IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 1, pp. 160–202. DOI:10.46586/tches.v2025.i1.160-202

Multiplying Polynomials without Powerful Multiplication Instructions

Vincent Hwang[1], YoungBeom Kim[2] and Seog Chung Seo[2]

[1] Max Planck Institute for Security and Privacy, Bochum, Germany
vincentvbh7@gmail.com
[2] Kookmin University, Seoul, Korea
darania@kookmin.ac.kr, scseo@kookmin.ac.kr

Abstract. We improve the performance of lattice-based cryptosystems Dilithium on Cortex-M3 with expensive multiplications. Our contribution is two-fold: (i) We generalize Barrett multiplication and show that the resulting shape-independent modular multiplication performs comparably to long multiplication on some platforms without special hardware when precomputation is free. We call a modular multiplication "shape-independent" if its correctness and efficiency depend only on the magnitude of moduli and not the shapes of the moduli. This was unknown in the literature even though modular multiplication has been studied for more than 40 years. In the literature, shape-independent modular multiplications often perform several times slower than long multiplications even if we ignore the cost of the precomputation. (ii) We show that polynomial multiplications based on Nussbaumer fast Fourier transform and Toom–Cook over $\mathbb{Z}_{2^k}$ perform the best when modular multiplications are expensive and k is not very close to the arithmetic precision.

For practical evaluation, we implement assembly programs for the polynomial arithmetic used in the digital signature Dilithium on Cortex-M3. For the modular multiplications in Dilithium, our generalized Barrett multiplications are 1.92 times faster than the state-of-the-art assembly-optimized Montgomery multiplications, leading to $1.38-1.51$ times faster Dilithium NTT/iNTT. Along with the improvement in accumulating products, the core polynomial arithmetic matrix-vector multiplications are $1.71-1.77$ times faster. We further apply the FFT-based polynomial multiplications over $\mathbb{Z}_{2^k}$ to the challenge polynomial multiplication ct_0, leading to 1.31 times faster computation for ct_0.

We additionally apply the ideas to Saber on Cortex-M3 and demonstrate their improvement to Dilithium and Saber on our 8-bit AVR environment. For Saber on Cortex-M3, we show that matrix-vector multiplications with FFT-based polynomial multiplications over $\mathbb{Z}_{2^k}$ are $1.42-1.46$ faster than the ones with NTT-based polynomial multiplications over NTT-friendly coefficient rings. When moving to a platform with smaller arithmetic precision, such as 8-bit AVR, we improve the matrix-vector multiplication of Dilithium with our Barrett-based NTT/iNTT by a factor of 1.87–1.89. As for Saber on our 8-bit AVR environment, we show that matrix-vector multiplications with NTT-based polynomial multiplications over NTT-friendly coefficient rings are faster than polynomial multiplications over $\mathbb{Z}_{2^k}$ due to the large k in Saber.

Keywords: Lattice-based cryptography · Dilithium · Saber · Barrett multiplication · Microcontroller · Nussbaumer FFT · Toom–Cook

1 Introduction

At PQCrypto 2016, the National Institute of Standards and Technology (NIST) announced the Post-Quantum Cryptography Standardization soliciting standards on post-quantum

Received: 2024-07-15 Accepted: 2024-09-15 Published: 2024-12-09

cryptography. Among various candidates, lattice-based cryptosystems usually exhibit reasonably balanced public key, secret key, ciphertext, and signature sizes. In most lattice-based cryptosystems, the designers choose between several algebraic structures while considering various implementation considerations. In the specifications of Kyber [ABD+20b], a key encapsulation mechanism standardized by NIST, and Dilithium [ABD+20a], a digital signature standardized by NIST, the authors wrote specific number-theoretic transforms (NTTs) over the so-called “NTT-friendly” primes to the schemes due to the quasi-linear algebraic complexity of multiplying polynomials with NTTs. On the other hand, the authors of Saber [DKRV20], a 3rd round KEM finalist, argued the efficiency of computing modulo a power of two while resorting to Toom–Cook and Karatsuba with less performant asymptotic runtime.

