Parallel Small Polynomial Multiplication for Dilithium: A Faster Design and Implementation

Jieyu Zheng
Department of Computer Science
Fudan university
Shanghai, China
jieyuzheng21@m.fudan.edu.cn

Feng He
Department of Computer Science
Fudan university
Shanghai, China
fhe20@fudan.edu.cn

Shiyu Shen
Department of Computer Science
Fudan University
Shanghai, China
shenshiyu21@m.fudan.edu.cn

Chenxi Xue
Department of Computer Science
Fudan University
Shanghai, China
cxxue21@m.fudan.edu.cn

Yunlei Zhao*
Department of Computer Science
Fudan University
Shanghai, China
ylzhao@fudan.edu.cn

ABSTRACT

The lattice-based signature scheme CRYSTALS-Dilithium is one of the two signature finalists in the third round NIST post-quantum cryptography (PQC) standardization project. For applications of low-power Internet-of-Things (IoT) devices, recent research efforts have been focusing on the performance optimization of PQC algorithms on embedded systems. In particular, performance optimization is more demanding for PQC signature algorithms that are usually significantly more time-consuming than PQC public-key encryption counterparts. For most cryptographic algorithms based on algebraic lattices including Dilithium, the fundamental and most time-consuming operation is polynomial multiplication over rings. For this computational task, number theoretic transform (NTT) is the most efficient multiplication method for NTT-friendly rings, and is now the typical technique for performing fast polynomial multiplications when implementing lattice-based PQC algorithms.

The key observation of this work is that, besides multiplications of polynomials of standard forms, Dilithium involves a list of multiplications for polynomials of very small coefficients. Can we have more efficient methods for multiplying such polynomials of small coefficients? Under this motivation, we present in this work a parallel small polynomial multiplication algorithm to speed up the implementations of Dilithium. We complete both C reference implementation and ARM Neon implementation. Moreover, we conducted some speed tests in combination with Becker's Neon NTT [4]. The results show that, in comparison with the C reference implementation of Dilithium submitted to the third round of the NIST PQC competition, our reference implementation with the proposed parallel small polynomial multiplication is faster: specifically, our Sign and Verify speed up 18% and 19% respectively for Dilithium-2 (30% and 7% for Dilithium-3, 27% and 3% for Dilithium-5, respectively). As for the Arm Neon implementation, we achieved a performance improvement of about 64% in Sign and 50% in Verify for Dilithium-2 (60% and 32% for Dilithium-3) compared with the C reference implementation of Dilithium submitted to the third round of the NIST PQC competition. We aslo compared our work with the state-of-the-art Arm Neon implementation of Dilithium [4], the results show our speed of Sign is 13.4% faster for Dilithium-2 and 8.0% faster for Dilithium-3, achieving a new record of fast Dilithium implementation.

CCS CONCEPTS

• **Security and privacy** → **Cryptography**.

KEYWORDS

Post-quantum cryptography, CRYSTAL-Dilithium, Digital signature, Arm Cortex A72, Polynomial multiplication

ACM Reference Format:
Jieyu Zheng, Feng He, Shiyu Shen, Chenxi Xue, and Yunlei Zhao. 2022. Parallel Small Polynomial Multiplication for Dilithium: A Faster Design and Implementation. In *Annual Computer Security Applications Conference (ACSAC), December 5–9, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3564625.3564629

1 INTRODUCTION

According to Shor's algorithm [32], almost all the current public-key cryptographic systems based on factoring and discrete logarithm will be broken once large-scale quantum computers become realistic. At present, there are five main types of post-quantum cryptography: lattice-based [24], code-based [23], multivariate-based [7], and isogeny-based [27]. Among them, lattice-based cryptography is commonly viewed as promising because its functionalities, security, size of key and ciphertext, and computation efficiency all perform well. In December 2016, the National Institute of Standards

*Corresponding author.

ACSAC, December 5–9, 2022, Austin, TX, USA

ACM ISBN 978-1-4503-9759-9/22/12…$15.00
https://doi.org/10.1145/3564625.3564629

and Technology (NIST) called for post-quantum cryptographic algorithm standards around the world, which is referred to as the NIST-PQC project for presentation simplicity. In the third round of the NIST-PQC project, there were seven finalists among which Dilithium [3] is one of the two digital signature finalists.

For applications of low-power Internet-of-Things (IoT) devices, recent research efforts have been focusing on the performance optimization of PQC algorithms on embedded systems. In particular, performance optimization is more demanding for PQC signature algorithms that are usually significantly more time-consuming than PQC public-key encryption counterparts. For most cryptographic algorithms based on algebraic lattices, the fundamental and most time-consuming operation is polynomial multiplication over rings. For this computational task, number theoretic transform (NTT) is the most efficient multiplication method for NTT-friendly rings. NTT is now the typical technique for performing fast polynomial multiplications when implementing lattice-based PQC algorithms, particularly for Dilithium that is defined over NTT-friendly rings. Recently, many studies have been conducted to improve the implementation performance of Dilithium both in software and in hardware [1, 4, 5, 9–17, 19, 28–30, 33, 34]. The work [1] proposed an implementation that applies NTT with a much smaller modulus q' for computing the products $\mathbf{c} \cdot \overrightarrow{\mathbf{s}}, \mathbf{c} \cdot \overrightarrow{\mathbf{e}}$ in Dilithium. The work [34] achieved a compact and high-performance hardware architecture for Dilithium. In order to adapt to low-power Internet-of-Things (IoT) devices, many works focused on the performance optimization of NIST-PQC algorithms on embedded systems. For ARM Cortex-M3 and Cortex-M4, the works of [6] and [2] worked on Kyber, [11] worked on Dilithium, and [1] worked on both Kyber and Dilithium. For ARM Cortex-A, the works of [25] and [31] had used Neon vector extension on lattice-based cryptography. [4] optimized Dilithium, Kyber and Saber with Armv8-A Neon vector instructions.

However, all the existing studies focus on NTT implementation optimizations or reduction algorithm improvements. In Dilithium Sign procedure and Verify procedure, we need to compute $\mathbf{c} \cdot \overrightarrow{\mathbf{s}} \in \mathcal{R}_q^{\ell}$ and $\mathbf{c} \cdot \overrightarrow{\mathbf{e}}, \mathbf{c} \cdot \overrightarrow{\mathbf{t}}_0, \mathbf{c} \cdot \overrightarrow{\mathbf{t}}_1 \in \mathcal{R}_q^k$. Although their coefficients are significantly small compared with the modulus q (for instance, $\|\mathbf{c}\|_\infty = 1$ and $\|\overrightarrow{\mathbf{s}}\|_\infty = \|\overrightarrow{\mathbf{e}}\|_\infty = 2 \ll q = 8380417$), the NTT technique is still applied with respect to the foregoing evaluations. A natural question arises as to whether we can compute these products in a faster way, by fully utilizing the fact that their magnitudes are much smaller than q.

In this paper, by exploiting the feature of the challenge polynomial $\mathbf{c}$ that has exactly τ number of ± 1's and the rest 0's where τ is fairly smaller than the polynomial dimension $n = 256$, we present the parallel small polynomial multiplication methods that can fastly compute all the product results of $\mathbf{c} \cdot \overrightarrow{\mathbf{s}}, \mathbf{c} \cdot \overrightarrow{\mathbf{e}}, \mathbf{c} \cdot \overrightarrow{\mathbf{t}}_0$, and $\mathbf{c} \cdot \overrightarrow{\mathbf{t}}_1$. We still use NTTs for computing the products of the remaining polynomials that are of the standard form. The results show that our parallel small polynomial multiplication algorithms are faster than NTT when computing these small polynomial multiplications. Therefore, we have a total improvement in Dilithium Sign and Verify procedure, achieving a new record of fast implementation of Dilithium.

We complete both C reference implementation and ARM Neon implementation of Dilithium with the proposed algorithm. Moreover, we do some speed tests in combination with Becker's Neon NTT. The results show that, in comparison with the C reference implementation of Dilithium submitted to the final round of the NIST PQC competition, our reference implementation with the proposed parallel small polynomial multiplication is faster: specifically, our Sign and Verify speed up 18% and 19% respectively for Dilithium-2 (30% and 7% for Dilithium-3, 27% and 3% for Dilithium-5, respectively). As for the Arm Neon implementation, we achieved a performance improvement of about 64% in Sign and 50% in Verify for Dilithium-2 (60% and 32% for Dilithium-3) with the C reference implementation of Dilithium submitted to the third round of the NIST PQC competition. Compared with the state-of-the-art Arm Neon implementation of Dilithium [4], our speed of Sign is 13.4% faster for Dilithium-2 and 8.0% faster for Dilithium-3.

2 PRELIMINARIES

Given any real number $x \in \mathbb{R}$, let $\lfloor x \rfloor$ denote the largest integer that is no more than x, and $\lfloor x \rceil := \lfloor x + 1/2 \rfloor$. For the positive integers $r, \alpha > 0$, let $r \bmod \alpha$ denote the unique integer $r' \in \{0, \cdots, \alpha - 1\}$ such that $\alpha \mid (r' - r)$, and let $r \bmod^{\pm} \alpha$ denote the unique integer $r'' \in \left\{ -\left\lfloor \frac{\alpha-1}{2} \right\rfloor, \cdots, 0, \cdots, \left\lfloor \frac{\alpha}{2} \right\rfloor \right\}$ such that $\alpha \mid (r'' - r)$.

In this work, let n be a power-of-two, and let q be a positive rational prime such that $q \equiv 1 \pmod{2n}$. By default, in Dilithium, we have $n = 256$ and $q = 8380417$.

Let $\mathbb{Z}_q \stackrel{\text{def}}{=} \mathbb{Z}/q\mathbb{Z}$, and $\mathcal{R} \stackrel{\text{def}}{=} \mathbb{Z}[x]/\langle x^n + 1 \rangle$; moreover, we define $\mathcal{R}_q \stackrel{\text{def}}{=} \mathcal{R}/q\mathcal{R} \cong \mathbb{Z}_q[x]/\langle x^n + 1 \rangle$. Every element $a \in \mathbb{Z}_q$ can be represented by a unique element in $\left\{ -\frac{q-1}{2}, \cdots, 0, \cdots, \frac{q-1}{2} \right\}$. Similarly, every element $\mathbf{a} \in \mathcal{R}$ can be uniquely written as $\mathbf{a} = \sum_{i=0}^{n-1} a_i \cdot x^n, a_i \in \mathbb{Z}$, whereas every element $\mathbf{a} \in \mathcal{R}_q$ can be uniquely written as $\mathbf{a} = \sum_{i=0}^{n-1} a_i \cdot x^n, a_i \in \mathbb{Z}_q$. Finally, (column) vectors over $\mathcal{R}$ (or $\mathcal{R}_q$) are represented by symbols like $\overrightarrow{\mathbf{a}}$, and matrices over $\mathcal{R}$ (or $\mathcal{R}_q$) are written in uppercase boldface letters (*e.g.*, $\mathbf{M}$).

For the element $a \in \mathbb{Z}_q$, we write $\|a\|_\infty$ for $|a \bmod^{\pm} q|$ (*i.e.*, the absolute value of $a \bmod^{\pm} q$), and define $\mathsf{Power2Round}_{q,d}(a) \stackrel{\text{def}}{=} (a_1, a_0)$, where $a_0 \stackrel{\text{def}}{=} a \bmod^{\pm} 2^d$ and $a_1 \stackrel{\text{def}}{=} (a - a_0)/2^d$. These notations can be naturally generalized to $\mathbf{a} = \sum a_i \cdot x^i \in \mathcal{R}_q$ in the component-wise manner, *i.e.*, $\mathsf{Power2Round}_{q,d}(\mathbf{a}) \stackrel{\text{def}}{=} (\mathbf{a}_0, \mathbf{a}_1)$.

