hashing



Conclusion

- Kyber and Dilithium still receiving significant attention after 5 years of NIST PQC
 - Plantard multiplication for Kyber
 - Switching coefficient rings for Dilithium
 - Compression within Dilithium signing
- Still not much attention to Falcon
- No hope to make SPHINCS+ with current parameters fast
 - Need: Smaller parameters, i.e., support fewer signatures



Conclusion

- Kyber and Dilithium still receiving significant attention after 5 years of NIST PQC
 - Plantard multiplication for Kyber
 - Switching coefficient rings for Dilithium
 - Compression within Dilithium signing
- Still not much attention to Falcon
- No hope to make SPHINCS+ with current parameters fast
 - Need: Smaller parameters, i.e., support fewer signatures



Conclusion

- New NIST PQC standards are vastly dominated by hashing
 - A hash accelerator will improve performance more than anything else!
 - Armv8.4-A (e.g., Arm Cortex-X2, Arm Neoverse V1, Apple M1) adds SHA-3 instructions, but no SHA-3 ISA support on most recent M-profile cores (Cortex-M85)
 - Co-processors are an option
- Memory is likely not a problem for Kyber and SPHINCS+
⇒ Virtually no performance penalty for stack-efficient implementations
- Memory is a huge challenge for Dilithium and Falcon (esp. signing)
⇒ 5 × slower for stack-efficient (5 KiB) Dilithium signing



Conclusion

- New NIST PQC standards are vastly dominated by hashing
 - A hash accelerator will improve performance more than anything else!
 - Armv8.4-A (e.g., Arm Cortex-X2, Arm Neoverse V1, Apple M1) adds SHA-3 instructions, but no SHA-3 ISA support on most recent M-profile cores (Cortex-M85)
 - Co-processors are an option
- Memory is likely not a problem for Kyber and SPHINCS+
 - ⇒ Virtually no performance penalty for stack-efficient implementations
- Memory is a huge challenge for Dilithium and Falcon (esp. signing)
 - ⇒ 5 × slower for stack-efficient (5 KiB) Dilithium signing



Conclusion

- New NIST PQC standards are vastly dominated by hashing
 - A hash accelerator will improve performance more than anything else!
 - Armv8.4-A (e.g., Arm Cortex-X2, Arm Neoverse V1, Apple M1) adds SHA-3 instructions, but no SHA-3 ISA support on most recent M-profile cores (Cortex-M85)
 - Co-processors are an option
- Memory is likely not a problem for Kyber and SPHINCS+
 - ⇒ Virtually no performance penalty for stack-efficient implementations
- Memory is a huge challenge for Dilithium and Falcon (esp. signing)
 - ⇒ 5 × slower for stack-efficient (5 KiB) Dilithium signing



Future of pqm4

- As of now: No plans for major changes to the framework and build system
- Focus: NIST PQC standards and candidates
 - Eliminated 3rd-round candidates removed; available in git history
 - Aiming to maintain the fastest (and smallest) implementation
- NIST PQC additional digital signature
 - Aiming to include suitable candidates shortly after they are posted
 - Ideally, through PQClean: <https://github.com/PQClean/PQClean>
 - We appreciate submission teams opening pull requests with their implementations
- Help always welcome
 - In particular: Looking for helping hands for integrating the new signature schemes
 - Please reach out to me if you want to get involved



Future of pqm4

- As of now: No plans for major changes to the framework and build system
- Focus: NIST PQC standards and candidates
 - Eliminated 3rd-round candidates removed; available in git history
 - Aiming to maintain the fastest (and smallest) implementation
- NIST PQC additional digital signature
 - Aiming to include suitable candidates shortly after they are posted
 - Ideally, through PQCclean: <https://github.com/PQCclean/PQCclean>
 - We appreciate submission teams opening pull requests with their implementations
- Help always welcome
 - In particular: Looking for helping hands for integrating the new signature schemes
 - Please reach out to me if you want to get involved



Future of pqm4

- As of now: No plans for major changes to the framework and build system
- Focus: NIST PQC standards and candidates
 - Eliminated 3rd-round candidates removed; available in git history
 - Aiming to maintain the fastest (and smallest) implementation
- NIST PQC additional digital signature
 - Aiming to include suitable candidates shortly after they are posted
 - Ideally, through PQClean: <https://github.com/PQClean/PQClean>
 - We appreciate submission teams opening pull requests with their implementations
- Help always welcome
 - In particular: Looking for helping hands for integrating the new signature schemes
 - Please reach out to me if you want to get involved



Thank you very much for your attention!
matthias@kannwischer.eu

<https://github.com/mupq/pqm4>

