Revisiting Keccak and Dilithium Implementations on ARMv7-M

Junhao Huang[1,2], Alexandre Adomnicăi[3], Jipeng Zhang[4], Wangchen Dai[5], Yao Liu[6], Ray C. C. Cheung[7], Çetin Kaya Koç[4,8,9], Donglong Chen[1*]

[1] Guangdong Provincial Key Laboratory IRADS, BNU-HKBU United International College, Zhuhai, China huangjunhao@uic.edu.cn,donglongchen@uic.edu.cn
[2] Hong Kong Baptist University, Hong Kong, China
[3] Independent researcher, Paris, France alexandre@adomnicai.me
[4] Nanjing University of Aeronautics and Astronautics, Nanjing, China jp-zhang@outlook.com
[5] Zhejiang Lab, Hangzhou, China w.dai@my.cityu.edu.hk
[6] Sun Yat-sen University, Zhuhai, China liuyao25@mail.sysu.edu.cn
[7] City University of Hong Kong, Hong Kong, China r.cheung@cityu.edu.hk
[8] Iğdır University, Turkey
[9] University of California Santa Barbara, Santa Barbara, USA cetinkoc@ucsb.edu
[*] Corresponding Author.

Abstract. Keccak is widely used in lattice-based cryptography (LBC) and its impact to the overall running time in LBC scheme can be predominant on platforms lacking dedicated SHA-3 instructions. This holds true on embedded devices for Kyber and Dilithium, two LBC schemes selected by NIST to be standardized as quantum-safe cryptographic algorithms. While extensive work has been done to optimize the polynomial arithmetic in these schemes, it was generally assumed that Keccak implementations were already optimal and left little room for enhancement.

In this paper, we revisit various optimization techniques for both Keccak and Dilithium on two ARMv7-M processors, i.e., Cortex-M3 and M4. For Keccak, we improve its efficiency using two architecture-specific optimizations, namely lazy rotation and memory access pipelining, on ARMv7-M processors. These optimizations yield performance gains of up to 24.78% and 21.4% for the largest Keccak permutation instance on Cortex-M3 and M4, respectively. As for Dilithium, we first apply the multi-moduli NTT for the small polynomial multiplication $c\mathbf{t}_i$ on Cortex-M3. Then, we thoroughly integrate the efficient Plantard arithmetic to the 16-bit NTTs for computing the small polynomial multiplications $c\mathbf{s}_i$ and $c\mathbf{t}_i$ on Cortex-M3 and M4. We show that the multi-moduli NTT combined with the efficient Plantard arithmetic could obtain significant speed-ups for the small polynomial multiplications of Dilithium on Cortex-M3. Combining all the aforementioned optimizations for both Keccak and Dilithium, we obtain 15.44% $\sim$ 23.75% and 13.94% $\sim$ 15.52% speed-ups for Dilithium on Cortex-M3 and M4, respectively. Furthermore, we also demonstrate that the Keccak optimizations yield 13.35% to 15.00% speed-ups for Kyber, and our Keccak optimizations decrease the proportion of time spent on hashing in Dilithium and Kyber by 2.46% $\sim$ 5.03% on Cortex-M4.

Keywords: Keccak, Dilithium, ARMv7-M, Plantard arithmetic, lattice-based cryptography

1 Introduction

Shor's algorithm on quantum computers will pose serious threats to the traditional public key cryptographic (PKC) standards, including RSA, ElGamal, and ECC that are based on the big integer factorization, discrete logarithm, and elliptic curve discrete logarithm problems. Faced with this challenge, many countries have been putting significant efforts for establishing future PKC standards, named as the post-quantum cryptography (PQC), i.e., cryptographic algorithms that are safe against quantum attacks. The NIST PQC standardization competition initiated in 2016 has received many submissions, in which five categories of mathematical hard problems were used to construct 82 PQC candidates. In 2022, four PQC finalists were announced by the NIST, in which Dilithium [DKL+18], a lattice-based cryptographic (LBC) digital signature scheme, has been selected to be standardized as ML-DSA [NIS23b]. Therefore, efficient implementations of the finalist Dilithium on various platforms will be a concentrated research area.

Dilithium is built upon the hardness of module learning-with-errors (MLWE) and module short-integer-solution (MSIS) problems. Similar to the other LBC schemes such as NewHope, Saber and Kyber, the time-consuming operations of Dilithium are the Keccak computations and polynomial multiplications, which are two key components in improving its efficiency. The polynomial multiplication in Dilithium is performed over the polynomial ring $\mathbb{Z}_q[X]/\left(X^{256}+1\right)$ and can be efficiently implemented with the number-theoretic transform (NTT), which has been thoroughly studied to improve the efficiency of LBC schemes in the past few years [Sei18, CHK+21, GKS20, ACC+22, ZHLR22, ZZH+21, HZZ+22, BHK+22, HZZ+23].

The modulus of Dilithium is a 23-bit prime number $q = 8380417$, which normally requires the use of the 32-bit NTT for polynomial multiplication. However, recent work shows that some polynomial multiplications of Dilithium involving the small polynomial c, whose coefficients are either 0 or ±1, can be accelerated with 16-bit NTT on Cortex-M4 [AHKS22] or parallel small polynomial multiplication (PSPM) on NEON / AVX2 / AVX-512 ISAs [ZHS+22, ZZS+23]. The PSPM proposed in [ZHS+22] is mainly suitable for ISA with a large register width so that it can compute several operations simultaneously. As for 32-bit processors like ARMv7-M, the PSPM is rather limited due to the small register width. As for using the 16-bit NTT in Dilithium, previous work [AHKS22] did not utilize the efficient Plantard arithmetic presented in [Pla21, HZZ+22]. Besides, their small 16-bit NTT only applies to $c\mathbf{s}_1$ and $c\mathbf{s}_2$ (abbr. as $c\mathbf{s}_i$ for $i = 1, 2$). The $c\mathbf{t}_0$ and $c\mathbf{t}_1$ (abbr. as $c\mathbf{t}_i$ for $i = 0, 1$) that share the similar property as $c\mathbf{s}_i$ have not been considered in their work [AHKS22]. Furthermore, we notice that previous works [ACC+22, CHK+21] have suggested that the multi-moduli NTT could be beneficial to Saber, NTRU, and LAC on some platforms like AVX2 or Cortex-M3. However, this technique has not been applied to Dilithium yet. Therefore, there are still various optimization strategies worth revisiting to further improve the performance of Dilithium.

The extensive research on polynomial arithmetic has already achieved excellent results. Recent research [GKS20] shows that the polynomial multiplication of Dilithium accounts for mainly less than 10% of the running time on Cortex-M4. On this platform, the most time-consuming operation in LBC schemes is actually Keccak, which is a multipurpose cryptographic primitive notably known for being the core of the SHA-3 and SHAKE standards [Dwo15]. Keccak is extensively used by many PQC schemes for various purposes, from seed expansion to CPA-to-CCA transforms. For example, Dilithium and Kyber, another LBC scheme to be standardized as ML-KEM [NIS23a], rely on Keccak to such an extent that hashing accounts for 85% of the running time on Cortex-M4 [GKS20]. While Keccak runs fast on high-end processors thanks to vectorization or dedicated instructions (e.g. ARMv8 SHA-3 extension) [BK22], its largest version does not show outstanding performance on constrained platforms (e.g. ARMv7-M microcontrollers) when fully implemented in software. This is mainly due to the lack of general-purpose registers to hold the entire 1600-bit internal state, leading to high register pressure and therefore register spilling (i.e. saving and restoring some intermediate variables to and from memory). By way of illustration, on ARM Cortex-M4 processors, the current fastest Keccak implementation spends around half of the running time in memory accesses. To reduce this overload, the Keccak designers suggested using a 12-round variant named TurboSHAKE [BDH+23] instead of the 24-round variant, but NIST was not in favour of such a decision[1]. This highlights the need for improvement on architectures such as ARMv7-M where Keccak does not show outstanding performance.

Our contributions. This paper revisits various optimization techniques for both Keccak and Dilithium on two ARMv7-M processors: Cortex-M3 and Cortex-M4. The contributions are summarized as follows.

- We improve Keccak's performance on Cortex-M3 and M4 by up to 24.78% and 21.4% utilizing two architecture-specific optimizations. The first one, denoted as lazy rotations, consists of taking advantage of the inline barrel shifter to eliminate explicit rotations in the linear layer. Note that this technique has been reported in the literature by Becker and Kannwischer in order to boost Keccak on ARMv8 architectures [BK22] but has not been ported to ARMv7-M so far. The second optimization consists of a more efficient memory access scheduling to avoid pipeline hazards, which results in over 12.84% performance improvements on ARMv7-M.
- We revisit the small NTT for $c\mathbf{s}_i$ and multi-moduli NTT for $c\mathbf{t}_i$ on ARMv7-M. While the potential benefits of integrating the multi-moduli NTT into Dilithium have been briefly discussed in [GKS20], they claimed that the multi-moduli NTT implementation using Montgomery arithmetic cannot outperform their constant-time 32-bit NTT implementation on Cortex-M3. This paper illustrates that leveraging the efficient Plantard arithmetic, the proposed multi-moduli NTT implementation on Cortex-M3 is totally constant-time and can indeed yield over 19.07% or 52.76% speed-ups when compared with the variable-time or constant-time 32-bit NTT in [GKS20], respectively.
- We then integrate the efficient Plantard arithmetic proposed in [Pla21, HZZ+22, HZZ+23] into the aforementioned 16-bit NTTs on Cortex-M3 and M4. Specifically, we focus on the NTTs performed over moduli 769 and 3329, and by leveraging the excellent properties of Plantard arithmetic, we eliminate all modular reductions in NTT and INTT over 769 and 3329. Moreover, we also demonstrate that the efficient Plantard arithmetic can be utilized to speed up the explicit Chinese remainder theorem (CRT) by 19.45% in the multi-moduli NTT implementation, which further improves the feasibility and efficiency of the multi-moduli NTT on Cortex-M3.

Combining all the aforementioned optimizations for both Keccak and Dilithium, we obtain 15.44% $\sim$ 23.75% and 13.94% $\sim$ 15.52% speed-ups for Dilithium on Cortex-M3 and Cortex-M4, respectively. Furthermore, we also demonstrate that the Keccak optimizations yield 13.35% $\sim$ 15.00% speed-ups for Kyber, and our Keccak optimizations decrease the proportion of time spent on hashing in Dilithium and Kyber by 2.46% $\sim$ 5.03% on Cortex-M4. Our implementations are publicly available at https://github.com/UIC-ESLAS/Dilithium-Multi-Moduli.

2 Preliminaries

This section first gives a brief introduction of the Dilithium digital signature scheme. Then, the time-consuming components of Dilithium, including polynomial multiplication and Keccak, are reviewed.