For a finite set S, $|S|$ denotes its cardinality, and $x \leftarrow S$ denotes the randomized operation of picking an element uniformly at random from the set S. We use standard notations and conventions below for writing probabilistic algorithms, experiments and interactive protocols. For an arbitrary probability distribution $\mathcal{D}$, the notation $x \leftarrow \mathcal{D}$ denotes the operation of picking an element according to the pre-defined distribution $\mathcal{D}$. If α is neither an algorithm nor a set, then $x \leftarrow \alpha$ is a simple assignment statement, which could also be written as $x := \alpha$ in this case. If A is a probabilistic algorithm, then $A(x_1, x_2, \cdots; r)$ represents the result of running the algorithm A on inputs $x_1, x_2, \cdots$ as well as the random coins r. We let $y \leftarrow A(x_1, x_2, \cdots)$ denote the experiment of picking r at random and outputting $y := A(x_1, x_2, \cdots; r)$. By $\Pr[R_1; \cdots; R_n : E]$ we denote the probability of event E, after the ordered execution of random processes $R_1, \cdots, R_n$. We say that a positive function $f(\lambda) > 0$ is *negligible* in λ, if for every $c > 0$ there exists a positive real $\lambda_c > 0$ such that $f(\lambda) < 1/\lambda^c$ for all $\lambda > \lambda_c$.

2.1 Digital Signature Scheme

A digital signature scheme Π consists of three probabilistic polynomial-time algorithms $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$.

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$. KeyGen is the key generation algorithm that, on input the security parameter 1^λ, outputs $(\mathsf{pk}, \mathsf{sk})$.
- $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, \mu)$. Sign is the signing algorithm that, on input the secret key sk as well as the message $\mu \in \{0,1\}^*$ to be signed, outputs the signature σ.
- $b := \mathsf{Verify}(\mathsf{pk}, \mu, \sigma)$. Verify is the *deterministic* verification algorithm that, on input the public key pk as well as the message / signature pair (μ, σ), outputs $b \in \{0,1\}$, indicating whether it accepts the incoming (μ, σ) as a *valid* one (*i.e.*, $b = 1$) or not (*i.e.*, $b = 0$).

We say a signature scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ is *correct*, if any sufficiently large λ, any $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ and any $\mu \in \{0,1\}^*$, it holds

$$\Pr[\mathsf{Verify}(\mathsf{pk}, \mu, \mathsf{Sign}(\mathsf{sk}, \mu)) = 1] = 1.$$

2.2 Module-LWE and Module-SIS

For the element $\mathbf{w} = \sum_{i=0}^{n-1} w_i x^i \in \mathcal{R}$, its ℓ_∞-norm is defined as $\|\mathbf{w}\|_\infty := \max \|w_i\|_\infty$. Likewise, for the element $\overrightarrow{\mathbf{w}} = [\mathbf{w}_i]_i \in \mathcal{R}^k$, its ℓ_∞-norm is defined as $\left\|\overrightarrow{\mathbf{w}}\right\|_\infty := \max_i \left\|\overrightarrow{\mathbf{w}}_i\right\|_\infty$. In particular, when the other parameters are clear from the context, let $S_\eta \subseteq \mathcal{R}$ denote the set of elements $\mathbf{w} \in \mathcal{R}$ such that $\|\mathbf{w}\|_\infty \leq \eta$.

The hard problems underlying the security of the digital signature Dilithium are Module-LWE (MLWE), Module-SIS (MSIS) (as well as a variant of MSIS problem). They were well studied in [18] and could be seen as a natural generalization of the Ring-LWE [22] and Ring-SIS problems [21, 26], respectively.

Fix the parameter $\ell \in \mathbb{N}$. The Module-LWE distribution (induced by $\overrightarrow{\mathbf{s}} \in \mathcal{R}_q^\ell$) is the distribution of the random pair $(\overrightarrow{\mathbf{a}_i}, \mathbf{b}_i)$ over the support $\mathcal{R}_q^\ell \times \mathcal{R}_q$, where $\overrightarrow{\mathbf{a}_i} \leftarrow \mathcal{R}_q^\ell$ is taken uniformly at random, and $\mathbf{b}_i := \overrightarrow{\mathbf{a}_i}^T \cdot \overrightarrow{\mathbf{s}} + \mathbf{e}_i$ with $\mathbf{e}_i \leftarrow S_\eta$ taken fresh for every sample. Given arbitrarily many samples drawn from the Module-LWE distribution induced by $\overrightarrow{\mathbf{s}} \leftarrow S_\eta^\ell$, the (search) Module-LWE problem asks to recover $\overrightarrow{\mathbf{s}}$. And the associated Module-LWE assumption states that given $\mathbf{A} \leftarrow \mathcal{R}_q^{k\times\ell}$ and $\overrightarrow{\mathbf{b}} := \mathbf{A}\overrightarrow{\mathbf{s}} + \overrightarrow{\mathbf{e}}$ where $k = \mathrm{poly}(\lambda)$ and $(\overrightarrow{\mathbf{s}}, \overrightarrow{\mathbf{e}}) \leftarrow S_\eta^\ell \times S_\eta^k$, no efficient algorithm can succeed in recovering $\overrightarrow{\mathbf{s}}$ with non-negligible probability, provided that the parameters are appropriately chosen.

Fix $p \in [1, \infty]$. Given $\mathbf{A} \leftarrow \mathcal{R}_q^{k\times\ell}$ where $k = \mathrm{poly}(\lambda)$, the Module-SIS problem (in ℓ_p-norm) parameterized by $\beta > 0$ asks to find a "short" yet nonzero pre-image $\overrightarrow{\mathbf{x}} \in \mathcal{R}_q^\ell$ in the lattice determined by $\mathbf{A}$, *i.e.*, $\overrightarrow{\mathbf{x}} \neq \mathbf{0}$, $\mathbf{A} \cdot \overrightarrow{\mathbf{x}} = \mathbf{0}$ and $\left\|\overrightarrow{\mathbf{x}}\right\| \leq \beta$. And the associated Module-SIS assumption (in ℓ_p-norm) states that no probabilistic polynomial-time algorithm can find a feasible pre-image $\overrightarrow{\mathbf{x}}$ with non-negligible probability, provided that the parameters are appropriately chosen. In the literature, the module-SIS problem in *Euclidean* norm, *i.e.*, $p = 2$, is well-studied; nevertheless, in Dilithium, we are mostly interested in the Module-SIS problem / assumption *in ℓ_∞-norm*, *i.e.*, $p = \infty$.

Algorithm 1 Pick an element from the set $B_\tau \subseteq \mathcal{R}$ uniformly at random

Input: $\mathbf{c} = c_0 c_1 \ldots c_{255} = 00\ldots0$

Output: $\mathbf{c} = \sum_{i=0}^{255} c_i \cdot x^i$

for $i \in \{256 - \tau, \cdots, 255\}$ **do**
  $j \leftarrow \{0, 1, \cdots, i\}$
  $b \leftarrow \{0, 1\}$
  $c_i := c_j$
  $c_j := (-1)^b$
end for
return $\mathbf{c} = \sum_{i=0}^{255} c_i \cdot x^i$

2.3 Hashing

As is in [8, 20], when the other related parameters are clear from the context, for every positive integer $\tau > 0$, let B_τ denote the subset of $\mathcal{R}$ consisting of elements of $\mathcal{R}$ with τ nonzero coefficients that are either -1 or 1 and the rest are 0; equivalently,

$$B_\tau = \left\{\mathbf{x} \in \mathcal{R} \,\middle|\, \|\mathbf{x}\|_\infty = 1, \|\mathbf{x}\|_0 = \tau\right\} \subseteq \mathcal{R}.$$

When the positive integer τ is fixed, let $H : \{0,1\}^* \to B_\tau$ denote a hash function that is modeled as a random oracle in this work. In practice, to pick a random element in B_τ, we can use an inside-out version of Fisher-Yates shuffle. as is done in [8, 20]. See Algorithm 1 for the full detail.

2.4 Extendable Output Function

The notion of extendable output function is applied in [8]. An *extendable output function* Sam is a function on bit string in which the output can be extended to any desired length, and the notation $y \in S := \mathsf{Sam}(x)$ represents that the function Sam takes as input x and then produces a value y that is distributed according to the pre-defined distribution S (or according to the uniform distribution over the pre-defined set S). The whole procedure is *deterministic* in the sense that for a given x will always output the same y, *i.e.*, the map $x \mapsto y$ is well-defined. For simplicity we always assume that the output distribution of Sam is perfect, whereas in practice it will be implemented by using some cryptographic hash functions (which are modelled as random oracle in this work) and produce an output that is *statistically close* to the perfect distribution.

2.5 Specification of Dilithium

Dilithium is a digital signature scheme based on algebraic lattice, which can be proven tightly secure in the quantum random oracle model under lattice assumptions. It plays a leading role in the NIST-PQC project, and is one of the two signature finalists in the third round of NIST-PQC. Please refer to Algorithms 2, 3 and 4 for the algorithmic specifications of Dilithium. The rounding functions Power2Round, HighBits and Decompose, and the hint functions MakeHint and UseHint, please see the paper [3].

Dilithium is parameterized by

$$n, q, k, \ell, d, \eta, \gamma_1, \gamma_2, \beta, \omega, \tau.$$

In particular, note that the vectors of polynomials $\overrightarrow{\mathbf{s}}, \overrightarrow{\mathbf{e}}, \overrightarrow{\mathbf{t}}_1, \overrightarrow{\mathbf{t}}_0$ are all "small" polynomials in the following sense:

- For every polynomial in $\overrightarrow{\mathbf{s}} \in \mathcal{R}_q^k$ and in $\overrightarrow{\mathbf{e}} \in \mathcal{R}_q^\ell$ each coefficient belongs to centrally symmetric "small" set
$$\{-\eta, 1-\eta, \cdots, 0, \cdots, \eta-1, \eta\};$$
- For every polynomial in $\overrightarrow{\mathbf{t}}_0$, each coefficient belongs to the *almost* centrally symmetric "small" set
$$\left\{1-2^{d-1}, \cdots, 0, \cdots, 2^{d-1}\right\} \subseteq \left\{-2^{d-1}, \cdots, 2^{d-1}\right\};$$
- For every polynomial in $\overrightarrow{\mathbf{t}}_1$, each coefficient belongs to the "small" set
$$\left\{0, 1, \cdots, 2^{23-d}-1\right\} \subseteq \left\{-2^{23-d}, \cdots, 2^{23-d}\right\}.$$

In Dilithium [20], three sets of recommended parameters are proposed, aiming to achieving NIST security level 2, level 3, and level 5, respectively. For simplicity, the terms Dilithium-2, Dilithium-3, and Dilithium-5 are applied to refer to these three instances of Dilithium. And the recommended parameters are shown in Table 1.

2.6 NTT Technique in Dilithium

Recall that the Number Theoretic Transform (NTT) could be seen as a variant of fast fourier transform (FFT) that works over the finite field $\mathbb{F}_q$ instead of the field of complex numbers. In particular, when handling polynomial multiplication operation over certain special polynomial rings, the time efficiency of the NTT technique outperforms that of the naive polynomial multiplication algorithm.