The software performance of polynomial multiplications has been extensively studied. On high-end processors, [Sei18, CHK+21, NG21, SKS+21a, SKS21b, BBCT22, BHK+22b, BMK+22, ORGF+22, ZHS+22, CCHY24, HLY24, Hwa24a] studied the interactions between algebraic structures and vectorization. On low-end processors and platforms, [KRS19, BKS19, MKV20, IKPC20, GKS21, ACC+21, CHK+21, ACC+22, IKPC22, AHY22] studied various aspects of microcontroller implementations, including performance cycles, memory usage, code size, and more. In these works, the designs of polynomial multipliers consist of (i) specific approaches implementing modular multiplications with the designated instruction set architectures/extensions and (ii) specific fast transformations converting the large multiplication tasks into several small multiplication tasks.

In the literature, there are many popular lines of modular multiplications. We call a modular multiplication “shape-independent” if its correctness and efficiency depend only on the magnitude of moduli and not the shapes of the moduli. There are three lines of shape-independent modular multiplications computing representatives of ab modulo q with precomputations – Montgomery, Barrett, and Plantard. Montgomery multiplication computes the modular product as the high part of the sum of long products [Mon85, Sei18]. Recently, [Pla21] proposed the unsigned Plantard multiplication using integer middle products, an operation that is no cheaper than a high multiplication[1]. The idea was later adapted to the signed version by [HZZ+22] using special multiplication instructions unseen on most of the architectures and by [AMOT22] using multiplication instructions with twice the precision. Montgomery multiplication amounts to computing two long/high multiplications and one low multiplication, and Plantard multiplication amounts to computing one middle product and one long/high multiplication. This work proposes a shape-independent modular multiplication that performs comparably to a long multiplication on some platforms without special hardware. We rely on two observations:

- Barrett multiplication: Barrett multiplication computes the modular product with one high multiplication and two low multiplications [Sho, BHK+22b].
- Approximation nature of the high multiplication: In Barrett multiplication, the high multiplication only needs to be approximately correct in contrast to Montgomery and Plantard multiplications.

Contribution 1: We generalize the notion of approximation that is suitable for approximating the high multiplication with multi-limb arithmetic and show that the resulting generalized Barrett multiplication performs comparably to long multiplication on our platform, defying the expectation that modular multiplication, such as Montgomery and Plantard multiplications, must amount to a long multiplication followed by non-negligible computations. To the best of our knowledge, this is the first time that shape-independent modular multiplication performs comparably to long

[1] Formally, one can implement a high multiplication with a middle product, so the cost of one middle product cannot be cheaper than one high multiplication.

multiplication on a platform like Cortex-M3 with a relatively simple cost model, even though people have been studying modular multiplications for more than 40 years. We also believe that our Barrett multiplication performs the best on other Cortex-M processors, such as Cortex-M0, Cortex-M0+, and Cortex-M23, with similar characteristics.

The second part of the paper challenges the notion of "NTT-friendliness" of coefficient rings in practice. Although FFT/NTT-based polynomial multiplication over arbitrary rings was already known more than 20 years ago [CK91], several recent works presented NTTs in severely restricted forms. NTT was usually presented as a fast transformation over an "NTT-friendly" coefficient ring $\mathbb{Z}_q$ for an odd q. In fact, NTTs can be defined over unital and possibly non-commutative rings by manufacturing various "NTT-friendly" structures while extending the polynomial rings.

Contribution 2: We propose a polynomial multiplier based on Nussbaumer FFT and Toeplitz matrix-vector product built upon Toom-4 over $\mathbb{Z}_{2^k}$, and show that the resulting implementation performs the best when k is not very close to the arithmetic precision. To the best of our knowledge, this is the first implementation of FFT-based polynomial multipliers over $\mathbb{Z}_{2^k}$. On the other hand, if k is close to the arithmetic precision, we show that NTT-based polynomial multiplications over NTT-friendly coefficient rings perform the best.

Our results bring insights into the impact of coefficient rings on polynomial multiplications. If the coefficient ring is an NTT-friendly one modulo a prime and the corresponding long/high multiplications are slow, then one should use Barrett multiplication instead of Montgomery multiplication. On the other hand, if the coefficient ring takes the form $\mathbb{Z}_{2^k}$ and k is small enough, FFT-based polynomial multiplication over $\mathbb{Z}_{2^k}$ performs the best; and when k is large, the standard NTT-based approach over NTT-friendly coefficient rings performs the best.