[1] https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/5HveEPBsbxY

Algorithm 1 Dilithium key generation (`keygen`) [DKL+18]

```
Output: $pk = (\rho, \mathbf{t}_1), sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
$\zeta \leftarrow \{0,1\}^{256}$
$(\rho, \rho', K) \in \{0,1\}^{256} \times \{0,1\}^{512} \times \{0,1\}^{256} := \mathrm{H}(\zeta)$
$\mathbf{A} \in R_q^{k \times \ell} := \mathrm{ExpandA}(\rho)$    ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$
$(\mathbf{s}_1, \mathbf{s}_2) \in S_\eta^\ell \times S_\eta^k := \mathrm{ExpandS}(\rho')$
$\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$    ▷ Compute $\mathbf{A}\mathbf{s}_1$ as $\mathrm{INTT}(\hat{\mathbf{A}} \cdot \mathrm{NTT}(\mathbf{s}_1))$
$(\mathbf{t}_1, \mathbf{t}_0) := \mathrm{Power2Round}_q(\mathbf{t}, d)$
$tr \in \{0,1\}^{256} := \mathrm{H}(\rho \| \mathbf{t}_1)$
return $(pk = (\rho, \mathbf{t}_1), sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0))$
```

Algorithm 2 Dilithium signature generation (`sign`) [DKL+18]

```
Input: Secret key $sk$ and message $M$
Output: $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$
$\mathbf{A} \in R_q^{k \times \ell} := \mathrm{ExpandA}(\rho)$    ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$
$\mu \in \{0,1\}^{512} := \mathrm{H}(tr \| M)$
$\kappa := 0, (\mathbf{z}, \mathbf{h}) := \perp$
$\rho' \in \{0,1\}^{512} := \mathrm{H}(K \| \mu)$ (or $\rho' \leftarrow \{0,1\}^{512}$ for randomized signing)
while $(\mathbf{z}, \mathbf{h}) = \perp$ do    ▷ Pre-compute $\hat{\mathbf{s}}_1 := \mathrm{NTT}(\mathbf{s}_1), \hat{\mathbf{s}}_2 := \mathrm{NTT}(\mathbf{s}_2)$, and $\hat{\mathbf{t}}_0 := \mathrm{NTT}(\mathbf{t}_0)$
    $\mathbf{y} \in \tilde{S}_{\gamma_1}^\ell := \mathrm{ExpandMask}(\rho', \kappa)$
    $\mathbf{w} := \mathbf{A}\mathbf{y}$    ▷ $\mathbf{w} := \mathrm{INTT}(\hat{\mathbf{A}} \cdot \mathrm{NTT}(\mathbf{y}))$
    $\mathbf{w}_1 := \mathrm{HighBits}_q(\mathbf{w}, 2\gamma_2)$
    $\tilde{c} \in \{0,1\}^{256} := \mathrm{H}(\mu \| \mathbf{w}_1)$
    $c \in B_\tau :=$ SampleInBall $(\tilde{c})$    ▷ Store $c$ in NTT representation as $\hat{c} = \mathrm{NTT}(c)$
    $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$    ▷ Compute $c\mathbf{s}_1$ as $\mathrm{INTT}(\hat{c} \cdot \hat{\mathbf{s}}_1)$
    $\mathbf{r}_0 := \mathrm{LowBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)$    ▷ Compute $c\mathbf{s}_2$ as $\mathrm{INTT}(\hat{c} \cdot \hat{\mathbf{s}}_2)$
    if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$, then $(\mathbf{z}, \mathbf{h}) := \perp$
    else
        $\mathbf{h} := \mathrm{MakeHint}_q(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0, 2\gamma_2)$    ▷ Compute $c\mathbf{t}_0$ as $\mathrm{INTT}(\hat{c} \cdot \hat{\mathbf{t}}_0)$
        if $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ or the # of 1's in $\mathbf{h}$ is greater than $\omega$, then $(\mathbf{z}, \mathbf{h}) := \perp$
    $\kappa := \kappa + \ell$
return $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$
```

2.1 Dilithium

Dilithium belongs to the LBC category, and its hardness is based on the MLWE and MSIS problems. Dilithium follows the "Fiat-Shamir with Aborts" approach proposed in [Lyu09, Lyu12] to construct the digital signature scheme. The key generation (`keygen`), signature generation (`sign`), and signature verification (`verify`) of Dilithium are shown in Algorithm 1, Algorithm 2, and Algorithm 3, respectively. Due to the introduction of the module structure, Dilithium needs to handle a large $k \times l$-dimensional matrix $\mathbf{A}$, where each entry is a degree-$(n-1)$ polynomial over the polynomial ring $R_q = \mathbb{Z}_q[X]/(X^n+1)$ with modulus $q = 2^{23} - 2^{13} + 1 = 8380417$. The module structure brings excellent flexibility to the implementation of Dilithium, which allows us to reuse the fundamental component for different parameter sets of Dilithium.

As shown in Algorithm 1∼3, Dilithium extensively uses SHA-3 or SHAKE primitives for expanding the matrix $\mathbf{A}$ (ExpandA), secret vector $\mathbf{s}_1$ and $\mathbf{s}_2$ (ExpandS), vector y (ExpandMask), small polynomial c with τ non-zero coefficients ±1's (SampleInBall) and other hashing (H). The MakeHint, UseHint, Power2Round functions are used to drop the

Algorithm 3 Dilithium signature verification (`verify`) [DKL+18]

```
Input: Public key $pk$, message $M$, and signature $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$
Output: Accept or reject
$\mathbf{A} \in R_q^{k\times\ell} := \mathrm{ExpandA}(\rho)$   ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$
$\mu \in \{0,1\}^{512} := \mathrm{H}\left(\mathrm{H}\left(\rho\|\mathbf{t}_1\right)\|M\right)$
$c := \mathrm{SampleInBall}(\tilde{c})$
$\mathbf{w}_1' := \mathrm{UseHint}_q\left(\mathbf{h}, \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2\right)$   ▷
    Compute $\mathrm{INTT}(\hat{\mathbf{A}} \cdot \mathrm{NTT}(\mathbf{z}) - \mathrm{NTT}(c) \cdot \mathrm{NTT}(\mathbf{t}_1 \cdot 2^d))$
return $[\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!]$ and $[\![\tilde{c} = \mathrm{H}\left(\mu\|\mathbf{w}_1'\right)]\!]$ and $[\![\#$ of 1 's in $\mathbf{h}$ is $\le \omega]\!]$
```

Table 1: Dilithium parameters [DKL+18]

NIST security level	2	3	5
q [modulus]	8380417	8380417	8380417
n [the order of polynomial]	256	256	256
d [drop bits from $\mathbf{t}$]	13	13	13
τ [# of ±1's in c]	39	49	60
γ_1 [$\mathbf{y}$ coefficient range]	2^{17}	2^{19}	2^{19}
γ_2 [low-order rounding range]	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
(k, l) [dimensions of $\mathbf{A}$]	(4,4)	(6,5)	(8,7)
η [secret key range]	2	4	2
$\beta = \tau \cdot \eta$ [$c\mathbf{s}_i$ coefficient range]	78	196	120
$\mathbf{t}_0$ coefficient range	2^{12}	2^{12}	2^{12}
$\mathbf{t}_1$ coefficient range	2^{10}	2^{10}	2^{10}

lower d bits of $\mathbf{t}$ and reduce the public key size. The parameters used in these algorithms are shown in Table 1. For details of these functions and parameters, we refer interested readers to the specification of Dilithium [DKL+18].

2.2 Polynomial multiplication

2.2.1 Number-Theoretic Transform

The polynomial multiplication of Dilithium is performed over the polynomial ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $q = 8380417$ and $n = 256$. The modulus q satisfies $q \equiv 1 \bmod 2n$ so that a $2n$-th root of unity ($\zeta = 1753$) exists. Since $\zeta^{256} = -1 \bmod q$, one can then split the cyclotomic polynomial as $X^{256} + 1 = X^{256} - \zeta^{256} = (X^{128} - \zeta^{128})(X^{128} + \zeta^{128})$ using the NTT, which is a variant of the fast Fourier transform (FFT) over a finite field. This factorization process can be iteratively continued for the two degree-128 polynomials separately. Note that $X^{128} + \zeta^{128}$ is equivalent to $X^{128} - \zeta^{384}$ and it can be further factorized as $X^{128} - \zeta^{384} = (X^{64} - \zeta^{192})(X^{64} + \zeta^{192})$. The parameters of Dilithium ($q \equiv 1 \bmod 2n$) enable us to perform a complete 8-layer NTT (complete NTT) and factorize the cyclotomic polynomial $X^n + 1$ into linear factors $X - \zeta^i$ with $i = 1, 3, 5, \ldots, 511$. (Note that there are also incomplete NTTs where we can only factorize the cyclotomic polynomial into factors $X^2 - \zeta^i$; see Kyber [BDK+18] for example.) The cyclotomic polynomial ring R_q is isomorphic to the product of the rings $\mathbb{Z}_q[X]/(X - \zeta^i)$ according to the Chinese remainder theorem (CRT):

$$a \mapsto \left(a(\zeta), a\left(\zeta^3\right), \ldots, a\left(\zeta^{511}\right)\right) : R_q \to \prod_i \mathbb{Z}_q[X]/\left(X - \zeta^i\right).$$

After the NTT transform, the degree-$(n-1)$ polynomial multiplication of a and b is transformed into n computationally inexpensive pointwise multiplications over NTT-domain $\hat{a}$ and $\hat{b}$. After the pointwise multiplications, the inverse NTT transform (INTT) would convert the NTT-domain elements back to the normal domain. Overall, the time complexity of the polynomial multiplication is reduced down to $O(n\log(n))$, and the polynomial multiplication ab using NTT can be described as $\mathrm{INTT}(\mathrm{NTT}(a)\circ\ \mathrm{NTT}(b))$.

2.2.2 Small polynomial multiplications in Dilithium

Because the modulus of Dilithium q is a 23-bit prime number, most of the polynomial multiplications over the polynomial ring R_q are implemented with 32-bit NTT. However, there is a special small polynomial c which has exactly τ ±1's and $256-\tau$ 0's. There are four small polynomial multiplications involving c in Algorithm 2 and Algorithm 3, namely $c\mathbf{s}_1, c\mathbf{s}_2, c\mathbf{t}_0,$ and $c\mathbf{t}_1$. For simplicity, we denote $c\mathbf{s}_1$ and $c\mathbf{s}_2$ as $c\mathbf{s}_i$ for $i=1,2$, and $c\mathbf{t}_0$ and $c\mathbf{t}_1$ as $c\mathbf{t}_i$ for $i=0,1$. Due to the special structure of c, these polynomial multiplications would normally generate a polynomial product with a relatively small coefficient range. For example, the coefficient range of $\mathbf{s}_i$ is $[-\eta,\eta]$, then the coefficients of the product $c\mathbf{s}_i$ are smaller than $\tau\cdot\eta$, denoted as $\beta=\tau\cdot\eta$. As for $c\mathbf{t}_i$, since the coefficients of $\mathbf{t}_0$ and $\mathbf{t}_1$ are smaller than 2^{10} and 2^{12}, the coefficients of the products $c\mathbf{t}_0$ and $c\mathbf{t}_1$ are smaller than $\tau\cdot2^{10}$ and $\tau\cdot2^{12}$ (denoted as $\beta'=\tau\cdot2^{10}$ or $\beta'=\tau\cdot2^{12}$), respectively.