To speed up the efficiency of algorithms in Dilithium, parameters are well-chosen so that NTT technique could be applied in the implementation of Dilithium. In particular, $n = 256 = 2^8$, and $q = 8380417$ is an odd prime such that $q \equiv 1 \pmod{2n}$. Thus, the multiplicative group $\mathbb{Z}_q^*$ is cyclic and has an element, say α, of order $2n = 512$, making

$$x^n + 1 \cong (x-\alpha) \cdot \left(x-\alpha^3\right) \cdots \left(x-\alpha^{511}\right).$$

Furthermore, each polynomial $\mathbf{a} \in \mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n+1\rangle$ can be uniquely represented in its CRT (Chinese Remainder Theorem) form as

$$\left[a(\alpha), a(\alpha^3), \cdots, a(\alpha^{2n-1})\right]^T \in \mathbb{F}_q^n.$$

It should be noted that the product of polynomials in CRT form can be evaluated in the component-wise manner. Therefore, with the aid of forward NTT operations and inverse NTT operations, we can compute the product of two polynomials in $\mathcal{R}_q$ in time $O(n \log n)$, instead of in time $O(n^2)$.

Algorithm 2 Key Generation Algorithm of Dilithium

Input: 1^λ
Output: $(\mathrm{pk} = (\rho, \overrightarrow{\mathbf{t}}_1), \mathrm{sk} = (\rho, K, tr, \overrightarrow{\mathbf{s}}, \overrightarrow{\mathbf{e}}, \overrightarrow{\mathbf{t}}_0))$

$(\rho, \rho', K) \in \{0,1\}^{256} \times \{0,1\}^{512} \times \{0,1\}^{256}$
$\mathbf{A} \in \mathcal{R}_q^{k\times\ell} := \mathsf{Sam}(\rho)$
$(\overrightarrow{\mathbf{s}}, \overrightarrow{\mathbf{e}}) \in S_\eta^\ell \times S_{\eta'}^k := \mathsf{Sam}(\rho')$
$\overrightarrow{\mathbf{t}} := \mathbf{A}\overrightarrow{\mathbf{s}} + \overrightarrow{\mathbf{e}}$
$(\overrightarrow{\mathbf{t}}_1, \overrightarrow{\mathbf{t}}_0) := \mathsf{Power2Round}_{q,d}\left(\overrightarrow{\mathbf{t}}\right)$
$\mathrm{tr} \in \{0,1\}^{256} := \mathrm{H}(\rho \| \mathbf{t}_1)$
return $(\mathrm{pk} = (\rho, \overrightarrow{\mathbf{t}}_1), \mathrm{sk} = (\rho, K, tr, \overrightarrow{\mathbf{s}}, \overrightarrow{\mathbf{e}}, \overrightarrow{\mathbf{t}}_0))$

Algorithm 3 The Sign Algorithm of Dilithium

Input: $\mathrm{sk} = (\rho, K, tr, \overrightarrow{\mathbf{s}}, \overrightarrow{\mathbf{e}}, \overrightarrow{\mathbf{t}}_0)$
Output: $\sigma = (\overrightarrow{\mathbf{z}}, \mathbf{c}, \overrightarrow{\mathbf{h}})$

$\mathbf{A} \in \mathcal{R}_q^{k\times\ell} := \mathsf{Sam}(\rho)$
$\overrightarrow{\mathbf{t}}_1 := \mathsf{Power2Round}_{q,d}\left(\overrightarrow{\mathbf{t}}\right)$
$\overrightarrow{\mathbf{t}}_0 := \overrightarrow{\mathbf{t}} - \overrightarrow{\mathbf{t}}_1 \cdot 2^d$
$r \leftarrow \{0,1\}^{256}$
$\overrightarrow{\mathbf{y}} \in S_{\gamma_1 - 1}^\ell := \mathsf{Sam}(r)$
$\overrightarrow{\mathbf{w}} := \mathbf{A}\overrightarrow{\mathbf{y}}$
$\overrightarrow{\mathbf{w}}_1 := \mathsf{HighBits}_q(\overrightarrow{\mathbf{w}}, 2\gamma_2)$
$\mathbf{c} \leftarrow H(\rho, \overrightarrow{\mathbf{t}}_1, \overrightarrow{\mathbf{w}}_1, \mu)$
$\overrightarrow{\mathbf{z}} := \overrightarrow{\mathbf{y}} + \mathbf{c} \cdot \overrightarrow{\mathbf{s}}$
$(\overrightarrow{\mathbf{r}}_1, \overrightarrow{\mathbf{r}}_0) := \mathsf{Decompose}_q(\overrightarrow{\mathbf{w}} - \mathbf{c}\overrightarrow{\mathbf{e}}, 2\gamma_2)$
Restart if $\|\overrightarrow{\mathbf{z}}\|_\infty \geq \gamma_1 - \beta$ or $\|\overrightarrow{\mathbf{r}}_0\|_\infty \geq \gamma_2 - \beta$ or $\overrightarrow{\mathbf{r}}_1 \neq \overrightarrow{\mathbf{w}}_1$
$\overrightarrow{\mathbf{h}} := \mathsf{MakeHint}_q(-\mathbf{c}\overrightarrow{\mathbf{t}}_0, \overrightarrow{\mathbf{w}} - \mathbf{c}\overrightarrow{\mathbf{e}} + \mathbf{c}\overrightarrow{\mathbf{t}}_0)$
Restart if $\left\|\mathbf{c}\overrightarrow{\mathbf{t}}_0\right\|_\infty \geq \gamma_2$ or the number of 1's in $\overrightarrow{\mathbf{h}}$ is greater than ω
return $(\overrightarrow{\mathbf{z}}, \mathbf{c}, \overrightarrow{\mathbf{h}})$

Algorithm 4 Verify Algorithm of Dilithium

Input: $\mathrm{pk} = (\rho, \overrightarrow{\mathbf{t}}_1), \mu \in \{0,1\}^*, (\overrightarrow{\mathbf{z}}, \mathbf{c}, \overrightarrow{\mathbf{h}})$
Output: $b \in \{0,1\}$

$\mathbf{A} \in \mathcal{R}_q^{k\times\ell} := \mathsf{Sam}(\rho)$
$\overrightarrow{\mathbf{w}}_1' := \mathsf{UseHint}_q(\overrightarrow{\mathbf{h}}, \mathbf{A}\overrightarrow{\mathbf{z}} - \mathbf{c}\overrightarrow{\mathbf{t}}_1 \cdot 2^d)$
$\mathbf{c}' \leftarrow H(\rho, \overrightarrow{\mathbf{t}}_1, \overrightarrow{\mathbf{w}}_1', \mu)$
if $\mathbf{c} = \mathbf{c}'$ and $\|\overrightarrow{\mathbf{z}}\|_\infty < \gamma_1 - \beta$ and the number of 1's in $\overrightarrow{\mathbf{h}}$ is $\leq \omega$ **then**
    return 1
else
    return 0
end if

3 A FAST MULTIPLICATION ALGORITHM FOR SMALL POLYNOMIALS

Recall that in the Sign algorithm and the Verify algorithm of Dilithium, we need to compute $\mathbf{c}\cdot\overrightarrow{\mathbf{s}} \in \mathcal{R}_q^{\ell}$ and $\mathbf{c}\cdot\overrightarrow{\mathbf{e}}, \mathbf{c}\cdot\overrightarrow{\mathbf{t}}_0, \mathbf{c}\cdot\overrightarrow{\mathbf{t}}_1 \in \mathcal{R}_q^{k}$. Although their coefficients are significantly "small" compared with q (for instance, $\|\mathbf{c}\|_\infty = 1, \|\overrightarrow{\mathbf{s}}\|_\infty = \|\overrightarrow{\mathbf{e}}\|_\infty = 2 \ll q = 8380417$), NTT technique is still applied with respect to the foregoing evaluations. A natural question arises as to whether we can compute these products in a faster way, by fully utilizing the facts that their magnitudes are much small compared with q. In this section, we give an affirmative answer to this interesting question. To be precise, this work is devoted to proposing a parallel algorithm that can handle the multiplications of $\mathbf{c} \in B_\tau$ with "small" polynomials in $\mathcal{R}_q$, and run fasters than the NTT technique does as is done in the implementation of Dilithium.

3.1 Index-Based Multiplication Algorithms

In this subsection, we shall propose two index-based polynomial multiplication algorithms, which are applicable for computing the product of $\mathbf{c} \in B_\tau$ and an arbitrary polynomial $\mathbf{a} \in \mathcal{R}_q$. These serve as the foundation of, as well as warm-up for, the design and analysis in Section 3.

We shall analyze properties of polynomial multiplication involving $\mathbf{c} \in B_\tau$ first. For simplicity, let $\mathbf{c} = \sum_{i=0}^{n-1} c_i \cdot x^i \in B_\tau, \mathbf{a} = \sum_{i=0}^{n-1} a_i \cdot x^i \in \mathcal{R}_q$, and let $\mathbf{u} = \mathbf{c}\cdot\mathbf{a} \in \mathcal{R}_q$. Therefore, if $\mathbf{u} = \sum_{i=0}^{n-1} u_i \cdot x^i$, then for every $0 \le i \le n-2$, we have

$$\begin{aligned} u_i &= \sum_{j=0}^{i} c_j \cdot a_{i-j} - \sum_{j=i+1}^{n-1} c_j \cdot a_{n+i-j} \\ &= \sum_{j=0}^{i} c_j \cdot a_{i-j} + \sum_{j=i+1}^{n-1} c_j \cdot (-a_{n+i-j}). \end{aligned}$$

and for $i = n-1$, we have

$$u_i = \sum_{j=0}^{n-1} c_j \cdot a_{i-j}.$$

The foregoing analysis, together with the fact that every $c_j \in \{-1, 0, 1\}$, implies the correctness of the following index-based

	Dilithium-2	Dilithium-3	Dilithium-5
q	8380417	8380417	8380417
n	256	256	256
d	13	13	13
(k,ℓ)	(4,4)	(6,5)	(8,7)
η	2	4	2
β	78	96	120
τ	39	49	60
ω	80	55	75
γ_1	2^{17}	2^{19}	2^{19}
γ_2	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$

Table 1: Recommended Parameters of Dilithium

Algorithm 5 An index-based multiplication algorithm for computing **ca**

Input: $\mathbf{c} = \sum_{i=0}^{n-1} c_i \cdot x^i \in B_\tau,\ \mathbf{a} = \sum_{i=0}^{n-1} a_i \cdot x^i \in \mathcal{R}_q$
Output: $\mathbf{u} = \mathbf{c}\cdot\mathbf{a} \in \mathcal{R}_q$

```
for i ∈ {0, 1, ··· , 2n − 2} do
    w_i := 0
end for
for i ∈ {0, 1, ··· , n − 1} do
    if c_i = 1 then
        for j ∈ {0, 1, ··· , n − 1} do
            w_{i+j} := w_{i+j} + a_j
        end for
    end if
    if c_i = −1 then
        for j ∈ {0, 1, ··· , n − 1} do
            w_{i+j} := w_{i+j} − a_j
        end for
    end if
end for
for i ∈ {0, 1, ··· , n − 1} do
    u_i := w_i − w_{i+n} (mod q)
end for
u := Σ_{i=0}^{n−1} u_i · x^i                ▷ u ∈ R_q
return u
```

polynomial multiplication algorithm, which is applicable for the computation of $\mathbf{c}\cdot\overrightarrow{\mathbf{s}}, \mathbf{c}\cdot\overrightarrow{\mathbf{e}}, \mathbf{c}\cdot\overrightarrow{\mathbf{t}}_0$ and $\mathbf{c}\cdot\overrightarrow{\mathbf{t}}_1$.