Applications and limitations of our Barrett multiplications. For a positive integer n, we identify $\mathbb{Z}_n := \left[-\frac{n}{2}, \frac{n}{2}\right) \cap \mathbb{Z}$ and define the map $\bmod^{\pm} n$ as the function mapping an integer z to the element in $\mathbb{Z}_n$ differing by a multiple of n. Let $\mathtt{R}$ be a power of two with exponent a power of two, $\log_2 \mathtt{R}$ be the arithmetic precision, q be an odd modulus with $\sqrt{\mathtt{R}} < q < \mathtt{R}$. For two integers $a, b \in \mathbb{Z}_{\mathtt{R}}$, this work studies the computation of a representative $c \in \mathbb{Z}_{\mathtt{R}}$ of ab modulo q. In practice, a modular multiplication usually admits an integer $\mathtt{B}$ with $q \le \mathtt{B} \le \mathtt{R}$ such that $c \in \mathbb{Z}_{\mathtt{B}}$. We study the relations between the efficiency of modular multiplications and the magnitude of q and $\mathtt{B}$, and do not exploit the structure of q other than being an odd integer[2]. In terms of the definability of shape-independent modular multiplications, Barrett, Montgomery, and Plantard multiplications compute a representative $c \in \mathbb{Z}_{\mathtt{B}}$ of ab modulo q with a pre-modular-multiplication by a constant in the Barrett case and a post-modular-multiplication by a constant in the Montgomery and Plantard cases. In our context, we focus on the 32-bit modular multiplications ($\mathtt{R} = 2^{32}$) in the Dilithium NTT/iNTT computations, where pre-modular-multiplications are carried out during the design phase rather than the computation phase. By relaxing the worst-case[3] of $\mathtt{B}$ to be five times larger than prior signed Barrett multiplication [BHK+22b], our generalized signed Barrett multiplication outperforms any other modular multiplications consisting of at least one high/long multiplication and non-negligible pre-/post-computation on Cortex-M3,

[2] In fact, Barrett multiplication works when q is even, but the resulting computation requires a slightly more involved justification and does not find applications in our context.

[3] By worst case, we mean the theoretical worst-case analysis, which does not imply that the resulting bound is attainable. The new bound $\mathtt{B}$ could be smaller than the theoretical worst-case analysis in practice. Typically, brute-force testing with ball analysis will reveal a much better bound. In this paper, we stick to the theoretical worst-case analysis since the bound is much more modular than the brute-force approach.

including Montgomery and Plantard multiplications as well as other approaches applying specialized modular reductions to the long products. Furthermore, we also show that our generalized Barrett multiplication only reduces the bit-sizes of valid moduli from 28.1926 to 26.0458 for an 8-layer 32-bit NTT with only the necessary modular multiplications.

Source code. Our source code is publicly available at `https://github.com/vincentvbh/PolyMul_Without_PowerfulMul`.

2 Preliminiaries

2.1 Integer Approximation

For a function $\llbracket \rrbracket : \mathbb{R} \to \mathbb{Z}$, [BHK⁺22b] call it an integer approximation if $\forall r \in \mathbb{R}, |r - \llbracket r \rrbracket| \leq 1$. Common examples are the floor function $\lfloor \rfloor$, ceiling function $\lceil \rceil$, and rounding-half-up function $\lfloor \rceil$. [BHK$^+$22b] chose $\lfloor \rceil_2 := r \mapsto 2 \left\lfloor \frac{r}{2} \right\rceil$ and demonstrated its benefit for the vector instruction set Neon in Armv8-A. It is easily seen that $\lfloor \rfloor, \lceil \rceil, \lfloor \rceil, \lfloor \rceil_2$ are all integer approximations.

2.2 Modular Multiplications

Throughout this paper, we consider $\mathtt{R} = 2^{32}$ and $q < \mathtt{R}$ an odd number, and focus on signed arithmetic. For an integer approximation $\llbracket \rrbracket$, we define the corresponding modular reduction $\bmod^{\llbracket \rrbracket} q : \mathbb{Z} \to \mathbb{Z}$ as $\bmod^{\llbracket \rrbracket} q := z \mapsto z - \left\llbracket \frac{z}{q} \right\rrbracket q$. Furthermore, we define $\left| \bmod^{\llbracket \rrbracket} q \right| := \max_{z \in \mathbb{Z}} \left| z \bmod^{\llbracket \rrbracket} q \right|$. If $\llbracket \rrbracket = \lfloor \rceil$, we denote $\bmod^{\lfloor \rceil}$ as $\bmod^{\pm}$.