According to [CHK+21, Section 2.4.6], these kinds of polynomial multiplications can be treated as multiplications over $\mathbb{Z}_{q'}/[X](X^n+1)$ with a prime modulus $q'>2\beta$ or $q'>2\beta'$. Therefore, instead of using 32-bit NTT over R_q for computing $c\mathbf{s}_i$ and $c\mathbf{t}_i$ as in the previous implementation [GKS20], recent research [AHKS22] shows that the polynomial multiplications of $c\mathbf{s}_i$ can be implemented with smaller and faster 16-bit NTT by taking advantage of the fact that 2β is smaller than 2^{16}. More precisely, it suggests to use Fermat number transform (FNT) with Fermat number $q'=257$ or 16-bit NTT with $q'=769$ for computing $c\mathbf{s}_i$ in different parameter sets. As for $c\mathbf{t}_i$, since the modulus q' that satisfies $q'>2\beta'$ is larger than 2^{16}, it cannot be accelerated with the faster 16-bit NTT. However, as we will show later, one can still choose a modulus smaller than the q of Dilithium to further speed up $c\mathbf{t}_i$ using the multi-moduli NTT technique on Cortex-M3.

2.2.3 Modular arithmetic

The fundamental operation of 16-bit or 32-bit NTT is the modular multiplication by a twiddle factor modulo a 16-bit or 32-bit modulus. There are different optimal algorithms dealing with the 16-bit and 32-bit modular arithmetic. For 16-bit modular arithmetic, recent work [Pla21, HZZ+22, HZZ+23] shows that one can efficiently compute the modular multiplication by the twiddle factor modulo a 16-bit modulus with the Plantard arithmetic. Plantard arithmetic has a special 16×32 multiplication. If the target platform can efficiently compute the 16×32 multiplication, then the Plantard multiplication by a constant is one multiplication faster than the widely-used Montgomery [Mon85] and Barrett multiplication [Bar86]. This could further speed up the butterfly unit in the NTT implementation.

As for 32-bit modular arithmetic, it is difficult to apply the Plantard arithmetic to 32-bit modulus on 32-bit ARMv7-M processors like Cortex-M3 and M4 because the Plantard arithmetic requires to handle the 32×64 multiplication in this case, which would bring an extra multiplication on 32-bit processors and eliminate the gain of the Plantard arithmetic. On the contrary, the most efficient 32-bit modular arithmetic on 32-bit ARMv7-M processors would still be the Montgomery arithmetic [GKS20].

2.2.4 The 16-bit NTT and 32-bit NTT

It should be noted that the constant-time 32-bit NTT is at least 2× slower than the constant-time 16-bit NTT on platforms like Cortex-M3 and AVX2. This is mainly due to the inherent architectural design of these platforms. More specifically, since Cortex-M3 does not have constant-time full multiplications like `UMULL, SMULL, UMLAL`, and `SMLAL`, the constant-time 32-bit modular multiplication proposed in [GKS20] is implemented using the schoolbook multiplication over two 16-bit limbs and is more expensive than 16-bit modular multiplication on Cortex-M3. As a result, the constant-time 32-bit NTT is about 3 times slower than the constant-time 16-bit NTT, and the variable-time 32-bit NTT is also 1.78× slower than the constant-time 16-bit NTT on Cortex-M3 according to [GKS20, Table 2]. This suggests that 16-bit NTT is a more favorable choice over 32-bit NTT on Cortex-M3. Apart from M3, AVX2 is another platform where 16-bit NTT is preferable to 32-bit NTT. The AVX2 instructions have the capability to process operations simultaneously over sixteen 16-bit coefficients or eight 32-bit coefficients when handling 16-bit NTT or 32-bit NTT. Employing the 32-bit NTT inherently diminishes the parallelism utilization of AVX2, resulting in decreased performance compared to the 16-bit NTT. According to the AVX2 reference implementations of Kyber[2] and Dilithium[3], the 16-bit NTT implementation of Kyber outperforms the 32-bit NTT implementation of Dilithium on AVX2 by a factor of 4.11×. To summarize, the architectural characteristics of Cortex-M3 and AVX2 lead to the constant-time 32-bit NTT being at least 2× slower than the constant-time 16-bit NTT.

2.2.5 Multi-moduli NTT and the explicit CRT

The slower 32-bit NTT on AVX2 motivates Chung et al. [CHK$^+$21] to replace the 32-bit NTT with multiple efficient 16-bit NTTs (multi-moduli NTT) with the help of the explicit CRT. Similar to [CHK$^+$21], Abdulrahman et al. [ACC$^+$22] further demonstrate the feasibility of using multi-moduli NTT for Saber on Cortex-M3 due to the much slower 32-bit NTT than 16-bit NTT. Nevertheless, the multi-moduli NTT technique has not been applied to Dilithium yet. It should be noted that Greconici et al. [GKS20] also briefly discussed the potential to use the multi-moduli NTT and CRT to replace the 32-bit NTT in Dilithium. However, they [GKS20] claimed that the multi-moduli NTT implementation with Montgomery arithmetic was slower than re-implementing the 32-bit multiplication with constant-time 16-bit schoolbook multiplication [GKS20, Section 4.1]. In this paper, we will revisit this idea and demonstrate that by leveraging the efficient Plantard arithmetic for 16-bit NTTs, the multi-moduli NTT remains a viable option for Dilithium on Cortex-M3.

The so-called multi-moduli NTT method [CHK$^+$21, ACC$^+$22] chooses s coprime NTT-friendly primes $q_i (i = 0, 1, \dots, s-1)$ whose product $q' = \prod_{i-0}^{s-1} q_i$ is larger than $2\beta'$. The polynomial multiplication over the composite modulus q' can be carried out by computing NTTs modulo each prime q_i. Recall that the polynomial multiplication of $c\mathbf{t}_i$ would produce a polynomial with coefficients smaller than a maximum value $\beta' = \tau 2^{10}$ or $\beta' = \tau 2^{12}$. According to [CHK$^+$21, Section 2.4.6], we can choose an NTT-friendly prime $q' > 2\beta'$ such that the polynomial multiplication of $c\mathbf{t}_i$ can be implemented with the NTT over q'. Since $2\beta' > 2^{16}$, the polynomial multiplication of $c\mathbf{t}_i$ cannot be accelerated with the 16-bit NTT. Therefore, instead of using the extremely slow 32-bit NTT for $c\mathbf{t}_i$ on Cortex-M3, we can also choose a composite modulus $q' = q_0 q_1$ and utilize the multi-moduli NTT to speed up $c\mathbf{t}_i$. By leveraging the efficient Plantard arithmetic for 16-bit NTTs, we can further optimize the multi-moduli NTT and make it practical for Dilithium on Cortex-M3.

After performing NTTs, pointwise multiplications and INTTs modulo each prime q_i, we can then reconstruct the final results using the explicit CRT. One can either use the Lagrangian interpolation CRT algorithm proposed in [MS90, Section 4] or the divided-difference interpolation CRT algorithm presented in [CHK$^+$21, Theorem 1]. It is suggested that the latter one is better when s is smaller. Because the coefficients

[2] `https://github.com/pq-crystals/kyber`

[3] `https://github.com/pq-crystals/dilithium`

produced in $c\mathbf{t}_i$ are quite small, we only need two primes ($s = 2$) for the multi-moduli NTT implementation; so the latter one is best suited for our case. Let q_0, q_1 be co-prime ($\gcd(q_0, q_1){=}1$), q be the modulus of Dilithium, and $m_1 = q_0^{-1} \bmod^{\pm} q_1$. Let $u \equiv u_i \bmod q_i, i = 0, 1$, where $|u_i| < q_i/2, |u| < q_0 q_1/2$, then the explicit solution for u and $u \bmod^{\pm} q$ is given by $u = u_0 + \big((u_1 - u_0)\, m_1 \bmod^{\pm} q_1\big)\, q_0$ and $u \bmod^{\pm} q = \big(u_0 + \big(\big((u_1 - u_0)\, m_1 \bmod^{\pm} q_1\big) \bmod^{\pm} q\big) \cdot q_0\big) \bmod^{\pm} q$, respectively.

2.3 Keccak

Keccak is built upon the sponge construction together with a cryptographic permutation Keccak-p$[b, n_r]$ where b and n_r refer to the bit-width of the permutation and the number of rounds, respectively. In this paper, we focus exclusively on instances where $b = 1600$, such as the variant Keccak-p[1600,24] used in the NIST standards. The state A is represented as an array of 5×5 lanes, each composed of $w = b/25$ bits. $A[x, y]$ refers to the lane at position (x, y) and $A[x, y, z]$ refers to the z-th bit of the lane. Keccak-p is an iterated permutation where each round consists of five consecutive operations θ, ρ, π, χ and ι, where χ is the only non-linear operation. See Listing 1 for a brief description of Keccak-p using pseudo-code.

```
# b refers to the permutation width while nr refers to the number of rounds
keccak-p[b,nr](A):
  A = roundperm(A,RC[i])                                          for i in 0..nr-1
  return A

# r[x,y] refer to rotation offsets while RC refers to the round constant
roundperm(A,RC):
  # theta step
  C[x]    = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4]          for x in 0..4
  D[x]    = C[x-1] xor rot(C[x+1],1)                                    for x in 0..4
  A[x,y] = A[x,y] xor D[x]                                   for (x,y) in (0..4,0..4)
  # rho and pi step
  B[y,2*x+3*y] = rot(A[x,y], r[x,y])                         for (x,y) in (0..4,0..4)
  # chi step
  A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y])          for (x,y) in (0..4,0..4)
  # iota step
  A[0,0] = A[0,0] xor RC
  return A
```

Listing 1: Pseudo-code of the Keccak-p cryptographic permutation.

2.4 Target platforms: ARMv7-M processors

ARMv7-M refers to the microcontroller profile of the ARMv7 architecture. It comes with sixteen 32-bit registers (`r0-r15`), out of which one is used as stack pointer (`r13`), one is used as link register (`r14`), and one for the program counter (`r15`). It supports the Thumb-2 technology, which means that 16-bit and 32-bit encoding of instructions can be freely mixed. In this paper, we consider the ARM Cortex-M3 and M4 processors.

The Cortex-M3 and M4 processors have a 3-stage pipeline. Most instructions take a single cycle, except branches, multiplication, and memory accesses which may take more cycles. While load (`ldr`) and store (`str`) instructions typically require 2 and 1 cycles, respectively, `ldr` instructions can take up to 3 cycles to complete in case of dependency with the previous instruction. In the absence of such dependency, n `ldr` can be pipelined together to be executed in $n + 1$ cycles, and `str` following `ldr` takes 0 cycle. Both of them are equipped with a barrel shifter, allowing the flexible second operands to be shifted or rotated as part of an instruction without affecting performance. However, when the amount to be shifted or rotated is specified by a register, the instruction will take an extra cycle to complete. Since Cortex-M4 is equipped with a digital signal processing (DSP) extension, it supports additional SIMD instructions that can simultaneously handle two 16-bit data in one instruction. While most of the multiplication instructions on Cortex-M4 are 1-cycle, the multiplication instructions on Cortex-M3 would take more cycles. For example, the multiply-and-add instruction (`mla`) takes two cycles. Furthermore, a critical consideration in cryptographic implementations is that the 32-bit full multiplication instructions on Cortex-M3 (`UMULL, SMULL, UMLAL`, and `SMLAL`) do not run in constant-time due to the early-termination mechanism [dG15]. The `UMULL` and `SMULL` would take 3-5 cycles, while `UMLAL` and `SMLAL` may take 4-7 cycles. Therefore, we should avoid using these variable-time instructions directly when dealing with secret inputs.