Conversely, we can implement the foregoing index-based polynomial multiplication algorithm in a slightly different manner as follows. First, define the sequence

$$v_{1-n}, v_{2-n}, \cdots, v_{-1}, v_0, v_1, \cdots, v_{n-2}, v_{n-1},$$

where

$$v_i = \begin{cases} a_i, & \text{if } 0 \le i \le n-1 \\ -a_{n+i}, & \text{if } 1-n \le i \le -1 \end{cases}.$$

Then, for every $0 \le i \le n-2$, we have

$$\begin{aligned} u_i &= \sum_{j=0}^{i} c_j \cdot v_{i-j} + \sum_{j=i+1}^{n-1} c_j \cdot (-v_{n+i-j}) \\ &= \sum_{j=0}^{i} c_j \cdot v_{i-j} + \sum_{j=i+1}^{n-1} c_j \cdot v_{i-j} \\ &= \sum_{j=0}^{n-1} c_j \cdot v_{i-j}. \end{aligned}$$

and for $i = n-1$,

$$u_i = \sum_{j=0}^{n-1} c_j \cdot v_{i-j}.$$

So, for every $0 \le i \le n-1$, we have

$$u_i = \sum_{j=0}^{n-1} c_j \cdot v_{i-j}.$$

Algorithm 6 An alternative index-based polynomial multiplication algorithm for computing **ca**

Input: $\mathbf{c} = \sum_{i=0}^{n-1} c_i \cdot x^i \in B_\tau$, $\mathbf{a} = \sum_{i=0}^{n-1} a_i \cdot x^i \in \mathcal{R}_q$
Output: $\mathbf{u} = \mathbf{c} \cdot \mathbf{a} \in \mathcal{R}_q$
1: **for** $i \in \{0, 1, \cdots, n-1\}$ **do**
2: $\quad w_i := 0$
3: $\quad v_i := a_i$
4: $\quad v_{i-n} := -a_i$
5: **end for**
6: **for** $i \in \{0, 1, \cdots, n-1\}$ **do**
7: $\quad$ **if** $c_i = 1$ **then**
8: $\quad\quad$ **for** $j \in \{0, 1, \cdots, n-1\}$ **do**
9: $\quad\quad\quad w_j := w_j + v_{j-i}$
10: $\quad\quad$ **end for**
11: $\quad$ **end if**
12: $\quad$ **if** $c_i = -1$ **then**
13: $\quad\quad$ **for** $j \in \{0, 1, \cdots, n-1\}$ **do**
14: $\quad\quad\quad w_j := w_j - v_{j-i}$
15: $\quad\quad$ **end for**
16: $\quad$ **end if**
17: **end for**
18: **for** $i \in \{0, 1, \cdots, n-1\}$ **do**
19: $\quad u_i := w_i \pmod q$ $\triangleright\ u_i \in \mathbb{F}_q$
20: **end for**
21: $\mathbf{u} := \sum_{i=0}^{n-1} u_i \cdot x^i$ $\triangleright\ \mathbf{u} \in \mathcal{R}_q$
22: **return u**

In other words, each $u_i \in \mathbb{F}_q$ could be seen as the inner product of the following two vectors

$$[c_0, c_1, \cdots, c_{n-1}]^T, [v_i, v_{i-1}, \cdots, v_{i-n+1}]^T \in \mathbb{F}_q^n$$

Furthermore, given that $\mathbf{c} \in B_\tau$ and hence every $c_r \in \{-1, 0, 1\}$ by assumption, the last equation can be rewritten as:

$$u_i = \sum_{c_j=1} v_{i-j} + \sum_{c_j=-1} (-v_{i-j}), \quad \forall 0 \le i \le n-1.$$

With this in mind, it is easy to derive an *alternative* index-based polynomial multiplication algorithm, which is applicable for the computation of $\mathbf{c} \cdot \overrightarrow{\mathbf{s}}, \mathbf{c} \cdot \overrightarrow{\mathbf{e}}\ \mathbf{c} \cdot \overrightarrow{\mathbf{t}}_0$, and $\mathbf{c} \cdot \overrightarrow{\mathbf{t}}_1$ as well.

Some remarks are in order. Note that when implementing Algorithm 6 in practice, we can pre-compute the sequence (v_i). This makes Algorithm 6 fit the design of Dilithium very well, as the following analysis shows. Take the product $\mathbf{c} \cdot \overrightarrow{\mathbf{s}}$ in the signing algorithm of Dilithium as an example. Recall that in the Sign algorithm of Dilithium, repetitions are necessary to generate a valid signature that does not leak the information regarding the secret key. Hence, in *each* call to the signing algorithm, we need to compute $\mathbf{c}' \cdot \overrightarrow{\mathbf{s}}, \mathbf{c}'' \cdot \overrightarrow{\mathbf{s}}, \cdots$, and the pre-computation of $\overrightarrow{\mathbf{s}}$ enables us to carry out the aforementioned multiplications in a faster manner than expected.

3.2 Make the Algorithm "Nonnegative" By Translations

In this subsection, we shall derive a "nonnegative" variant of Algorithm 6, which can be applied for computing the product of $\mathbf{c} \in B_\tau$ and a polynomial $\mathbf{a} \in \mathcal{R}_q$, *provided that* $\|\boldsymbol{a}\|_\infty = U \ll q$.

Recall that in Section 3.1, we have proposed two index-based polynomial multiplication algorithms involving $\mathbf{c} \in B_\tau$, *i.e.*, Algorithms 5 and 6. In particular, the pre-computation feature of Algorithm 6 fits the Sign algorithm of Dilithium very well, as the remark shows.

Nevertheless, careful analysis shows that there are still two issues in Algorithm 6:

- First, by definition, the value of v_i may be negative during computation in Algorithm 6;
- Moreover, the *key operations* in Algorithm 6 are the addition operation (Line 9) and the substraction operation (Line 14). As a result, the intermediate value w_i may be negative as well.

As we shall see in Section 3.3, these *possibly* negative intermediate values make us hard to derive a parallel and fast version of Algorithm 6.

Recall that the components of $\overrightarrow{\mathbf{s}}, \overrightarrow{\mathbf{e}}, \overrightarrow{\mathbf{t}}_0, \overrightarrow{\mathbf{t}}_1$ are "small" polynomials, in the sense that for the coefficients of each component, their magnitudes are much smaller than q. To be precise,

- Each coefficient in the components of $\mathbf{s}_1, \mathbf{s}_2$ belongs to $\{-\eta, \cdots, \eta\}$; and
- Each coefficient in the components of $\mathbf{t}_0$ belongs to
$$\left\{1 - 2^{d-1}, \cdots, 2^{d-1}\right\},$$
where $d = 13$; and
- Each coefficient in the components of $\mathbf{t}_1$ belongs to
$$\left\{0, \cdots, 2^{23-d} - 1\right\}.$$

Intuitively, we can first *translate* those coefficients into a nonnegative region by translation, and then manage to *cancel out* the effect of the translations in the end. The following analysis confirms the correctness of this intuition. *It should be stressed that when* $\mathbf{c} \in B_\tau$ *is identified with an n-dimensional column vector in* $\mathbb{F}_q^n$ *in the natural manner, the fact that the hamming weight of* $\mathbf{c}$ *is always the constant* τ *plays an essential role for the final cancellation operation.*

Still we assume $\mathbf{c} = \sum c_i \cdot x^i \in B_\tau$, and $\mathbf{a} = \sum a_i \cdot x^i \in \mathcal{R}_q$; moreover, we assume $U \overset{\text{def}}{=} \|\mathbf{a}\|_\infty \ll q$, and hence,

$$a_i \in \{-U, 1-U, \cdots, 0, \cdots, U-1, U\}, \quad \forall 0 \le i \le n-1.$$

To carry out the translation, define the sequence

$$(v_{n-1}, v_{n-2}, \cdots, v_1, v_0, v_{-1}, \cdots, v_{2-n}, v_{1-n})$$

as follows:

$$v_i = \begin{cases} U + a_i, & \text{if } 0 \le i \le n-1 \\ U - a_{n+i}, & \text{if } 1-n \le i \le -1 \end{cases}.$$

In other word, every v_i is larger than its "expected" value by U, and hence $v_i \in \{0, 1, \cdots, 2U\}$. Furthermore, in each key operation, we can translate the intermediate value w_j by U deliberately. Given

Algorithm 7 An index-based polynomial multiplication algorithm with translations

Input: $\mathbf{c} = \sum_{i=0}^{n-1} c_i \cdot x^i \in B_\tau$, $\mathbf{a} = \sum_{i=0}^{n-1} a_i \cdot x^i \in \mathcal{R}_q$
Output: $\mathbf{u} = \mathbf{c} \cdot \mathbf{a} \in \mathcal{R}_q$

1: **for** $i \in \{0, 1, \cdots, n-1\}$ **do**
2:     $w_i := 0$
3:     $v_i := U + a_i$
4:     $v_{i-n} := U - a_i$
5: **end for**
6: **for** $i \in \{0, 1, \cdots, n-1\}$ **do**
7:     **if** $c_i = 1$ **then**
8:         **for** $j \in \{0, 1, \cdots, n-1\}$ **do**
9:             $w_j := w_j + v_{j-i}$
10:         **end for**
11:     **end if**
12:     **if** $c_i = -1$ **then**
13:         **for** $j \in \{0, 1, \cdots, n-1\}$ **do**
14:             $w_j := w_j + (2U - v_{j-i})$
15:         **end for**
16:     **end if**
17: **end for**
18: **for** $i \in \{0, 1, \cdots, n-1\}$ **do**
19:     $u_i := w_i - \tau U \pmod{q}$ ▷ $u_i \in \mathbb{F}_q$
20: **end for**
21: $\mathbf{u} := \sum_{i=0}^{n-1} u_i \cdot x^i$ ▷ $\mathbf{u} \in \mathcal{R}_q$
22: **return u**

Algorithm 8 An index-based polynomial multiplication algorithm with translations (another version of Algorithm 7)

Input: $\mathbf{c} = \sum_{i=0}^{n-1} c_i \cdot x^i \in B_\tau$, $\mathbf{a} = \sum_{i=0}^{n-1} a_i \cdot x^i \in \mathcal{R}_q$
Output: $\mathbf{u} = \mathbf{c} \cdot \mathbf{a} \in \mathcal{R}_q$

for $i \in \{0, 1, \cdots, n-1\}$ **do**
    $w_i := 0$
    $\overline{v}_{i-n} = v_i := U + a_i$
    $\overline{v}_i = v_{i-n} := U - a_i$
end for
for $i \in \{0, 1, \cdots, n-1\}$ **do**
    if $c_i = 1$ **then**
        for $j \in \{0, 1, \cdots, n-1\}$ **do**
            $w_j := w_j + v_{j-i}$
        end for
    end if
    if $c_i = -1$ **then**
        for $j \in \{0, 1, \cdots, n-1\}$ **do**
            $w_j := w_j + \overline{v}_{j-i}$
        end for
    end if
end for
for $i \in \{0, 1, \cdots, n-1\}$ **do**
    $u_i := w_i - \tau U \pmod{q}$ ▷ $u_i \in \mathbb{F}_q$
end for
$\mathbf{u} := \sum_{i=0}^{n-1} u_i \cdot x^i$ ▷ $\mathbf{u} \in \mathcal{R}_q$
return u

that $\mathbf{c} \in B_\tau$ is of Hamming weight τ when identified with an n-dimensional vector in $\mathbb{F}_q^n$, the value w_i would be larger than its "expected" value by τU exactly in the end, which enables us to cancel out the effect of these foregoing translations easily via the substraction by the constant τU.