Arithmetic modulo $\mathtt{R} = 2^m$. In most modern platforms, since elements are usually presented as bit strings with power-of-two bits, arithmetic modulo a power of two $\mathtt{R} = 2^m$ with m a power of two attributes to straightforward quantification of performance cycles. For two m-bit integers a and b, we call an instruction long multiplication if it computes the $2m$-bit result ab. If an instruction computes a value sufficiently close to the upper m bits of ab, we call it a high multiplication. Further, we call an instruction low multiplication if it computes a value sufficiently close to the lower m bits of ab. For simplicity, we also call the subtractive and accumulative variants long, high, and low multiplications.

Montgomery multiplication. Let $a, b \in \left[-\frac{\mathtt{R}}{2}, \frac{\mathtt{R}}{2}\right)$ be two m-bit integers. Montgomery multiplication computes a representative of $ab\mathtt{R}^{-1} \bmod q$ with two long multiplications and one low multiplication as shown in Algorithm 1. Intuitively, we first compute a value c satisfying $c \equiv 0 \pmod{\mathtt{R}}$, $c \equiv ab \pmod{q}$, and $|c| < |ab| + \frac{\mathtt{R}q}{2}$. Since $\frac{c}{\mathtt{R}}$ is an integer and $\mathtt{R}$ is coprime to q, we have $\frac{c}{\mathtt{R}} \equiv ab\mathtt{R}^{-1} \pmod{q}$.

Barrett multiplication. Barrett multiplication was first introduced only in the reduction form, reducing a value by subtracting a reasonably approximated multiple of q [Bar86]. [Sho] proposed the multiplicative form for unsigned arithmetic, and [BHK$^+$22b] proposed the signed multiplication with integer approximations as shown in Algorithm 2. [BHK$^+$22b] computed a representative of $ab \bmod q$ by pulling the operand b to the approximation and showed that the result is an m-bit value by establishing a correspondence between Barrett and Montgomery multiplications and reusing the bound from Montgomery multipilication.

Algorithm 1 Montgomery multiplication.

```
Inputs: $a, b$.
Output: $c \equiv ab\mathtt{R}^{-1} \bmod^{\pm} q$.
1: $\mathtt{tfull} = a \cdot b$
2: $\mathtt{tlo} = \mathtt{tfull} \bmod^{\pm} \mathtt{R}$
3: $q' = -q^{-1} \bmod^{\pm} \mathtt{R}$
4: $\mathtt{tlo} = \mathtt{tlo} \cdot q' \bmod^{\pm} \mathtt{R}$
5: $\mathtt{tfull} = \mathtt{tfull} + \mathtt{tlo} \cdot q$
6: $c = \left\lfloor \frac{\mathtt{tfull}}{\mathtt{R}} \right\rfloor$
```

Algorithm 2 Barrett multiplication.

```
Inputs: $a, b$.
Output: $c \equiv ab \bmod^{\pm} q$.
1: $\mathtt{tlo} = a \cdot b \bmod^{\pm} \mathtt{R}$
2: $\mathtt{thi} = \left\lfloor \frac{a \cdot \left\lfloor \frac{b\mathtt{R}}{q} \right\rceil}{\mathtt{R}} \right\rceil$
3: $c = \mathtt{tlo} - (\mathtt{thi} \cdot q \bmod^{\pm} \mathtt{R})$
```

A correspondence between Barrett and Montgomery multiplications. [BHK⁺22b] showed that for an integer approximation $\llbracket\rrbracket$, we have

$$ab - \left\lfloor \frac{a \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket}{\mathtt{R}} \right\rceil q = \frac{a \left(b\mathtt{R} \bmod^{\llbracket\rrbracket} q\right) + \left(a \left(b\mathtt{R} \bmod^{\llbracket\rrbracket} q\right) \left(-q^{-1}\right) \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}}.$$