3 Keccak Optimizations on ARMv7-M

This section first reviews the existing optimization techniques and analyses the factors that affect the overall Keccak performance on ARMv7-M. Subsequently, two architecture-specific optimization techniques for enhancing Keccak on ARMv7-M are introduced.

3.1 Existing optimization techniques on ARMv7-M

The designers of Keccak have summarized many possible optimizations for software and hardware implementations in a dedicated document [BDH+12]. The two most useful ones on ARMv7-M are described below.

Bit-interleaving. The intuitive way to implement Keccak-p in software is to follow a lane-wise architecture (i.e. a register contains one or multiple lanes), as described in Listing 1. On platforms where registers are not large enough to handle an entire lane, the designers recommend using the bit-interleaving technique. For $b = 1600$ on 32-bit architectures, it consists of storing bits at odd positions in one register, and bits at even positions in another register. In this way, 64-bit rotations can be easily handled by separate 32-bit rotations without additional operations. Note that this requires some extra calculations for rearranging the lanes at the beginning and at the end of the permutation. However it is calculated at a higher level (i.e. when adding and extracting data to and from the state), and therefore it is not taken into account when benchmarking the permutation itself.

Efficient in-place processing. When updating the internal state during the round function, it is possible to store all processed data back into the same memory location it was loaded from. In this way, only a single instance of the state (instead of two) must be preserved. Because the π operation moves lanes within the state, it requires defining a mapping between the lane coordinates and the memory location depending on the round number. The Keccak designers propose a linear map using a matrix of order 4, which means the state will return to its initial memory location after 4 rounds.

Performance analysis. On 32-bit architectures, each round of Keccak-p$[1600, \cdot]$ consists of 152 XORs, 50 ANDs, 50 NOTs and 58 rotations thanks to the bit-interleaving technique. On ARM, it is possible to merge 1 AND and 1 NOT into a single `bic` instruction and the rotations during the θ step can be easily combined with an XOR thanks to the inline barrel shifter. This leads to 250 instructions per round overall: 152 `eor`, 50 `bic` and 48 `ror`. Assuming logical instructions take 1 cycle, the raw cost of Keccak-p$[1600, 24]$ on this platform should theoretically be $250 \times 24 = 6\,000$ clock cycles. However, given the fact that the state is 1600-bit long and that ARMv7-M only offers 14 32-bit general-purpose registers to work with, it implies that performance will inevitably bear the cost of many loads and stores on the stack. As reference, we consider the ARMv7-M implementation from the extended Keccak Code Package (XKCP) [BDH+]. Because it contains optimized, free and open-source implementations for various platforms and architectures, it is often used as a third-party software component. According to previous research, the assembly implementation of Keccak-p[1600, 24] from XKCP requires 12 969 clock cycles on the Cortex-M4 [Sto19], meaning that around 54% of the cycles are spent in memory accesses. However, there is still room for improvement as explained hereafter.

3.2 Pipelining memory accesses

The ARMv7-M assembly implementation provided by XKCP at the time of writing[4] works as follows. At the beginning of each round, all the parity lanes (namely `D[x]` on line 10 of Listing 1) are precomputed. To compute half a parity lane, the code relies on a macro `xor5` which consists of 5 `ldr` and 4 exclusive-OR (`eor`) instructions, as detailed in Listing 2. Once this precomputing phase is complete, it executes the θ, ρ, π, χ and ι steps by a group of 5 half lanes at a time.

On the Cortex-M3 and M4, it is clear that the `xor5` macro suffers pipeline stalls since only 2 out of the 5 `ldr` instructions are consecutive. While grouping all these loads together would allow saving 3 cycles per macro call, this requires to change the way variables are assigned to registers since `r1` is used as the only destination of `ldr` instructions in order to preserve the other registers for subsequent calculations. Nevertheless, we managed to relax the register pressure thanks to another register allocation, allowing us to pipeline all the 5 `ldr` together within the `xor5` macro as detailed in Listing 3. To further improve memory access pipelining, we also reordered some other instructions throughout the code. Notably, we moved `str` instructions after multiple `ldr` as much as possible. While these pipelining optimizations have been carried out manually, tools are being developed to achieve optimal assembly implementations in an automated manner as recently illustrated by the SLOTHY optimizer based on constraint solving [ABKK24].

```
.macro xor5   result,b,g,k,m,s
    ldr       \result, [r0, #\b]
    ldr       r1, [r0, #\g]
    eors      \result, \result, r1
    ldr       r1, [r0, #\k]
    eors      \result, \result, r1
    ldr       r1, [r0, #\m]
    eors      \result, \result, r1
    ldr       r1, [r0, #\s]
    eors      \result, \result, r1
.endm
```

Listing 2: Original ARMv7-M assembly code from [BDH+] to compute half a parity lane. Loads from memory are not fully grouped and thus not optimally pipelined on M3 and M4 processors.

```
.macro xor5   result,b,g,k,m,s
    ldr       \result, [r0, #\b]
    ldr       r1,  [r0, #\g]
    ldr       r5,  [r0, #\k]
    ldr       r11, [r0, #\m]
    ldr       r12, [r0, #\s]
    eors      \result, \result, r1
    eors      \result, \result, r5
    eors      \result, \result, r11
    eors      \result, \result, r12
.endm
```

Listing 3: ARMv7-M assembly code after optimization to compute half a parity lane. Loads from memory are now fully grouped and thus optimally pipelined on M3 and M4 processors.

3.3 Lazy rotations

The XKCP implementation makes use of explicit rotations for the ρ step through `ror` instructions. While it makes the code easy to follow, it requires 47 such instructions per round. As recently proposed by Becker and Kannwischer on AArch64 [BK22], one can omit those explicit rotations by means of *lazy rotations* (i.e. rotating the second operands thanks to the inline barrel shifter) during subsequent operations. They recommend to defer the explicit rotations until the θ step in the next round: once all the (unrotated) parity lanes have been calculated, then they are rotated explicitly. By proceeding this way, the (unrotated) state lanes are lazily rotated when treated as second operands during the XOR with the (rotated) parity lanes, so that the internal state is back to the classical

[4] https://github.com/XKCP/XKCP/commit/7fa59c0ec4b5802b7c269ddd9ef0ef35999b4f0f

representation and the process can be reiterated thereafter. While it would be also possible to keep deferring rotations, it would require to fully unroll the permutation code, resulting in substantial impacts on code size.

On AArch64, it leads to 3 explicit rotations instead of 5 since 2 deferred rotation values are in fact 0. While it should theoretically result in 6 explicit rotations on ARMv7-M because of its 32-bit architecture, it is possible to stick to 3 rotations overall as described below. Thanks to the bit-interleaving technique, computing a 1-bit rotation on a 64-bit word can be achieved using a single 32-bit rotation only. Therefore, when computing the parity lanes, only 5 `eor` instructions need to rotate their second operand, leaving the barrel shifter available for deferred rotations during the remaining 5 `eor`. In the end, only 3 explicit rotations remain per round instead of 47.

We implemented this technique on ARMv7-M along with the in-place processing optimization in two different ways. While the most efficient approach is to use lazy rotations for all rounds, it requires to have specific routines for the first and last rounds to deal with input and output misalignments since the internal state is expected to be properly aligned at function entry and exit. When considering in-place processing, and therefore a quadruple round routine, it results in a code size increase by half since the first and last rounds should both come in two variants. In order to boost the performance while limiting the impact on code size, we also propose a variant where lazy rotations are applied for three-quarters of the rounds only. This way, explicit rotations are used every 4 rounds to ensure the internal state is correctly aligned when entering the quadruple round. Still, a potential drawback of deferring rotations is that it affects the code readability, which may make the integration of side-channel countermeasures (e.g. masking) more cumbersome.

4 Dilithium Optimizations on ARMv7-M

This section presents the small NTT for $c\mathbf{s}_i$ in Dilithium on Cortex-M3 and M4, as well as the multi-moduli NTT for $c\mathbf{t}_0$ in Dilithium on Cortex-M3. Then, the efficient 16-bit NTT and explicit CRT implementations with the Plantard arithmetic are introduced. Finally, the security and extensibility of this work will be briefly discussed.

4.1 Small NTTs for $c\mathbf{s}_i$ on Cortex-M3 and M4

The polynomial multiplication of $c\mathbf{s}_i$ produces a polynomial with a coefficient range smaller than $\beta = \tau\eta$. Hence, the polynomial multiplication of $c\mathbf{s}_i$ can be treated as polynomial multiplication over the NTT-friendly polynomial ring $\mathbb{Z}_{q'}[X]/(X^n + 1)$ with $q' > 2\beta$ [CHK+21, Section 2.4.6]. The coefficient range β equals 78, 196 and 120 for all three Dilithium parameter sets (see Table 1), respectively. Based on this observation, previous work [AHKS22] implemented $c\mathbf{s}_i$ with the small NTT over 769 ($769 > 2 \times 196$) in Dilithium3 and FNT over 257 ($257 > 2 \times 120$) in Dilithium2 and Dilithium5 on Cortex-M4. Since the pointwise multiplication and INTT are the core computations during the rejection sampling in `sign`, the FNT over 257 implementation in [AHKS22] provides faster pointwise multiplication, which enables better Dilithium2 and Dilithium5 than the NTT over 769. Therefore, we also reuse the FNT over 257 for Dilithium2 and Dilithium5 on Cortex-M4. The NTT over 769 with Plantard arithmetic is adopted for Dilithium3 on Cortex-M4.

On Cortex-M3, we decide to reuse the NTT over 769 to compute $c\mathbf{s}_i$ for all three Dilithium parameter sets for the following reasons. (1) The 16-bit NTT over 769 can be speeded up with the efficient Plantard arithmetic while the FNT over 257 is already very efficient, and there is not much space for further optimization. (2) We will later show that the NTT over 769 together with NTT over 3329 can enable an efficient multi-moduli NTT implementation on Cortex-M3 (see Subsection 4.2). Therefore, the NTT over 769 can be reused for computing both $c\mathbf{s}_i$ and $c\mathbf{t}_i$. Although the FNT over 257 can also enable the multi-moduli NTT together with the NTT over 7681, the FNT over 257 is only applicable for Dilithium2 and Dilithium5 and we would need two different multi-moduli NTT implementations for all variants of Dilithium. On the other hand, the NTT over 769 can be reused for all three Dilithium variants, and using the NTT over 769 allows us to reuse one multi-moduli NTT implementation for all three variants of Dilithium.