We can summarize the foregoing analysis into the following Algorithm 7.

Some remarks regarding Algorithm 7 are in order.

- First, note that in Line 14 (of Algorithm 7), we always have $2U - v_{j-i} \geq 0$, and hence the key operation in Line 14 is an addition operation (instead of a substraction operation), which guarantees that $w_j \geq 0$ always holds.
- Moreover, note that in Algorithm 7, each v_i belongs to the set $\{0, 1, \cdots, 2U\}$, and hence the intermediate value w_i always belongs to the set $\{0, 1, \cdots, 2\tau U\}$. In other words, during computations in Algorithm 7, the nonnegative w_i is upper-bounded by $2\tau U$, which is much smaller than q. As we shall see in Section 3.3, this "small" upper-bound $2\tau U$ for w_j is *essential* for us to build the *parallel* variant of Algorithm 7.

3.3 The Parallel Multiplication Algorithm for Small Polynomials

In this subsection, we consider the problem of computing the product $\mathbf{c} \cdot \overrightarrow{\mathbf{a}}$, where

- $\mathbf{c} \in B_\tau$;
- $\overrightarrow{\mathbf{a}} = \left[\mathbf{a}^{(0)}, \cdots, \mathbf{a}^{(r-1)}\right]^T \in \mathcal{R}_q^r$;
- There exists a common magnitude upper-bound $U > 0$ such that
$$\left\|\mathbf{a}^{(j)}\right\|_\infty \leq U, \quad \forall j \in \{0, 1, \cdots, r-1\}.$$

Here, $\overrightarrow{\mathbf{a}}$ could be seen as a "model" for the $\overrightarrow{\mathbf{s}}, \overrightarrow{\mathbf{e}}, \overrightarrow{\mathbf{t}}_0, \overrightarrow{\mathbf{t}}_1$ in Dilithium.

First, it is trivial to see that we can compute every $\mathbf{c} \cdot \mathbf{a}^{(j)}$ by applying either Algorithms 5 or Algorithm 6 or Algorithm 7 in the *sequential* manner. However, as we shall see later, we can actually do much better.

Recall that in Section 3.2, we have proposed an improved index-based polynomial multiplication algorithm, *i.e.*, Algorithm 7, which is "nonnegative" in the sense that except for the final returned values, all the involving intermediate values w_i and v_i in Algorithm 7 are always nonnegative. Moreover, careful analysis shows that, in Algorithm 7, the intermediate values w_i and v_i are not only nonnegative, but also "small" in the sense that the maximal upper-bound is

$$\max(2U, 2\tau U) = 2\tau U.$$

For instance, in Dilithium-2 [20], for $\overrightarrow{\mathbf{s}} \in \mathcal{R}_q^\ell$ and $\overrightarrow{\mathbf{e}} \in \mathcal{R}_q^k$ we have $\tau = 39$ and $U = 2$, making

$$2\tau U < 2^8 \ll q = 8380417.$$

The observation that intermediate values in Algorithm 7 are always small nonnegative integers motivates us to design its parallel version, which is best captured by the following simplified analysis. Given four *nonnegative* integers $0 \le a_0, b_0, a_1, b_1 \le \alpha$, we are asked to find $a = a_0 + a_1$ and $b = b_0 + b_1$. In addition to the direct method, we can obtain the values of a, b in the following *indirect* manner: letting $v_0 = a_0 \cdot M + b_0$ and $v_1 = a_1 \cdot M + b_1$. When $M > 2\alpha$, it is routine to see

$$a = \lfloor (v_0 + v_1)/M \rfloor, \quad b = (v_0 + v_1) \bmod M.$$

Note the assumption that all the given values are nonnegative makes it possible for us to recover the intact a and b from the sum $v_0 + v_1$.

Likewise, given $\mathbf{c} \in B_\tau$ and the components

$$\mathbf{s}^{(0)} = \sum_{i=0}^{n-1} s_i^{(0)} \cdot x^i, \quad \mathbf{s}^{(1)} = \sum_{i=0}^{n-1} s_i^{(1)} \cdot x^i \in \mathcal{R}_q$$

of $\overrightarrow{\mathbf{s}}$ in Dilithium-3, if we define

$$v_i = \begin{cases} \left(U + s_i^{(0)}\right) \cdot 2^8 + \left(U + s_i^{(1)}\right), & \text{if } 0 \le i \le n-1 \\ \left(U - s_{n+i}^{(0)}\right) \cdot 2^8 + \left(U - s_{n+i}^{(1)}\right), & \text{if } 1-n \le i \le -1 \end{cases}$$

then by making necessary adaption to Algorithm 7, we can extract the pair $\left(u_i^{(0)}, u_i^{(1)}\right)$ intact from $w_i \in \left\{0, 1, \cdots, 2^{16}-1\right\}$ in the end.

In sum, we can derive the following Algorithm 9, which could be seen as the *parallel* version of Algorithm 7. It is not hard to verify the correctness of Algorithm 9, provided that the constants U and M satisfy $M > 2\tau U$.

Some remarks about the implementation of Algorithm 9 are in order.

First, recall that to make Algorithm 9 work properly, the parameters U and M should satisfy $M > 2\tau U$. In practice M is usually to be a power-of-two, say $M = 2^{\lceil \log_2 (1+2\tau U) \rceil}$, which would make it easy for us to recover $u_i^{(j)}$ from w_i via bit-shift operations.

Moreover, as the parallel version of Algorithm 7, a single call to Algorithm 9 recovers *several*, instead of one, products of $\mathbf{c}$ and "small" polynomials. Hence, when solving the problem proposed at the beginning of this subsection, it is routine to see that the *average* performance of Algorithm 9 outperforms that of Algorithm 7 in practice.

Last but not the least, Algorithm 9 can perform *even* better in practice. Note that in the key operations of Algorithm 9, only addition and substraction operations are involved. Nowadays for most of the PCs and servers, their processors are of 64-bit; when we run programs on those processors, the addition/substraction of two 64-bit integers usually takes 1 CPU cycle, which is the same as the addition / substraction of two 32-bit integers. In particular, this holds true for the *Intel Core i7-6600U processor*, on which Dilithium team tested the benchmark programs submitted to NIST's third-round PQC selection program [20]. Therefore, if $r \cdot \log_2 M \le 64$, when we deploy Algorithm 9 in PCs or servers with 64-bit processor(s), a single call to Algorithm 9 takes *almost* the same CPU cycles as a single call to Algorithm 7 does in general.

In sum, when we use Algorithm 9 to compute the products $\mathbf{c} \cdot \overrightarrow{\mathbf{s}}, \mathbf{c} \cdot \overrightarrow{\mathbf{e}}$ and $\mathbf{c} \cdot \overrightarrow{\mathbf{t}}_0$ in the signing procedure of Dilithium, $\mathbf{c} \cdot \overrightarrow{\mathbf{t}}_1$ in the verifying procedure of Dilithium, intuitively we can speed up the signing procedure when running it on 64-bit processors, which is confirmed by our experimental results, as we shall see in Section 5.

Algorithm 9 A parallel index-based polynomial multiplication algorithm with translations

Input: $\left(\mathbf{c}, \overrightarrow{\mathbf{a}}\right)$, where
- $\mathbf{c} = \sum_{i=0}^{n-1} c_i \cdot x^i \in B_\tau$;
- $\overrightarrow{\mathbf{a}} = \left\{\mathbf{a}^{(j)}\right\} \in \mathcal{R}_q^r$;
- Every $\mathbf{a}^{(j)} = \sum_{i=0}^{n-1} a_i^{(j)} \cdot x^i \in \mathcal{R}_q$;
- Every $a_i^{(j)} \in \{-U, \cdots, U\}$

Output: $\overrightarrow{\mathbf{u}} = \left[\mathbf{u}^{(0)}, \cdots, \mathbf{u}^{(r-1)}\right]^T \in \mathcal{R}_q^r$, where
- $\mathbf{u}^{(j)} = \mathbf{c} \cdot \mathbf{a}^{(j)} \in \mathcal{R}_q$;

for $i \in \{0, 1, \cdots, n-1\}$ **do**
    $w_i := 0$
    $v_i := 0$
    $v_{i-n} := 0$
    for $j = 0$ **to** $r-1$ **do**
        $v_i := v_i \cdot M + \left(U + a_i^{(j)}\right)$
        $v_{i-n} := v_{i-n} \cdot M + \left(U - a_i^{(j)}\right)$
    end for
end for
$\gamma := 2U \cdot \frac{M^r - 1}{M-1}$ ▷ $\gamma \in \mathbb{Z}^{>0}$
for $i \in \{0, 1, \cdots, n-1\}$ **do**
    if $c_i = 1$ **then**
        for $j \in \{0, 1, \cdots, n-1\}$ **do**
            $w_j := w_j + v_{j-i}$
        end for
    end if
    if $c_i = -1$ **then**
        for $j \in \{0, 1, \cdots, n-1\}$ **do**
            $w_j := w_j + (\gamma - v_{j-i})$ ▷ $\gamma = 2U \cdot \frac{M^r - 1}{M-1}$
        end for
    end if
end for
for $i \in \{0, 1, \cdots, n-1\}$ **do**
    $t := w_i$
    for $j = 0$ **to** $r-1$ **do**
        $u_i^{(r-1-j)} := (t \bmod M) - \tau U \pmod q$
        $t := \lfloor t/M \rfloor$
    end for
end for
for $j \in \{0, 1, \cdots, r-1\}$ **do**
    $\mathbf{u}^{(j)} := \sum_{i=0}^{n-1} u_i^{(j)} \cdot x^i$
end for
$\overrightarrow{\mathbf{u}} := \left[\mathbf{u}^{(0)}, \cdots, \mathbf{u}^{(r-1)}\right]^T$
return $\overrightarrow{\mathbf{u}}$

4 IMPLEMENTATION

Nowadays most PCs or embedded devices support 64-bit processors, so we concentrate our improved implementation on 64-bit computer systems. We will use such as uint64_t to store values in C implementation, so as to fully utilize the parallel advantage of Algorithm 9. But our methods work also on 32-bit processors.

Moreover, we create an optimized version of C implementation by replacing the polynomial multiplications $\mathbf{c}\cdot\overrightarrow{\mathbf{s}}, \mathbf{c}\cdot\overrightarrow{\mathbf{e}}, \mathbf{c}\cdot\overrightarrow{\mathbf{t}}_0$ and $\mathbf{c}\cdot\overrightarrow{\mathbf{t}}_1$ in Dilithium with various instances of Algorithm 9. And indeed, it can improve the efficiency of the Sign and Verify of Dilithium, with benchmark details given in Section 5.

Finally, for adapting to low-power devices, we select the Cortex-A72 (ARMv8) platform, and use Neon SIMD instructions to improve C implementation. The algorithm details are shown in Algorithm 10 and Appendix B. The Neon register bank consists of 32 64-bit registers, and the Neon unit can view the same register bank as 16 128-bit quadword registers. With its SIMD characteristic, we can operate more coefficients in parallel than C reference implementation, and thus obtain a further speed-up over the reference implementation.