4.2 Multi-moduli NTT for $c\mathbf{t}_i$ on Cortex-M3

4.2.1 Parameters choice

As for $c\mathbf{t}_i$, the coefficient range of the product is smaller than $\tau 2^{10}$ or $\tau 2^{12}$, and the maximum value of τ is 60 according to Table 1. Therefore, the coefficient range of $c\mathbf{t}_i$ is $\beta' = \tau 2^{12} = 60 \times 2^{12} = 245760$. These polynomial multiplications can be treated as the polynomial multiplication over $\mathbb{Z}_{q'}/[X](X^n + 1)$ with a 32-bit NTT-friendly modulus $q' > 2\beta' = 491520$ [CHK+21, Section 2.4.6]. As mentioned in Subsection 2.2, either the variable-time or constant-time 32-bit NTT is much slower than 16-bit NTT on Cortex-M3. Recent research [HZZ+23] shows that the 16-bit NTT can be further optimized using the efficient Plantard arithmetic on Cortex-M3. The 16-bit NTT with Plantard arithmetic in [HZZ+23] is 2.41× faster than the variable-time 32-bit NTT and 4.11× faster than the constant-time 32-bit NTT on Cortex-M3.

Based on this observation, instead of choosing a 32-bit NTT-friendly prime and directly using the slow 32-bit NTT for computing $c\mathbf{t}_i$, we choose a composite modulus $q' = 769 \times 3329 = 2560001$ defined by the product of two 16-bit NTT-friendly primes 769 and 3329. It is easy to see that the selected modulus q' is larger than $2\beta'$; so the polynomial multiplication of $c\mathbf{t}_i$ can be treated as a multiplication over $\mathbb{Z}_{q'}[X]/(X^n + 1)$ with $q' = q_0 \times q_1, q_0 = 769$ and $q_1 = 3329$. Using this composite modulus, we could utilize the multi-moduli NTT technique to speed up $c\mathbf{t}_i$ in Dilithium. It should be noted that this technique has been adopted for Saber, NTRU and LAC [CHK+21, ACC+22] but has not been applied to Dilithium yet. Previous work [GKS20] also briefly discussed the potential to replace the slow 32-bit NTT with multi-moduli NTT, but they claimed that the multi-moduli NTT implementation with Montgomery arithmetic was slower than their schoolbook-based 32-bit NTT [GKS20, Section 4.1]. This paper revisits this idea and shows that together with the efficient Plantard arithmetic for 16-bit modulus, we are able to speed up the multi-moduli NTT and utilize it to compute $c\mathbf{t}_0$ on Cortex-M3.

4.2.2 Incomplete NTTs over 769 and 3329

Unlike Dilithium where the parameters q and n satisfy $2n|(q-1)$, the moduli $q_0 = 769$ and $q_1 = 3329$ only satisfy $n|(q_i - 1)$ for $i = 0, 1$. That means there exists an n-th root of unity ζ_i for the corresponding q_i such that the cyclotomic polynomial $X^n + 1$ can be factorized to degree-1 factors $X^2 - \zeta_i^j$ with $j = 1, 3, 5, \ldots, 255$. According to CRT, the polynomial ring R_{q_i} is isomorphic to the product of the degree-1 polynomial rings, namely $R_{q_i} \cong \prod \mathbb{Z}_{q_i}[X]/(X^2 - \zeta_i^j)$. Using the incomplete NTTs, the performance of NTT and INTT can be improved because they only need to compute 7 layers of NTT or INTT. On the other hand, the pointwise multiplications for these incomplete NTTs are performed modulo $X^2 - \zeta_i^j$ which is a little more complicated than pointwise multiplication modulo the degree-0 $X - \zeta$ in Dilithium. Nevertheless, these pointwise multiplications have been greatly improved by using lazy reduction [ABCG20] and asymmetric multiplication strategies [ACC+22, AHKS22]. Overall, using the incomplete NTTs could still obtain performance improvement, which can be demonstrated by the parameters choice of Kyber [BDK+18].

Algorithm 4 Multi-moduli NTT for computing 32-bit NTT on Cortex-M3

```
Input: Declare arrays: int32_t c_32[256],t_32[256],tmp_32[256],res_32[256]
Input: Declare pointers: {
    int16_t *cl_16=(int16_t*)c_32;
    int16_t *ch_16=(int16_t*)&c_32[128];
    int16_t *tl_16=(int16_t*)t_32;
    int16_t *th_16=(int16_t*)&t_32[128];
    int16_t *tmpl_16=(int16_t*)tmp_32;
    int16_t *tmph_16=(int16_t*)&tmp_32[128];
cl_16[256] ← $c$, ch_16[256] ← $c$    ▷ Pre-store $c$ in the bottom and top halves of c_32 as 16-bit arrays
tl_16[256] ← $t$, th_16[256] ← $t$    ▷ Pre-store $t$ in the bottom and top halves of t_32 as 16-bit arrays
cl_16[256] = $\mathrm{NTT}_{q_0}$(cl_16)    ▷ $\hat{c}_0 = \mathrm{NTT}_{q_0}(c)$
ch_16[256] = $\mathrm{NTT}_{q_1}$(ch_16)    ▷ $\hat{c}_1 = \mathrm{NTT}_{q_1}(c)$
tl_16[256] = $\mathrm{NTT}_{q_0}$(tl_16)    ▷ $\hat{t}_0 = \mathrm{NTT}_{q_0}(t)$
th_16[256] = $\mathrm{NTT}_{q_1}$(th_16)    ▷ $\hat{t}_1 = \mathrm{NTT}_{q_1}(t)$
tmpl_16[256] = $\mathrm{basemul}_{q_0}$(cl_16, tl_16)    ▷ $\hat{c}_0 \cdot \hat{t}_0 = \mathrm{basemul}_{q_0}(\hat{c}_0, \hat{t}_0)$
tmph_16[256] = $\mathrm{basemul}_{q_1}$(ch_16, th_16)    ▷ $\hat{c}_1 \cdot \hat{t}_1 = \mathrm{basemul}_{q_1}(\hat{c}_1, \hat{t}_1)$
tmpl_16[256] = $\mathrm{INTT}_{q_0}$(tmpl_16)    ▷ $\mathrm{INTT}_{q_0}(\hat{c}_0 \cdot \hat{t}_0)$
tmph_16[256] = $\mathrm{INTT}_{q_1}$(tmph_16)    ▷ $\mathrm{INTT}_{q_1}(\hat{c}_1 \cdot \hat{t}_1)$
res_32[256] = CRT(tmpl_16, tmph_16)    ▷ $\mathrm{CRT}(\mathrm{INTT}_{q_0}(\hat{c}_0 \cdot \hat{t}_0), \mathrm{INTT}_{q_1}(\hat{c}_1 \cdot \hat{t}_1))$
return res_32
```

4.2.3 NTT with composite modulus

In this paper, we are dealing with NTT over the composite modulus $q' = q_0q_1$ with $q_0 = 769$ and $q_1 = 3329$. Let ζ_0 and ζ_1 be the 256-th roots of unity in $\mathbb{Z}_{q_0}$ and $\mathbb{Z}_{q_1}$, respectively. According to the CRT and incomplete NTTs, we have the following isomorphism (the proof is similar to [ACC+22, Appendix E]):

$$\begin{aligned}
&\mathbb{Z}_{q_0q_1} \cong \mathbb{Z}_{q_0} \times \mathbb{Z}_{q_1};\\
&\mathbb{Z}_{q_0}[X]/(X^{256}+1) \cong \mathbb{Z}_{q_0}[X]/(X^2 - \zeta_0^j), j = 1,3,5,\ldots,255;\\
&\mathbb{Z}_{q_1}[X]/(X^{256}+1) \cong \mathbb{Z}_{q_1}[X]/(X^2 - \zeta_1^j), j = 1,3,5,\ldots,255;
\end{aligned} \tag{1}$$

Overall, the polynomial multiplication over $\mathbb{Z}_{q'}[X]/(X^n+1)$ can be separately computed using the faster 16-bit NTTs over $R_{q_i} = \mathbb{Z}_{q_i}[X]/(X^n+1)$ [ACC+22, Section 4.2.1]. After the 16-bit NTTs, pointwise multiplications and INTTs, one then uses the explicit CRT described in Subsubsection 2.2.5 to reconstruct the 32-bit results.

Memory layout. The workload of the multi-moduli NTT is illustrated in Algorithm 4. To clearly understand the workload, we need to describe the memory layout for the multi-moduli NTT. Let t be a polynomial in $\mathbf{t}_i$ with $i = 0, 1$. To compute the polynomial multiplication ct, previous implementation using 32-bit NTT [GKS20] stores c and t in 32-bit arrays, i.e., `int32_t c_32[256]`, `t_32[256]`. However, according to the coefficient range of c and t (see Table 1), they can actually be stored in 16-bit arrays. In order to utilize the multi-moduli NTT to compute ct, we modify the `unpack_sk` or `poly_challenge` functions to generate two copies of c and t and pre-store them as 16-bit arrays in the bottom and top halves of `c_32` and `t_32`. We use 16-bit pointers (`int16_t *cl_16,*ch_16,*tl_16,*th_16`) to access the bottom and top halves of these 32-bit arrays as 16-bit arrays. Overall, as shown in Algorithm 4, the multi-moduli NTT for ct needs to compute four NTTs, two pointwise multiplications, two INTTs, and one CRT. Compared to the 32-bit NTT implementation, our multi-moduli NTT implementation requires 1024 bytes of memory (`tmp_32`) for the pointwise multiplication and CRT computation. However, when compared to a very similar implementation for Saber in [ACC+22, Algorithm 5], our implementation only needs one-third of their memory. The main difference is that we can reuse the memory of c and t for in-place NTT/INTT implementation using the new `unpack_sk` and `poly_challenge` functions; thus achieve significant memory savings.

4.2.4 Multi-moduli NTT in `sign` and `verify`

`sign`. The proposed multi-moduli NTT is applicable to $c\mathbf{t}_0$ in `sign` (Algorithm 2) and $c\mathbf{t}_1$ in `verify` (Algorithm 3) on Cortex-M3. The present multi-moduli NTT incorporates two NTTs over 769 and 3329. Notably, the NTT over 769 is also utilized for the computation of $c\mathbf{s}_i$ in `sign`. Previous work [AHKS22] utilized the FNT over 257 or NTT over 769 for $c\mathbf{s}_i$ on Cortex-M4. However, their $c\mathbf{t}_0$ in `sign` was implemented with the 32-bit NTT of Dilithium. Consequently, they needed to perform both a 32-bit NTT and a 16-bit NTT on c for the subsequent computations. In contrast, by employing multi-moduli NTT over the chosen modulus $q' = q_0 q_1$, we only require one multi-moduli NTT operation on c. The computation of $c\mathbf{s}_i$ can leverage the NTT-domain $\hat{c}$ over q_0 generated in the multi-moduli NTT, thereby eliminating an extra NTT computation on c.

`verify`. Although the multi-moduli NTT is beneficial in `sign`, it is somewhat impractical in `verify`. One of the main step in `verify` is Step 4 in Algorithm 3, i.e. $\mathbf{Az} - c\mathbf{t}_1 \times 2^d$. It is recommended to perform the INTT after the subtraction. When using the 32-bit NTT for $c\mathbf{t}_1 \times 2^d$, the subtraction can be carried out after the pointwise multiplication, as both operations utilize the same NTT over the modulus q. However, when employing a multi-moduli NTT over $q' = q_0 q_1$ for $c\mathbf{t}_1$ and a 32-bit NTT over q for $\mathbf{Az}$, they are in different NTT domains and can not be directly subtracted. Consequently, using the multi-moduli NTT in `verify` necessitates an additional INTT computation for a polynomial vector. Furthermore, since c and $\mathbf{t}_1$ are public, previous implementation on Cortex-M3 [GKS20] chooses to utilize variable-time 32-bit NTT, which is faster than constant-time one. Although our multi-moduli NTT is faster than the variable-time 32-bit NTT (see Section 5), the extra INTT computation for a polynomial vector outweighs the benefits of using the multi-moduli NTT. Therefore, the proposed multi-moduli NTT implementation offers advantages solely in accelerating the signature generation process of Dilithium on Cortex-M3.

4.3 Efficient 16-bit NTTs with Plantard arithmetic

We now present the efficient 16-bit NTT implementations using the Plantard arithmetic on Cortex-M3 and M4. The 16-bit NTTs we used in this paper include NTTs over $q_0 = 769$ and $q_1 = 3329$. Note that NTT over 769 is used on both platforms while NTT over 3329 is only deployed on Cortex-M3.

4.3.1 Plantard arithmetic for 16-bit modulus

The efficient implementation of Plantard multiplication by a constant for 16-bit modulus q_i on Cortex-M3 is shown in Algorithm 5 [HZZ+23]. For its efficient implementation on Cortex-M4, we refer to [HZZ+22, Algorithm 11]. In order to utilize the efficient Plantard arithmetic in our 16-bit NTT implementation, an important positive integer α_i that satisfies $q_i < 2^{16-\alpha_i-1}$ is required. For the moduli $q_0 = 769$ and $q_1 = 3329$ that we used in this paper, we set $\alpha_0 = 5$ for $q_0 = 769$ and $\alpha_1 = 3$ for $q_1 = 3329$. Besides, we also need to precompute $q_0' = q_0^{-1} \bmod 2^{32}$ and $q_1' = q_1^{-1} \bmod 2^{32}$ to carry out the Plantard arithmetic.

Algorithm 5 Efficient Plantard multiplication by a constant for 16-bit modulus q_i on Cortex-M3 [HZZ+23]

```
Input: Two signed integers $a, b$ such that $a \in (q_i 2^{16} - q_i 2^{16+\alpha_i}, 2^{32} - q_i 2^{16+\alpha_i})$, a precomputed 32-bit integer $bq'_i$ where $b$ is a constant and $q'_i = q_i^{-1} \bmod^{\pm} 2^{32}$
Output: $r = ab(-2^{-32}) \bmod^{\pm} q_i$
$bq'_i \leftarrow bq_i^{-1} \bmod 2^{32}$                ▷ precomputed
mul $r, a, bq'_i$
add $r, 2^{\alpha_i}, r$, asr#16
mul $r, r, q_i$
asr $r, r$, #16
return $r$
```

In 2023, Huang et al. [HZZ+23] showed that the input range of the Plantard multiplication can be enlarged to $ab \in (q_i 2^{16} - q_i 2^{16+\alpha_i}, 2^{32} - q_i 2^{16+\alpha_i})$. When b is a constant, we can always reduce it down to $[0, q_i)$; so when replacing the specific q_i and $b_{max} = q_i - 1$ into the input range $ab \in (q_i 2^{16} - q_i 2^{16+\alpha_i}, 2^{32} - q_i 2^{16+\alpha_i})$, we have $a \in [-2645q_0, 4541q_0]$ and $a \in [-137q_1, 230q_1]$ for $q_0 = 769$ and $q_1 = 3329$, respectively. When performing NTT on Cortex-M3, each coefficient loaded into the register could be bigger than a 16-bit signed integer but we need to ensure that it is reduced down to a 16-bit signed integer when we need to store it back to the 16-bit memory [HZZ+23]. However, the NTT on Cortex-M4 utilizes the SIMD instruction and the coefficients must fit in 16-bit signed integers. Hence, the safe coefficient range for a 16-bit signed integer is in $[-42q_0, 42q_0]$ and $[-9q_1, 9q_1]$ for q_0 and q_1, respectively.

4.3.2 NTTs over 769 and 3329

Butterfly units. As the core operation of NTT, the efficiency of butterfly units is essential in NTT/INTT implementation. We adopt the state-of-the-art butterfly combination presented in [ACC+22, AHKS22] for NTTs over 769 and 3329 on Cortex-M3 and M4, namely using Cooley-Turkey (CT) butterfly for both NTT and INTT. We follow the twiddle factor precomputation in [HZZ+22, Section 4.2.1] in order to utilize the efficient Plantard multiplication by a constant for the modular multiplication by twiddle factor, namely $\zeta'_i = \zeta_i \cdot (-2^{32} \bmod q_i) \cdot (q_i^{-1} \bmod 2^{32}) \bmod 2^{32}$.

Layer merging. We follow the state-of-the-art 4+3 and 3+4 layer merging strategies in previous work [AHKS22, HZZ+22] for NTT and INTT on Cortex-M4. As for Cortex-M3, the 3+3+1 and 3+1+3 layer merging strategies for NTT and INTT on Cortex-M3 are adopted as in [HZZ+23].

Range analysis. The input range of the 16-bit NTT must be smaller than the coefficient range of $\mathbf{t}_0$, i.e., 2^{12}. When using CT butterfly and Plantard arithmetic for NTT, each layer of NTT would increase the coefficient size by $q_i/2$; so seven layers of NTT would increase the coefficient size by $3.5q_i$. For both moduli 769 and 3329, NTT would generate coefficients smaller than $2^{12} + 3.5q_i < 2^{15}$, which can fit in 16-bit signed integers. Therefore, the NTT with Plantard arithmetic does not need any modular reduction. We will see that after the pointwise multiplication described in Subsubsection 4.3.4, the input range of INTT is in $[-(q_i + 1)/2, q_i/2)$.

On Cortex-M4, we are using INTT over 769 for $c\mathbf{s}_i$ and we need to ensure that every coefficient must not overflow a 16-bit signed integer. Since the INTT is implemented with CT butterfly similar to the implementation in [ACC+22, AHKS22], it would bring additional twisting in the last layer of INTT. To minimize the side-effect of the additional twisting, they [ACC+22, AHKS22] introduced the light butterfly which could omit the

Algorithm 6 The explicit CRT with Plantard arithmetic on Cortex-M3

```
Input: $u_0 = u \bmod q_0, u_1 = u \bmod q_1, m_1 = q_0^{-1} \bmod^{\pm} q_1, m_1' = m_1 \cdot (-2^{32} \bmod q_1) \cdot (q_1^{-1} \bmod 2^{32}) \bmod 2^{32}, q_1 2^{\alpha_1} < 2^{15}$
Output: $u = u_0 + ((u_1 - u_0)m_1 \bmod^{\pm} q_1)q_0$
1: sub $t, u_1, u_0$
2: mul $t, t, m_1'$
3: add $t, 2^{\alpha_1}, t, \mathrm{asr}\#16$
4: mul $t, t, q_1$
5: asr $t, t, \#16$          ▷ $t \leftarrow (u_1 - u_0)m_1 \bmod^{\pm} q_1$
6: mla $u, t, q_0, u_0$      ▷ $u \leftarrow u_0 + tq_0$
7: return $u$
```

same amount of modular multiplication. The first three layers and the first iteration of the last three layers of INTT are implemented with light butterflies. Since the input range of INTT is smaller $q_0/2$, the first 3-layer INTT with light butterfly increase some of the coefficients up to $4.5q_0$. The fourth layer of INTT with CT butterfly further increases these coefficients up to $5q_0$. Then, in the final three layers of INTT with light butterfly, these coefficients would be increased to $40.5q_0$, which can still fit in a 16-bit signed integer for q_0 (see Subsubsection 4.3.1). Therefore, the INTT over 769 can totally eliminate the modular reduction of coefficients. If we use NTT over 3329 for $c\mathbf{s}_i$, the INTT over 3329 needs modular reduction for 16 coefficients before the final three layers of INTT [HZZ+22].

As for Cortex-M3, since the 16-bit coefficients are loaded into 32-bit registers, they can temporarily surpass 16-bit signed integers but need to be reduced down to 16-bit when they are stored back to 16-bit memory. Because we are using the 3+1+3 layer merging strategy, we just need to ensure that the coefficients can fit in 16-bit signed integers after the third and fourth layers of INTT. As mentioned above, the first four layers of INTT with CT butterfly produce coefficients smaller than $5q_i$, which can fit in 16-bit signed integers for both q_0 and q_1. And the final three layers of INTT with light butterfly would increase the coefficients up to $40.5q_i$. For $q_1 = 3329$, $40.5q_1$ surpass the 16-bit signed integer but fit in 32-bit signed integer, which is allowed on Cortex-M3. These coefficients will be reduced in the modular multiplication with the twiddle factor and 128^{-1} in the final layer of INTT. Therefore, we do not need any modular reduction in INTTs over 769 and 3329 on Cortex-M3.

4.3.3 The explicit CRT implementation with Plantard arithmetic on Cortex-M3

As shown in Algorithm 4, the explicit CRT is used to reconstruct the 32-bit results after INTT. The computation $u = u_0 + \big((u_1 - u_0)\, m_1 \bmod^{\pm} q_1\big)\, q_0$ is performed for 256 pairs of coefficients (u_0 and u_1). During the computation, we can see that $m_1 = q_0^{-1} \bmod^{\pm} q_1$ is a constant. Therefore, the modular multiplication with m_1 modulo q_1 could be optimized with the Plantard multiplication by a constant. The instruction sequence of the explicit CRT with Plantard arithmetic on Cortex-M3 is shown in Algorithm 6. To use the Plantard multiplication by a constant, we need to precompute the term $m_1' = m_1 \cdot (-2^{32} \bmod q_1) \cdot (q_1^{-1} \bmod 2^{32}) \bmod 2^{32}$ so that the Plantard arithmetic could produce result in normal domain. After Step 5 in Algorithm 6, the modular result t using the Plantard arithmetic is in range $[-(q_1+1)/2, q_1/2)$. The rest of the computation $u = u_0 + t \cdot q_0$ would be much smaller than the modulus q of Dilithium; so no further modular reductions are required. Overall, using the Plantard arithmetic in the explicit CRT computation could save one multiplication for each coefficient compared to the Montgomery-based implementation.