5 EXPERIMENTAL RESULTS

Algorithm 10 Neon implementation of Algorithm 9

Input: $(\mathbf{c}, \overrightarrow{\mathbf{a}})$, where $\overrightarrow{\mathbf{a}} = [a^{(0)}, \cdots, a^{(r-1)}]^T \in \mathcal{R}_q^r$, every $a^{(j)} = \sum_{i=0}^{n-1} a_i^{(j)} \cdot x^i \in \mathcal{R}_q$, and $c = \sum_{i=0}^{n-1} c_i \cdot x^i \in B_\tau$

Output: $\overrightarrow{\mathbf{u}} = \mathbf{c}\cdot\overrightarrow{\mathbf{a}} = [u^{(0)}, \cdots, u^{(r-1)}]^T \in \mathcal{R}_q^r$, where $u^{(j)} = c\cdot a^{(j)} = \sum_{i=0}^{n-1} u_i^{(j)} \cdot x^i \in \mathcal{R}_q$

prepare_table_offset(s, c)
$\gamma := 2U \cdot \frac{M^r-1}{M-1}$
for $i \in \{0, 1, \cdots, n-1\}$ **do**
  $w_i := 0$
  $v_i := 0$
  $v_{i+n} := 0$
  $v_{i+2n} := 0$
end for
for $i \in \{0, 8, \cdots, n-8\}$ **do**
  for $j \in (0, 1, \cdots, r-1)$ **do**
    neon_cal_table$(v_{i+n}, a_i^{(j)}, \log_2(M), U)$
  end for
  neon_st_table(v_{i+n}, γ)
end for
for $i \in \{0, 8, \cdots, n-8\}$ **do**
  if $c_i = 0$ **then**
    continue
  end if
  neon_array_acc(w, v_{s_i})
end for
for $i \in \{0, 8, \cdots, n-8\}$ **do**
  $t := w_i$
  for $j \in (0, 1, \cdots, r-1)$ **do**
    neon_evaluate_[cs, ct0, ct1]$(u_i^{(r-1-j)}, t, M-1, \tau U)$
  end for
end for
return $\mathbf{u} = [u^{(0)}, \cdots, u^{(r-1)}]^T$

Recall that in Section 3.3, we have proposed a parallel algorithm, *i.e.*, Algorithm 9, that is suitable to compute the scalar product of $\mathbf{c} \in B_\tau$ and "small" column vectors $\overrightarrow{\mathbf{a}} \in \mathcal{R}_q^r$. Clearly, this parallel algorithm handles the products $\mathbf{c}\cdot\overrightarrow{\mathbf{s}}, \mathbf{c}\cdot\overrightarrow{\mathbf{e}}, \mathbf{c}\cdot\overrightarrow{\mathbf{t}}_0, \mathbf{c}\cdot\overrightarrow{\mathbf{t}}_1$ very well. Intuitively, when computing these products, it runs faster than the classic NTT technique does via parallelization, and hence can speed up the Sign and Verify procedures of Dilithium. We present both the C reference implementation and ARM Neon optimized implementation of Dilithium [20], by implementing the polynomial multiplications $\mathbf{c}\cdot\overrightarrow{\mathbf{s}}, \mathbf{c}\cdot\overrightarrow{\mathbf{e}}, \mathbf{c}\cdot\overrightarrow{\mathbf{t}}_0, \mathbf{c}\cdot\overrightarrow{\mathbf{t}}_1$ with the various instances of Algorithm 9.

In this section, we provide several experiments and benchmarks to confirm this intuition. For C implementation, benchmark tests are run on an Intel(R) Core(TM) i7-10510U CPU at 2.3GHz (16 GB memory) with Turbo Boost and Hyperthreading disabled. The operating system is Ubuntu 20.04 LTS with Linux Kernel 4.4.0 and the gcc version is 9.4.0. The compiler flag is listed as follows: -Wall -march=native -mtune=native -O3 -fomit-frame-pointer -Wno-unknown-pragma. For the Arm neon implementation, our embedded platform is Raspberry Pi 4B (RPi 4 Model B) with ARMv8-A

instruction sets, Cortex-A72 (1.8 GHz) CPU and 4GB RAM, and it supports ARMv8-A Neon SIMD instructions. A Neon register has 128-bit, so we can operate two 64-bit elements or four 32-bit elements at the same time. We did not use O3 optimization in Arm Neon benchmarktests, and our compiling options are '-Wall -Wextra -Wpedantic -Wmissing-prototypes -Wredundant-decls -Wshadow -Wvla -Wpointer-arith -march=native -mtune=native -g. We run the corresponding Sign and Verify algorithms for 10000 times and calculate the average CPU cycles. Because we didn't change Keygen algorithms, we only compare Sign and Verify CPU cycles here.

First and foremost, we list (in Tables 2,3 and 4) the parameterized parameters when implementing Algorithm 9 on Dilithium-2, Dilithium-3, and Dilithium-5, respectively.

	τ	U	$2\tau U$	M	r
$\mathbf{c}\cdot\overrightarrow{\mathbf{s}}, \mathbf{c}\cdot\overrightarrow{\mathbf{e}}$ in Dilithium-2	39	2	156	2^8	$8 = 4 + 4$
$\mathbf{c}\cdot\overrightarrow{\mathbf{t}_0}$ in Dilithium-2	39	2^{12}	319488	2^{19}	$4 = 2 + 2$
$\mathbf{c}\cdot\overrightarrow{\mathbf{t}_1}$ in Dilithium-2	39	2^{10}	79872	2^{17}	$4 = 2 + 2$

Table 2: Parallel Parameters for Dilithium-2

	τ	U	$2\tau U$	M	r
$\mathbf{c}\cdot\overrightarrow{\mathbf{s}}$ in Dilithium-3	49	4	392	2^9	5
$\mathbf{c}\cdot\overrightarrow{\mathbf{e}}$ in Dilithium-3	49	4	392	2^9	6
$\mathbf{c}\cdot\overrightarrow{\mathbf{t}_0}$ in Dilithium-3	49	2^{12}	401408	2^{19}	$6 = 3 + 3$
$\mathbf{c}\cdot\overrightarrow{\mathbf{t}_1}$ in Dilithium-3	49	2^{10}	100352	2^{17}	$6 = 3 + 3$

Table 3: Parallel Parameters for Dilithium-3

It should be *emphasized* that

- For every component of $\overrightarrow{\mathbf{s}}$, each coefficient belongs to the centrally symmetric set $\{-\eta, \cdots, \eta\}$. Thus, $U = \eta$. Similar considerations carries over to $\overrightarrow{\mathbf{e}}$.

- For every component of $\overrightarrow{\mathbf{t}}_0$, each coefficient belongs to the centrally symmetric set $\left\{-2^{d-1}+1,\cdots,2^{d-1}\right\}$, which is, however, not centrally symmetric. For ease of implementation, it is wise to set $U = 2^{d-1}$ and assume that each coefficient in $\overrightarrow{\mathbf{t}}_0$ belongs to $\left\{-2^{d-1},\cdots,2^{d-1}\right\}$.
- For every component of $\overrightarrow{\mathbf{t}}_1$, each coefficient belongs to the centrally symmetric set $\left\{0,1,\cdots,2^{23-d}-1\right\}$, which is obvious not centrally symmetric, either. For ease of implementation, it is wise to set $U = 2^{23-d}$ and assume that each coefficient in $\overrightarrow{\mathbf{t}}_1$ belongs to $\left\{-2^{23-d},\cdots,2^{23-d}\right\}$.

Experiments soon confirm that such relaxations on the handling of $\overrightarrow{\mathbf{t}}_0$ and $\overrightarrow{\mathbf{t}}_1$ simplify our considerations significantly.

In Table 5, the second column lists the mean CPU cycles of each algorithm under consideration in the *original* implementation of Dilithium [20]; whereas the last column lists the mean CPU cycles of each algorithm under consideration when we replace the NTT evaluation of $\mathbf{c}\cdot\overrightarrow{\mathbf{s}},\mathbf{c}\cdot\overrightarrow{\mathbf{e}},\mathbf{c}\cdot\overrightarrow{\mathbf{t}}_0,\mathbf{c}\cdot\overrightarrow{\mathbf{t}}_1$ with various instances of Algorithm 9 and keep all the other settings in the original Dilithium implementation unchanged. In sum, these comparisons demonstrate the power of Algorithm 9 (and its like) in a quantitative way.

It turns out that we can indeed improve the efficiency of the Sign algorithm and the Verify algorithm in Dilithium, as shown in Table 5. Specifically, our C implementation using parallel small polynomial multiplication is 18% faster in Sign and 19% in Verify respectively for Dilithium-2 (30% and 7% in Dilithium-3, 27% and 3% in Dilithium-5). As for the Arm Neon implementation, we achieved a performance improvement of about 64% in Sign and 50% in Verify for Dilithium-2(60% and 32% for Dilithium-3) compared to reference implementation as shown in Table 6. Compared with state-of-the-art Dilithium Arm Neon implementation [4], our speed of Sign is 13.4% faster in Dilithium-2 and 8.0% faster in Dilithium-3. More detailed CPU cycles data is shown in Table 7.

	τ	U	$2\tau U$	M	r
$\mathbf{c}\cdot\overrightarrow{\mathbf{s}}$ in Dilithium-5	60	2	240	2^9	7
$\mathbf{c}\cdot\overrightarrow{\mathbf{e}}$ in Dilithium-5	60	2	240	2^9	$8=4+4$
$\mathbf{c}\cdot\overrightarrow{\mathbf{t}_0}$ in Dilithium-5	60	2^{12}	491520	2^{19}	$8=3+3+2$
$\mathbf{c}\cdot\overrightarrow{\mathbf{t}_1}$ in Dilithium-5	60	2^{10}	122880	2^{17}	$8=3+3+2$

Table 4: Parallel Parameters for Dilithium-5

	Before	After
Sign in Dilithium-2	748500	628171
Verification in Dilithium-2	185022	166419
Sign in Dilithium-3	1281313	891613
Verification in Dilithium-3	291921	269118
Sign in Dilithium-5	1577046	1148236
Verification in Dilithium-5	474205	456961

Table 5: Reference Implementation Comparitive Results in cpucycles.

	Reference code	Our work
Sign in Dilithium-2	8223359	2934124
Verification in Dilithium-2	1941673	1231796
Sign in Dilithium-3	12166847	4927678
Verification in Dilithium-3	3063575	2073518

Table 6: Our neon implementation and reference Implementation Comparitive Results in cpucycles.

	[4]	Our work
Sign in Dilithium-2	3327206	2934124
Verification in Dilithium-2	1191080	1231796
Sign in Dilithium-3	5321618	4927678
Verification in Dilithium-3	2018945	2073518

Table 7: Arm neon Implementation Comparitive Results in cpucycles.

	Our Algorithm	NTT
$\mathbf{c}\cdot\overrightarrow{\mathbf{s}}\&\mathbf{c}\overrightarrow{\mathbf{e}}$ in Dilithium-3	8477	73924
$\mathbf{c}\cdot\overrightarrow{\mathbf{t}}_0$ in Dilithium-3	14132	88198
$\mathbf{c}\cdot\overrightarrow{\mathbf{t}}_1$ in Dilithium-3	11760	90992

Table 8: Polynoimal Multiplication Comparitive Results in cpucycles.

To better demonstrate the performance of our parallel small polynomial multiplication algorithm, we test separately the polynomial multiplication in Dilithium-3, such as $\mathbf{c}\cdot\overrightarrow{\mathbf{s}},\mathbf{c}\cdot\overrightarrow{\mathbf{e}},\mathbf{c}\cdot\overrightarrow{\mathbf{t}}_0$ and $\mathbf{c}\cdot\overrightarrow{\mathbf{t}}_1$. In Table 8, compared with NTT, our algorithm speeds up 88% for both $\mathbf{c}\cdot\overrightarrow{\mathbf{s}}$ and $\mathbf{c}\cdot\overrightarrow{\mathbf{e}}$. For $\mathbf{c}\cdot\overrightarrow{\mathbf{t}}_0$ our algorithm is 84% faster, and $\mathbf{c}\cdot\overrightarrow{\mathbf{t}}_1$ has an improvement of 87%. The test results show that our algorithm has better performance than NTT when calculating small polynomial multiplication. Not only for Dillithium, our algorithm can be applied to speed up other cryptographic schemes which contain small polynomial multiplication arithmetic.