4.3.4 Asymmetric multiplication for pointwise multiplication

The pointwise multiplication of incomplete NTTs over 769 and 3329 requires computing 128 degree-1 polynomial multiplications over $X^2 - \zeta_0^{2br_7(i)+1}$ and $X^2 - \zeta_1^{2br_7(i)+1}$ for $q_0 = 769$ and $q_1 = 3329$, respectively, with $i = 0, 1, \ldots, 127$ and $br_7(i)$ denotes the bit reversal of the 7-bit integer i. For simplicity, we denote the corresponding $\zeta_0^{2br_7(i)+1}$ and $\zeta_1^{2br_7(i)+1}$ as ζ below. We adopt the asymmetric multiplication proposed in [ACC$^+$22, AHKS22] to speed up the degree-1 pointwise multiplication. For the polynomial multiplication ct, after the NTT transform, both c and t are transformed into NTT-domain in the form of $\hat{c}_{2i} + \hat{c}_{2i+1}X$ and $\hat{t}_{2i} + \hat{t}_{2i+1}X$. Then, the pointwise multiplication over $X^2 - \zeta$ is computed as $(\hat{c}_{2i} + \hat{c}_{2i+1}X) \circ (\hat{t}_{2i} + \hat{t}_{2i+1}X) = (\hat{c}_{2i}\hat{t}_{2i} + \hat{c}_{2i+1}\hat{t}_{2i+1}\zeta) + (\hat{c}_{2i}\hat{t}_{2i+1} + \hat{c}_{2i+1}\hat{t}_{2i})X$. Note that c is reused for $c\mathbf{s}_1$, $c\mathbf{s}_2$, $c\mathbf{t}_0$, and $c\mathbf{t}_1$, we can precompute the term $\hat{c}_{2i+1}\zeta$ with Plantard multiplication by a constant in advance to reduce repeated computations. Then, two Plantard reductions are used to perform $\hat{c}_{2i} \cdot \hat{t}_{2i} + \hat{c}_{2i+1} \cdot \hat{t}_{2i+1}\zeta \bmod q_i$ and $\hat{c}_{2i} \cdot \hat{t}_{2i+1} + \hat{c}_{2i+1} \cdot \hat{t}_{2i} \bmod q_i$, and all coefficients are reduced down to $[-(q_i+1)/2, q_i/2)$.

4.4 Security and extensibility discussions

We ensure that all the secret key related operations in `sign`, i.e., $c\mathbf{s}_i$ and $c\mathbf{t}_0$, are constant-time. Therefore, our implementation is secure against simple power analysis (SPA) side-channel attacks and timing attacks. In terms of extensibility, the proposed multi-moduli NTT and small 16-bit NTTs for Dilithium can be adapted to other platforms like AVX2, given that the 16-bit NTT[5] on AVX2 is 4.11× faster than the 32-bit NTT[6]. However, since the Plantard arithmetic introduces the 16×32-bit multiplication, which can not be efficiently implemented on AVX2. Consequently, employing the Plantard arithmetic in the 16-bit NTTs might not be the optimal choice on AVX2. Reusing the state-of-the-art 16-bit NTT with Montgomery arithmetic from Kyber's AVX2 reference implementation would still yield performance improvements for the small polynomial multiplications in Dilithium. Regarding the optimization techniques for Keccak, although they are tightly linked to the ARMv7-M architecture, they can be of significant interest for all PQC schemes whose performances are dominated by hashing calculations.

5 Results and Comparisons

5.1 Benchmark settings

For our benchmarks, we used the development boards described below. All implementations were compiled using `arm-none-eabi-gcc 10.2.1` along with the `-O3` optimization flag. We adopted the same configuration as the pqm3[7] and pqm4 [KRSS19], and the clock cycles were measured using the pqm3 and pqm4 frameworks on Cortex-M3 and M4, respectively.

ATSAM3X8E. It features a Cortex-M3 running at 84MHz, 512KB of flash memory and 96KB of SRAM. The core was clocked at 16 MHz to execute code from flash with zero-wait state.

STM32F407VG. It features a Cortex-M4 running up to 168 MHz, 1024 KB of flash memory and 192 KB of RAM divided into three blocks: two contiguous blocks of SRAM connected to the bus matrix with different interconnects, and a core coupled memory (CCM) block which is connected directly to the core. We exclusively used the 64 KB CCM

[5] https://github.com/pq-crystals/kyber
[6] https://github.com/pq-crystals/dilithium
[7] https://github.com/mupq/pqm3

Table 2: Keccak-p[1600, 24] benchmark on Cortex-M3 and M4.

Ref.	**Implementation characteristics***		**Speed** (clock cycles)		**Code size**	**RAM**
	`ldr/str`	lazy `ror`	M3	M4	(bytes)	(bytes)
XKCP	mostly grouped	✗	13 015	11 725	5 576	**264**
This work	grouped	✗	10 785	10 219	5 772	**264**
	grouped	✓ (3/4)	9 981	9 415	6 556	**264**
	grouped	✓ (4/4)	**9 789**	**9 218**	9 536	**264**

*All listed implementations take advantage of the in-place processing and bit-interleaving techniques.

to achieve the best performance with a clock core speed of 24 MHz to execute code from flash with zero-wait state.

5.2 Performance of Keccak and polynomial-related operations

Keccak permutation. Table 2 provides the benchmark of Keccak-p[1600, 24] for different implementation characteristics. The first one, denoted by `ldr/str`, refers to the memory access pipelining strategy. In this context, the term "mostly grouped" signifies that while the majority of memory accesses are packed together; this is not the case during the parity lanes precomputation. Another characteristic indicates the utilization and degree of lazy rotations. The pipeline optimizations achieved by our new register allocation result in significant performance improvements of 17.13 % and 12.84 % on Cortex-M3 and M4, respectively. Note that it comes at the cost of a slight code size increase due to the fact that some `ldr` instructions are now using the higher-half registers (i.e. `r8` to `r14`) as the destination, leading to a 32-bit encoding (versus 16-bit when using lower-half registers). When combined with the use of lazy rotations, we achieve up to 24.78 % and 21.4 % performance boosts on Cortex-M3 and M4, respectively. Nevertheless, the majority of the remaining instructions that were initially encoded on 16 bits have now been 32-bit encoded. This is because the second operands undergo systematic rotation using the inline barrel shifter. As discussed in Section 3.3, the variant that utilizes lazy rotations for all rounds incurs a 50% increase in code size for a minor performance gain of aroung 2%. While we could have potentially made it more compact, we believe that doing so would have compromised the performance advantage, resulting in no overall benefits compared to the other variant. Therefore, when code size is a critical factor, we suggest to favour the implementation that uses lazy rotations for three-quarters of the rounds only in order to minimize the memory footprint, which is also adopted in the following benchmark.

NTT-related functions. Table 3 presents results for NTT-related functions on Cortex-M3 and M4. Due to the instruction limits of Cortex-M3, the 16-bit NTT on Cortex-M3 is significantly faster than both constant-time and variable-time 32-bit NTT. To be more specific, the 16-bit NTT, INTT, and pointwise multiplication are 4.22×, 4.29×, and 2.14× faster than the constant-time 32-bit NTT, INTT, and pointwise multiplication, respectively. Compared to the 32-bit variable-time NTT, INTT, and pointwise multiplication, the speed-ups are 2.48×, 2.46×, and 1.24×, respectively.

The highly efficient 16-bit NTT implementations makes the proposed multi-moduli NTT practical on Cortex-M3. Compared to the constant-time 32-bit NTT, the proposed multi-moduli NTT, INTT and pointwise multiplication implementations yield 52.76% $\sim$ 54.76% performance improvements. Besides, the multi-moduli NTT and INTT also obtain 19.47% and 19.07% speed-ups compared with the variable-time 32-bit NTT and INTT. The pointwise multiplication is 63.05% slower than theirs but we will later show that our

Table 3: Performance of NTT-related functions on Cortex-M3 and M4. Averaged over 1000 executions.

Platform	Prime	Ref.	NTT	INTT	Pointwise	CRT
M3	8380417	[GKS20] constant-time	33 077	36 661	8 528	✗
	8380417	[GKS20] variable-time	19 405	21 051	4 944	✗
	3329×7681	[ACC+22] (Saber)	16 770	19 056	11 927	4 637
	769	This work	7 830	8 543	3 989	✗
	769×3329	This work	15 626	17 037	8 061	3 735
M4	8380417	[AHKS22]	8 066	8 388	1 931	✗
	257	[AHKS22]	5 497	5 540	1 201	✗
	769	[AHKS22]	5 180	5 512	1 715	✗
	769	This work	4 456	4 593	1 717	✗

Table 4: Performance of small polynomial multiplications on Cortex-M3 and M4. The secret vector $\mathbf{s}_1$ is l-dimensional vector while the other three vectors ($\mathbf{s}_2, \mathbf{t}_0,$ and $\mathbf{t}_1$) are all k-dimensional vectors. Averaged over 1000 executions.

Platform	Operation	Dilithium2		Dilithium3		Dilithium5	
		[GKS20]	This work	[GKS20]	This work	[GKS20]	This work
M3	$c\mathbf{s}_1$	346k	106k	424k	128k	580k	172k
	$c\mathbf{s}_2$	346k	106k	502k	150k	658k	194k
	$c\mathbf{t}_0$	269k	195k	328k	284k	446k	372k
	$c\mathbf{t}_1$	213k	195k	311k	284k	409k	372k
		[AHKS22]	This work	[AHKS22]	This work	[AHKS22]	This work
M4	$c\mathbf{s}_1$	56k	–	79k	70k	92k	–
	$c\mathbf{s}_2$	56k	–	89k	78k	110k	–

multi-moduli NTT implementation for small polynomial multiplications is still better than the one that uses variable-time 32-bit NTT. We also compare our multi-moduli NTT implementation with a similar implementation for Saber in [ACC+22]. Although [ACC+22] adopted a faster 6-layer NTT/INTT, our 7-layer NTT/INTT implementation using the efficient Plantard arithmetic is still slightly faster, obtaining 6.82%, 10.60%, and 32.41% speed-ups for NTT, INTT, and pointwise multiplication, respectively. Moreover, since the explicit CRT can also be optimized with the efficient Plantard multiplication by a constant, our CRT implementation is also 19.45% faster. In summary, using Plantard arithmetic in NTT, INTT and CRT makes the multi-moduli NTT more efficient on Cortex-M3.

As for Cortex-M4, we integrate the efficient Plantard arithmetic for the NTT over 769. As shown in Table 3, our NTT and INTT implementations achieves 13.98% $\sim$ 18.94% speed-ups compare with FNT over 257 and NTT over 769 in [AHKS22], respectively. The speed-ups of 16-bit NTT and INTT versus 32-bit NTT and INTT are 44.76% and 45.24%, respectively. Because the proposed multi-moduli NTT implementation requires to perform double 16-bit NTTs, INTTs, and pointwise multiplications, the 16-bit NTT/INTT needs to be at least 50% faster than the 32-bit NTT/INTT in order to make it faster than the 32-bit NTT. Therefore, the multi-moduli NTT is not applicable on Cortex-M4.

Small polynomial multiplications. Table 4 compares the performance of small polynomial multiplications $c\mathbf{s}_i$ and $c\mathbf{t}_i$ on Cortex-M3 and M4. On Cortex-M3, there are three different

Table 5: Performance of Dilithium on Cortex-M3. Averaged over 1000 executions.