Our proposed parallel small polynomial multiplication algorithm brings additional memory cost for storing intermidiate values and preparing parallel precomputation table. For resource-constrained device Arm Cortex A72, the extra memory cost is acceptable since it costs 49.152KB to store $\mathbf{c}\cdot\overrightarrow{\mathbf{s}}$, $\mathbf{c}\cdot\overrightarrow{\mathbf{e}}$, $\mathbf{c}\cdot\overrightarrow{\mathbf{t}}_0$ and $\mathbf{c}\cdot\overrightarrow{\mathbf{t}}_1$, and the core we use provides 4GB RAM.

6 RESISTANCE TO SIDE-CHANNEL ATTACKS

The implementation of the cryptographic primitive must be constant in time to prevent the leakage of secret information. In our implementation we introduce **if/else** branching structures. The presence of the branch structure may lead to differences in runtime and power consumption, which may result in side-channel attacks. The presence of the branch structure may lead to differences in runtime and power consumption, which may result in side-channel attacks. However, the seed $\tilde{c}$ used to generate c and the generation algorithm are public, which means that the difference in runtime and power consumption does not lead to leakage of private information.

Besides, our implementation is parallel, with the coefficients of multiple polynomials being packaged and operated together, so there is no leakage of individual data. Taking these factors into account, our implementation is resistant to side-channel attacks.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we exhibit a small polynomial multiplication parallel algorithm which can compute the products of "small" polynomials more quickly than general NTT. We complete the C reference implementation of Dilithium using our algorithm. Also, we improve the algorithm by Neon vector extension on the Cortex-A72 platform.

The evaluation results show that our algorithm outperforms NTT in Dilithium polynomial multiplication computation $\mathbf{c}\cdot\overrightarrow{\mathbf{s}}$, $\mathbf{c}\cdot\overrightarrow{\mathbf{e}}$, $\mathbf{c}\cdot\overrightarrow{\mathbf{t}}_0$ and $\mathbf{c}\cdot\overrightarrow{\mathbf{t}}_1$. We have a total performance improvement in both C reference implementation and Arm Cortex A72 implementation, achieving the new record of fast Dilithium implementation.

We believe that there is still something that can be improved for our Neon implementation, and may find better ways to optimize the algorithm. And we should do further study on ARMv8 architecture and instructions in the future. Besides Cortex-A, we are also going to transplant the parallel small polynomial multiplication into other embedded platforms, such as Cortex-M. Finally, we suggest the parallel small polynomial multiplication technique may have independent interest, and have more applications beyond implementation optimization of Dilithium.

ACKNOWLEDGMENTS

We would like to thank the people who provide helpful suggestions and discussions for this paper: Boyue Fang, Boru Gong, Zhichuang Liang and Zhongyi Ni. This paper is supported By The National Natural Science Foundation of China (No.U1536205, No.61472084), The National Key Research and Development Program of China (No.2017YFB0802000), Shanghai Innovation Action Project (No.16DZ1100200), Shanghai Science and Technology Development Funds (No.16JC1400801), Technical Standard Project of Shanghai Scientific and Technological Committee (No.21DZ2200500), Shandong Provincial Key Research and Development Program (No.2017CXG0701, No.2018CXGC0701).

REFERENCES

[1] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Daan Sprenkels. 2022. Faster Kyber and Dilithium on the Cortex-M4. In *Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13269)*, Giuseppe Ateniese and Daniele Venturi (Eds.). Springer, 853–871. https://doi.org/10.1007/978-3-031-09234-3_42
[2] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. 2020. Cortex-M4 optimizations for {R, M} LWE schemes. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020, 3 (2020), 336–357. https://doi.org/10.13154/tches.v2020.i3.336-357
[3] Roberto Avanzi, Joppe Bos, and Léo Ducas. 2020. CRYSTALS-Dilithium. In *Submission to the NIST Post-Quantum Cryptography Standardization Project.* https://pq-crystals.org/dilithium
[4] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. 2022. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022, 1 (2022), 221–244. https://doi.org/10.46586/tches.v2022.i1.221-244
[5] Luke Beckwith, Duc Tri Nguyen, and Kris Gaj. 2022. High-Performance Hardware Implementation of Lattice-Based Digital Signatures. *Cryptology ePrint Archive* (2022).
[6] Leon Botros, Matthias J Kannwischer, and Peter Schwabe. 2019. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In *International Conference on Cryptology in Africa*. Springer, 209–228.
[7] Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. 2000. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 392–407.
[8] Léo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. Crystals–dilithium: Digital signatures from module lattices. (2018).
[9] Alexander El-Kady, Apostolos P Fournaris, Thanasis Tsakoulis, Evangelos Haleplidis, and Vassilis Paliouras. 2021. High-Level Synthesis design approach for Number-Theoretic Transform Implementations. In *2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 1–6.
[10] Ruben Gonzalez, Andreas Hülsing, Matthias J Kannwischer, Juliane Krämer, Tanja Lange, Marc Stöttinger, Elisabeth Waitz, Thom Wiggers, and Bo-Yin Yang. 2021. Verifying post-quantum signatures in 8 kb of RAM. In *International Conference on Post-Quantum Cryptography*. Springer, 215–233.
[11] Denisa OC Greconici, Matthias J Kannwischer, and Daan Sprenkels. 2021. Compact dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 1–24.
[12] Naina Gupta, Arpan Jati, Anupam Chattopadhyay, and Gautam Jha. 2022. Lightweight Hardware Accelerator for Post-Quantum Digital Signature CRYSTALS-Dilithium. *Cryptology ePrint Archive* (2022).
[13] James Howe and Bas Westerbaan. 2022. Benchmarking and Analysing the NIST PQC Finalist Lattice-Based Signature Schemes on the ARM Cortex M7. *Cryptology ePrint Archive* (2022).
[14] Murat Burhan İlter, Neşe Koçak, Erkan Uslu, Oğuz Yayla, and Nergiz Yuca. 2021. On the Number of Arithmetic Operations in NTT-based Polynomial Multiplication in Kyber and Dilithium Cryptosystems. In *2021 14th International Conference on Security of Information and Networks (SIN)*, Vol. 1. IEEE, 1–7.
[15] Patrick Karl, Jonas Schupp, Tim Fritzmann, and Georg Sigl. 2022. Post-Quantum Signatures on RISC-V with Hardware Acceleration. *Cryptology ePrint Archive* (2022).
[16] Youngbeom Kim, Jingyo Song, Taek-Young Youn, and Seog Chung Seo. 2022. Crystals-Dilithium on ARMv8. *Security and Communication Networks* 2022 (2022).
[17] Georg Land, Pascal Sasdrich, and Tim Güneysu. 2021. A hard crystal-implementing dilithium on reconfigurable hardware. In *International Conference on Smart Card Research and Advanced Applications*. Springer, 210–230.
[18] Adeline Langlois and Damien Stehlé. 2015. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography* 75, 3 (2015), 565–599.
[19] Jihye Lee, Whijin Kim, Sohyeon Kim, and Ji-Hoon Kim. 2022. Post-Quantum Cryptography Coprocessor for RISC-V CPU Core. In *2022 International Conference on Electronics, Information, and Communication (ICEIC)*. IEEE, 1–2.
[20] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. 2017. Crystals-dilithium. *Submission to the NIST Post-Quantum Cryptography Standardization [NIS]* (2017).
[21] Vadim Lyubashevsky and Daniele Micciancio. 2006. Generalized compact knapsacks are collision resistant. In *International Colloquium on Automata, Languages, and Programming*. Springer, 144–155.
[22] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On ideal lattices and learning with errors over rings. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 1–23.
[23] Robert J McEliece. 1978. A public-key cryptosystem based on algebraic. *Coding Thv* 4244 (1978), 114–116.
[24] Daniele Micciancio and Oded Regev. 2009. Lattice-based cryptography. In *Post-quantum cryptography*. Springer, 147–191.
[25] Duc Tri Nguyen and Kris Gaj. 2021. Optimized software implementations of CRYSTALS-Kyber, NTRU, and Saber using NEON-based special instructions of ARMv8. In *Proceedings of the NIST 3rd PQC Standardization Conference (NIST PQC 2021)*.
[26] Chris Peikert and Alon Rosen. 2006. Efficient collision-resistant hashing from worst-case assumptions on cyclic lattices. In *Theory of Cryptography Conference*. Springer, 145–166.
[27] Cong Peng, Jianhua Chen, Sherali Zeadally, and Debiao He. 2019. Isogeny-based cryptography: A promising post-quantum technique. *IT Professional* 21, 6 (2019), 27–32.
[28] Luis J Dominguez Perez et al. 2021. Implementing CRYSTAL-Dilithium on FRDM-K64. In *2021 IEEE 12th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*. IEEE, 0178–0183.
[29] Prasanna Ravi, Romain Poussier, Shivam Bhasin, and Anupam Chattopadhyay. 2020. On Configurable SCA Countermeasures Against Single Trace Attacks for the NTT - A Performance Evaluation Study over Kyber and Dilithium on the ARM Cortex-M4. In *Security, Privacy, and Applied Cryptography Engineering - 10th International Conference, SPACE 2020, Kolkata, India, December 17-21, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12586)*, Lejla Batina, Stjepan

Picek, and Mainack Mondal (Eds.). Springer, 123–146. https://doi.org/10.1007/978-3-030-66626-2_7

[30] Sara Ricci, Lukas Malina, Petr Jedlicka, David Smékal, Jan Hajny, Peter Cibik, Petr Dzurenda, and Patrik Dobias. 2021. Implementing crystals-dilithium signature scheme on fpgas. In *The 16th International Conference on Availability, Reliability and Security*. 1–11.

[31] Pakize Sanal, Emrah Karagoz, Hwajeong Seo, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. 2021. Kyber on ARM64: Compact implementations of Kyber on 64-bit ARM Cortex-A processors. In *International Conference on Security and Privacy in Communication Systems*. Springer, 424–440.

[32] Peter W Shor. 1999. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review* 41, 2 (1999), 303–332.

[33] Deepraj Soni and Ramesh Karri. 2021. Efficient hardware implementation of pqc primitives and pqc algorithms using high-level synthesis. In *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 296–301.

[34] Cankun Zhao, Neng Zhang, Hanning Wang, Bohan Yang, Wenping Zhu, Zhengdong Li, Min Zhu, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2022. A Compact and High-Performance Hardware Architecture for CRYSTALS-Dilithium. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022, 1 (2022), 270–295.

A SECURITY MODEL FOR DIGITAL SIGNATURE SCHEME

The (strong) security for a signature scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ is defined a security game between the challenger and an adversary A, which consists of the following three *consecutive* phases:

- Setup. Given the security parameter 1^λ, the challenger runs $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$. The public key pk is given to adversary A, whereas the secret key sk is kept in private.
- Challenge. A can make signature queries on its will. Suppose A makes at most q_s signature queries. Each signature query consists of the following steps: (1) A adaptively chooses the message $\mu_i \in \{0,1\}^*$, $1 \leq i \leq q_s$, based upon its entire view, and sends μ_i to the challenger; (2) Given the secret key sk as well as the message μ_i to be signed , the challenger generates the associated signature, denoted σ_i, and sends it back to A.
- Output. Finally, A outputs a pair (μ, σ), and wins if
  (1) $\mathsf{Verify}(\mathsf{pk}, \mu, \sigma) = 1$
  (2)
$$(\mu, \sigma) \notin \{(\mu_1, \sigma_1), \cdots, (\mu_{q_s}, \sigma_{q_s})\}.$$

We say the signature scheme Π is *strongly existentially unforgeable under adaptive chosen-message attack* (or SEU-CMA secure for short), if for every probabilistic polynomial-time attacker A, the probability that A wins in the foregoing security game is negligible in λ.