Operation	Dilithium2		Dilithium3		Dilithium5	
	[GKS20]	This work	[GKS20]	This work	[GKS20]	This work
`keygen`	$2\,059k$	$1\,739k$	$3\,594k$	$3\,011k$	✗	$5\,034k$
`sign`	$7\,139k$	$5\,582k$	$11\,916k$	$9\,087k$	✗	$20\,193k$
`verify`	$1\,949k$	$1\,648k$	$3\,283k$	$2\,755k$	✗	$4\,694k$

implementation strategies based on whether the operations are secret-related operations. The $c\mathbf{s}_i$ is secret-related and previous work [GKS20] implements it with constant-time 32-bit NTT on Cortex-M3. Compared to their implementation with constant-time 32-bit NTT, our $c\mathbf{s}_i$ implementation with 16-bit NTT is not only constant-time but also $3.26\times \sim 3.39\times$ faster. Since Dilithium's security does not rely on $\mathbf{t}_i$ being secret, the $c\mathbf{t}_0$ implementation in [GKS20] is not totally constant-time. They implement both c and $\mathbf{t}_0$ with 32-bit constant-time NTT, while the pointwise multiplication and INTT are implemented in variable-time. Compared to their $c\mathbf{t}_0$ implementation, our multi-moduli NTT implementation is constant-time and shows 27.51%, 13.41%, and 16.59% speed-ups for three variants of Dilithium, respectively. Besides, despite the $c\mathbf{t}_1$ implementation in `verify` in [GKS20] is totally variable-time, our multi-moduli NTT implementation still obtains 8.45%, 8.68%, and 9.05% speed-ups for three variants of Dilithium, respectively. However, because using multi-moduli NTT in `verify` would introduce an extra INTT for a k-dimensional polynomial vector, we did not apply it to `verify`. As for Cortex-M4, by utilizing the efficient Plantard arithmetic on the NTT over 769 for Dilithium3, we yield 11.39% and 12.36% speed-ups for $c\mathbf{s}_1$ and $c\mathbf{s}_2$ on Cortex-M4. Since we reuse the FNT over 257 for Dilithium2 and Dilithium5, we do not provide comparison of $c\mathbf{s}_i$ for them.

5.3 Performance of schemes

Performance on Cortex-M3. Combining the optimizations of Keccak and polynomial multiplication, our Dilithium implementation shows significant improvements compared with the state-of-the-art implementation on Cortex-M3. As shown in Table 5, we yield 15.54% and 16.22% speed-ups for the `keygen` of Dilithium2 and Dilithium3, respectively. The speed-ups for the `verify` of Dilithium2 and Dilithium3 are 15.44% and 16.08%, respectively. The speed-ups for `sign` are larger than `keygen` and `verify` thanks to our polynomial multiplication optimizations, ranging from 21.81% and 23.74% for Dilithium2 and Dilithium3, respectively. Since Dilithium5 consumes a large stack usage and it cannot be directly deployed in Cortex-M3, we adopt a basic on-the-fly matrix generation strategy, which is similar to streaming $\mathbf{A}$ in [GKS20, Section 5.3], to avoid storing the whole matrix; thus making Dilithium5 deployable on Cortex-M3. Since both pqm3 and [GKS20] did not provide Dilithium5 implementation on Cortex-M3, we could not conduct any comparison.

Performance and hash profiling on Cortex-M4. Because Keccak is extensively used by the PQC algorithms selected by NIST for standardization, our Keccak optimizations can also improve their performance on ARMv7-M. To highlight the improvements brought by our Keccak optimizations, we provide benchmarks and hash profiling based on the pqm4[KRSS19] for both Dilithium and Kyber on Cortex-M4. As shown in Table 6, for most of the Dilithium variants, the Keccak optimizations yield speed-ups ranging from 13.94% to 15.52%. However, because the Keccak permutation is not the primary operation in the rejection sampling process, the speed-up on the `sign` algorithm of Dilithium5 is relatively small, around 9.42%. Similarly, the speed-version Kyber KEM protocols in

Table 6: Performance and hash profiling of Kyber and Dilithium on the Cortex-M4 using the pqm4 framework. Averaged over 1000 executions.

Scheme	**Keccak Impl.**	**keygen**		**sign/encaps**		**verify/decaps**	
		speed	hashing	speed	hashing	speed	hashing
Dilithium2	XKCP	$1\,595k$	83.47%	$4\,052k$	64.53%	$1\,576k$	80.47%
	This work	$1\,357k$	80.57%	$3\,487k$	60.02%	$1\,350k$	77.2%
Dilithium3	XKCP	$2\,828k$	85.54%	$6\,523k$	62.95%	$2\,702k$	82.62%
	This work	$2\,394k$	82.92%	$5\,574k$	58.97%	$2\,302k$	79.61%
Dilithium5	XKCP	$4\,817k$	86.6%	$8\,534k$	68.08%	$4\,714k$	84.69%
	This work	$4\,069k$	84.14%	$7\,730k$	63.05%	$3\,998k$	81.95%
Kyber512	XKCP	$432k$	80.12%	$527k$	82.86%	$472k$	73.76%
	This work	$369k$	76.75%	$448k$	79.85%	$409k$	69.74%
Kyber768	XKCP	$704k$	79.04%	$860k$	82.38%	$778k$	74.75%
	This work	$604k$	75.59%	$732k$	79.32%	$674k$	70.84%
Kyber1024	XKCP	$1\,122k$	79.58%	$1\,314k$	82.46%	$1\,208k$	76.07%
	This work	$962k$	76.18%	$1\,119k$	79.41%	$1\,043k$	72.29%

[KRSS19, HZZ+22] also experience speed-ups of 13.35% to 15.00% due to our Keccak optimizations. Additionally, we present the hash profiling results for these two schemes to showcase the efficiency of our Keccak optimizations. As can be seen from Table 6, our Keccak optimizations decrease the proportion of time spent on hashing by 2.46% $\sim$ 5.03%.

Acknowledgments

This work is partially supported by the National Natural Science Foundation of China (62002023, 62002239, 62372417, and 62132008), Guangdong Provincial Key Laboratory IRADS (2022B1212010006, R0400001-22), Guangdong Province General Universities Key Special Fund (New Generation Information Technology) (2023ZDZX1033), Natural Science Foundation of Jiangsu Province (BK20220075), Fok Ying-Tong Education Foundation for Young Teachers in the Higher Education Institutions of China (No.20193218210004), Jiangsu Province 100 Foreign Experts Introduction Plan (BX2022012), TÜBİTAK Projects (2232-118C332 and 1001-121F348), ITF project ITS/098/22, and the InnoHK Project CIMDA.

References

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R, M} LWE schemes. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):336–357, 2020.

[ABKK24] Amin Abdulrahman, Hanno Becker, Matthias J. Kannwischer, and Fabien Klein. Fast and clean: Auditable high-performance assembly via constraint solving. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(1):87–132, 2024.

[ACC+22] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):127–151, 2022.

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In Giuseppe Ateniese and Daniele Venturi, editors, *Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings*, volume 13269 of *Lecture Notes in Computer Science*, pages 853–871. Springer, 2022.

[Bar86] Paul Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986.

[BDH+] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. XKCP: eXtended Keccak Code Package. https://github.com/XKCP/XKCP. commit 7fa59c0.

[BDH+12] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview. https://keccak.team/files/Keccak-implementation-3.2.pdf, 2012. Accessed: 2023-05-26.

[BDH+23] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, Ronny Van Keer, and Benoît Viguier. TurboSHAKE. IACR Cryptol. ePrint Arch., Paper 2023/342, 2023. https://eprint.iacr.org/2023/342.

[BDK+18] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 353–367. IEEE, 2018.

[BHK+22] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):221–244, 2022.

[BK22] Hanno Becker and Matthias J. Kannwischer. Hybrid Scalar/Vector Implementations of Keccak and SPHINCS+ on AArch64. In Takanori Isobe and Santanu Sarkar, editors, *Progress in Cryptology – INDOCRYPT 2022*, pages 272–293, Cham, 2022. Springer International Publishing. https://eprint.iacr.org/2022/1243.

[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT Multiplication for NTT-unfriendly Rings New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):159–188, 2021.

[dG15] Wouter de Groot. *A Performance Study of X25519 on Cortex-M3 and M4*. PhD thesis, Eindhoven University of Technology Eindhoven, The Netherlands, 2015.

[DKL+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):238–268, Feb. 2018.

[Dwo15] Morris Dworkin. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, 2015-08-04 2015.

[GKS20] Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. Compact Dilithium Implementations on Cortex-M3 and Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):1–24, Dec. 2020.

[HZZ$^+$22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved Plantard Arithmetic for Lattice-based Cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):614–636, 2022.

[HZZ$^+$23] Junhao Huang, Haosong Zhao, Jipeng Zhang, Wangchen Dai, Lu Zhou, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Yet another Improvement of Plantard Arithmetic for Faster Kyber on Low-end 32-bit IoT Devices. *CoRR*, abs/2309.00440, 2023.

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4. IACR Cryptol. ePrint Arch., Paper 2019/844, 2019. https://eprint.iacr.org/2019/844.

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 598–616. Springer, 2009.

[Lyu12] Vadim Lyubashevsky. Lattice Signatures without Trapdoors. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*, pages 738–755. Springer, 2012.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[MS90] Peter L Montgomery and Robert D Silverman. An FFT extension to the P-1 factoring algorithm. *Mathematics of Computation*, 54(190):839–854, 1990.

[NIS23a] NIST. FIPS 203 (Initial Public Draft): Module-Lattice-Based Key-Encapsulation Mechanism Standard. https://csrc.nist.gov/pubs/fips/203/ipd, 2023.

[NIS23b] NIST. FIPS 204 (Initial Public Draft): Module-Lattice-Based Digital Signature Standard. https://csrc.nist.gov/pubs/fips/204/ipd, 2023.

[Pla21] Thomas Plantard. Efficient Word Size Modular Arithmetic. *IEEE Trans. Emerg. Top. Comput.*, 9(3):1506–1518, 2021.

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. *IACR Cryptol. ePrint Arch., Paper 2018/039*, 2018. http://eprint.iacr.org/2018/039.

[Sto19] Ko Stoffelen. Efficient Cryptography on the RISC-V Architecture. In *Progress in Cryptology - LATINCRYPT 2019: 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2-4, 2019, Proceedings*, pages 323–340, Berlin, Heidelberg, 2019. Springer-Verlag.

[ZHLR22] Jipeng Zhang, Junhao Huang, Zhe Liu, and Sujoy Sinha Roy. Time-memory trade-offs for Saber+ on memory-constrained RISC-V platform. *IEEE Trans. Computers*, 71(11):2996–3007, 2022.

[ZHS$^+$22] Jieyu Zheng, Feng He, Shiyu Shen, Chenxi Xue, and Yunlei Zhao. Parallel Small Polynomial Multiplication for Dilithium: A Faster Design and Implementation. In *Annual Computer Security Applications Conference, ACSAC 2022, Austin, TX, USA, December 5-9, 2022*, pages 304–317. ACM, 2022.

[ZZH$^+$21] Lirui Zhao, Jipeng Zhang, Junhao Huang, Zhe Liu, and Gerhard Hancke. Efficient implementation of kyber on mobile devices. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 506–513. IEEE, 2021.

[ZZS$^+$23] Jieyu Zheng, Haoliang Zhu, Zhenyu Song, Zheng Wang, and Yunlei Zhao. Optimized Vectorization Implementation of CRYSTALS-Dilithium. *CoRR*, abs/2306.01989, 2023.