The standard security game, *i.e.*, the EU-CMA security game, could be define by requiring that A wins if and only if (1) $\mathsf{Verify}(\mathsf{pk}, \mu, \sigma) = 1$ and (2) $\mu \notin \left\{\mu_1, \mu_2, \cdots, \mu_{q_s}\right\}$. Then Π is called *(standard) existentially unforgeable under adaptive chosen-message attack* (or EU-CMA secure for short), if for every probabilistic polynomial-time attacker A, the probability that A wins in this standard security game is negligible (in λ).

It should be noted that the lattice-based digital signature scheme Dilithium can be proven secure in the SEU-CMA game in the random oracle model [8, 20].

B ARM NEON IMPLEMENTATION DETAILS FOR ALGORTHM 9

Algorithm 11 prepare_table_offset(s, c)

Input: s(uint16_t∗), $c = \sum_{i=0}^{n-1} c_i \cdot x^i \in B_\tau$
Output: $s_i = (c_i = 1)\ ?\ n - i : ((c_i = -1)\ ?\ 2n - i : 0)$
$s = \{0\}$
for $i \in (0, 1, \cdots, n-1)$ **do**
  if $c_i = 1$ **then**
    $s_i := n - i$
  end if
  if $c_i = -1$ **then**
    $s_i := 2n - i$
  end if
end for

Algorithm 12 neon_array_acc(w, v)

Input: w(uint64_t∗), v(uint64_t∗)
Output: $w_i = w_i + v_i, i \in \{0, n-1\}$
mov cnt, #32
while $cnt != 0$ **do**
  ld1 $\{v1, v2, v3, v4\}, [w]$
  ld1 $\{v5, v6, v7, v8\}, [v]$, #64
  add $v1, v1, v5$
  add $v2, v2, v6$
  add $v3, v3, v7$
  add $v4, v4, v8$
  st1 $\{v1, v2, v3, v4\}, [w]$, #64
  sub cnt, cnt, #1
end while

Algorithm 13 neon_cal_table(v, a, m, U)

Input: v(uint64_t∗), a(int32_t∗), $m = log_2(M), U = U$
Output: $v_i = (v_i \ll m) | (U + a_i), i \in \{j, j+1, \cdots, j+7\}$
ld1 $\{v1, v2, v3, v4\}, [a]$
dup $v0, U$
saddl $v1, v1, v0$ ▷ $k_1 = (uint64\_t)(a_i + U)$
saddl $v2, v2, v0$
saddl $v3, v3, v0$
saddl $v4, v4, v0$
ld1 $\{v5, v6, v7, v8\}, [v]$
dup $v0, m$
ushl $v5, v5, v0$ ▷ $k_2 = v_i \ll m$
ushl $v6, v6, v0$
ushl $v7, v7, v0$
ushl $v8, v8, v0$
orr $v5, v5, v1$ ▷ $k_2 = k_1 | k_2$
orr $v6, v6, v2$
orr $v7, v7, v3$
orr $v8, v8, v4$
st1 $\{v5, v6, v7, v8\}, [v]$

Algorithm 14 neon_st_table(v, γ)

Input: v(uint64_t$*$), γ(uint64_t)
Output: $v_{i+2n} = v_i = \gamma - v_{i+n}, i \in \{j, j+1, \cdots, j+7\}$

dup $v0, \gamma$
ld1 $\{v1, v2, v3, v4\}, [v]$
sub $v1, v0, v1$ ▷ $k = \gamma - v_{i+n}$
sub $v2, v0, v2$
sub $v3, v0, v3$
sub $v4, v0, v4$
sub v, v, #256 $*$ 8
st1 $\{v1, v2, v3, v4\}, [v]$ ▷ $v_i = k$
add v, v, #256 $*$ 8 $*$ 2
st1 $\{v1, v2, v3, v4\}, [v]$ ▷ $v_{i+2n} = k$

Algorithm 15 neon_reduce32(a, t, ad, mu, s)

Input: $a \le 2^{31} - 2^{22} - 1$, t(temporary register), $ad = 2^{22}$, $mu = -Q$, $s = 23$
Output: $a = a \mod Q$

add t, a, ad ▷ $t = a + (1 \ll 22)$
sshr t, t, s ▷ $t = t \gg 23$
mla a, t, mu ▷ $t = a + (t \cdot (-Q))$

Algorithm 16 neon_evaluate(u, t, m, tU, s)

Input: u(int32_t$*$), t(uint64_t$*$), $m = M - 1$, $tU = \tau U$, $s = \log_2(M)$
Output: $u_i = t_i \& m - tU, i \in \{j, j+1, \cdots, j+7\}$
Output: $t_i = t_i \gg s, i \in \{j, j+1, \cdots, j+7\}$

ld1 $\{v1, v2, v3, v4\}, [t]$
dup $v0, m$
and $v5, v1, v0$ ▷ $k_1 = t_i \& m$
and $v6, v2, v0$
and $v7, v3, v0$
and $v8, v4, v0$
dup $v0, tU$
sub $v5, v5, v0$ ▷ $k_1 = k_1 - tU$
sub $v6, v6, v0$
sub $v7, v7, v0$
sub $v8, v8, v0$
xtn $v9, v5$ ▷ $k_2 = (int32\_t)k_1$
xtn $v10, v6$
xtn $v11, v7$
xtn $v12, v8$
ushr $v1, v1, s$ ▷ $t_i = t_i \gg s$
ushr $v2, v2, s$
ushr $v3, v3, s$
ushr $v4, v4, s$

Algorithm 17 neon_evaluate_cs(u, t, m, tU)

Input: u(int32_t$*$), t(uint64_t$*$), $m = M - 1$, $tU = \tau U$
Output: $u_i = t_i \& m - tU, i \in \{j, j+1, \cdots, j+7\}$
Output: $t_i = t_i \gg 8$(or 9)$, i \in \{j, j+1, \cdots, j+7\}$

neon_evaluate u, t, m, tU, #8(or #9) ▷ (#8 in Dilithium2, #9 in Dilithium3/5)
st1 $\{v1, v2, v3, v4\}, [t]$
st1 $\{v9, v10, v11, v12\}, [u]$

Algorithm 18 neon_evaluate_ct0(u, t, m, tU)

Input: u(int32_t$*$), t(uint64_t$*$), $m = M - 1$, $tU = \tau U$
Output: $u_i = t_i \& m - tU, i \in \{j, j+1, \cdots, j+7\}$
Output: $t_i = t_i \gg 19, i \in \{j, j+1, \cdots, j+7\}$

neon_evaluate u, t, m, tU, #19
st1 $\{v1, v2, v3, v4\}, [t]$
st1 $\{v9, v10, v11, v12\}, [u]$

Algorithm 19 neon_evaluate_ct1(u, t, m, tU, q)

Input: u(int32_t$*$), t(uint64_t$*$), $m = M - 1$, $tU = \tau U$, $q = -Q$
Output: $u_i = t_i \& m - tU \pmod Q, i \in \{j, j+1, \cdots, j+7\}$
Output: $t_i = t_i \gg 17, i \in \{j, j+1, \cdots, j+7\}$

mov $w5$, #4194304 ▷ $w5 = 2^{22}$
dup $v14, w5$
dup $v15, q$
neon_evaluate u, t, m, tU, #17
shl $v9, v9$, #13 ▷ $k_2 = k_2 \ll 13$
shl $v10, v10$, #13
shl $v11, v11$, #13
shl $v12, v12$, #13
neon_reduce32 $v9, v13, v14, v15$, #23 ▷ $k_2 = k_2 \mod Q$
neon_reduce32 $v10, v13, v14, v15$, #23
neon_reduce32 $v11, v13, v14, v15$, #23
neon_reduce32 $v12, v13, v14, v15$, #23
st1 $\{v1, v2, v3, v4\}, [t]$
st1 $\{v9, v10, v11, v12\}, [u]$

C ANOTHER VARIANT OF ALGORITHM 9

Algorithm 20 A parallel index-based polynomial multiplication algorithm with translations (another version of Algorithm 9)

Input: $\left(\mathbf{c}, \overrightarrow{\mathbf{a}}\right)$, where

- $\mathbf{c} = \sum_{i=0}^{n-1} c_i \cdot x^i \in B_\tau$;
- $\overrightarrow{\mathbf{a}} = \left\{\mathbf{a}^{(j)}\right\} \in \mathcal{R}_q^r$;
- Every $\mathbf{a}^{(j)} = \sum_{i=0}^{n-1} a_i^{(j)} \cdot x^i \in \mathcal{R}_q$;
- Every $a_i^{(j)} \in \{-U, \cdots, U\}$

Output: $\overrightarrow{\mathbf{u}} = \left[\mathbf{u}^{(0)}, \cdots, \mathbf{u}^{(r-1)}\right]^T \in \mathcal{R}_q^r$, where

- $\mathbf{u}^{(j)} = \mathbf{c} \cdot \mathbf{a}^{(j)} \in \mathcal{R}_q$;

for $i \in \{0, 1, \cdots, n-1\}$ **do**
    $w_i := 0$
    $v_i = v_{i-n} := 0$
    $\overline{v}_i = \overline{v}_{i-n} := 0$
    for $j = 0$ **to** $r-1$ **do**
        $v_i := v_i \cdot M + \left(U + a_i^{(j)}\right)$
        $v_{i-n} := v_{i-n} \cdot M + \left(U - a_i^{(j)}\right)$
        $\overline{v}_i := \overline{v}_i \cdot M + \left(U - a_i^{(j)}\right)$
        $\overline{v}_{i-n} := \overline{v}_{i-n} \cdot M + \left(U + a_i^{(j)}\right)$
    end for
end for
$\gamma := 2U \cdot \frac{M^r-1}{M-1}$ ▷ $\gamma \in \mathbb{Z}^{>0}$
for $i \in \{0, 1, \cdots, n-1\}$ **do**
    if $c_i = 1$ **then**
        for $j \in \{0, 1, \cdots, n-1\}$ **do**
            $w_j := w_j + v_{j-i}$
        end for
    end if
    if $c_i = -1$ **then**
        for $j \in \{0, 1, \cdots, n-1\}$ **do**
            $w_j := w_j + \overline{v}_{j-i}$ ▷ $\gamma = 2U \cdot \frac{M^r-1}{M-1}$
        end for
    end if
end for
for $i \in \{0, 1, \cdots, n-1\}$ **do**
    $t := w_i$
    for $j = 0$ **to** $r-1$ **do**
        $u_i^{(r-1-j)} := (t \bmod M) - \tau U \pmod{q}$
        $t := \lfloor t/M \rfloor$
    end for
end for
for $j \in \{0, 1, \cdots, r-1\}$ **do**
    $\mathbf{u}^{(j)} := \sum_{i=0}^{n-1} u_i^{(j)} \cdot x^i$
end for
$\overrightarrow{\mathbf{u}} := \left[\mathbf{u}^{(0)}, \cdots, \mathbf{u}^{(r-1)}\right]^T$
return $\overrightarrow{\mathbf{u}}$