Their proof clearly transfers to the following generalization: For integer approximations $\llbracket\rrbracket_0, \llbracket\rrbracket_1$, we have

$$ab - \left\llbracket \frac{a \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket_0}{\mathtt{R}} \right\rrbracket_1 q = \frac{a \left(b\mathtt{R} \bmod^{\llbracket\rrbracket_0} q\right) + \left(a \left(b\mathtt{R} \bmod^{\llbracket\rrbracket_0} q\right) \left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}\right) q}{\mathtt{R}}.$$

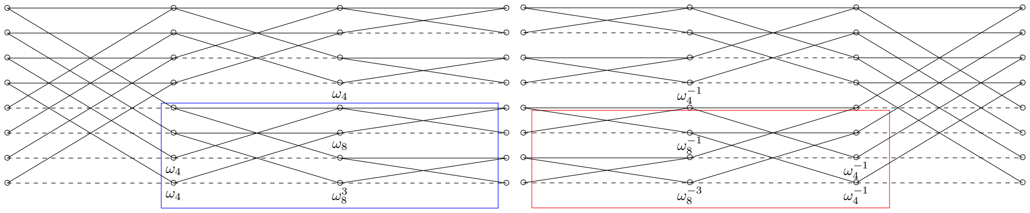
Applying the correspondence, we have $\left| ab - \left\llbracket \frac{a \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket_0}{\mathtt{R}} \right\rrbracket_1 q \right| \leq \frac{|a| \left| \bmod^{\llbracket\rrbracket_0} q \right| + \left| \bmod^{\llbracket\rrbracket_1} \mathtt{R} \right| q}{\mathtt{R}}$. When $\llbracket\rrbracket_0 = \llbracket\rrbracket_1 = \lfloor\rceil$, we have $\left| ab - \left\llbracket \frac{a \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket_0}{\mathtt{R}} \right\rrbracket_1 q \right| \leq \frac{q}{2} \left(1 + \frac{|a|}{\mathtt{R}}\right)$.

2.3 Number Theoretic Transform

Let R be a commutative ring with identity. For an n-th root of unity $\omega_n \in R$, we call ω_n principal if $\forall j = 1, \dots, n-1, \sum_{i=0}^{n-1} \omega_n^{ij} = 0$. If there is a principal n-th root of unity for a large n, we call R an "NTT-friendly" coefficient ring. In this paper, we only consider n a power of two, whose condition is equivalent to $\omega_n^{\frac{n}{2}} = -1$ [Für09, Lemma 2.1]. For an invertible element $\zeta \in R$, we have the following isomorphism by the Chinese remainder theorem for polynomial rings:

$$\frac{R[x]}{\langle x^n - \zeta^n \rangle} \cong \prod \frac{R[x]}{\left\langle x^{\frac{n}{2}} \pm \zeta^{\frac{n}{2}} \right\rangle} \cong \cdots \cong \prod_{i_0, \dots, i_{\log_2 n - 1} = 0, 1} \frac{R[x]}{\left\langle x - \zeta \omega_n^{\sum_j i_j 2^j} \right\rangle}.$$

This is the radix-2 Cooley–Tukey FFT for a discrete weighted transform [CT65, CF94]. We call $\zeta = \omega_n$ the cyclic case and $\zeta = \omega_{2n}$ the negacyclic case, and illustrate the idea in Figure 1. If $R = \mathbb{Z}_{2^{2^t}+1}$, this is called Fermat number transform (FNT) since $2^{2^t} + 1$ is a Fermat number [SS71, AB74]. There are several benefits while operating in this coefficient ring. First of all, since $2^{2^t} = -1 \in \mathbb{Z}_{2^{2^t}+1}$, we have 2 a principal 2^{t+1}-th root of unity for Cooley–Tukey FFT. This improves the performance of twiddle factor multiplications since multiplications by powers of two are fast [SS71, AB74, AHKS22, BHK⁺22a]. The second benefit is the efficient modular reduction [SS71, AHKS22, BHK⁺22a].



(a) NTT over $x^8 - 1$ and over $x^4 + 1$.

(b) Inverse of NTT over $x^8 - 1$ and over $x^4 + 1$.

Figure 1: NTT and its inverse over $x^8 - 1$ and $x^4 + 1$. $\omega_n = \omega^{8/n}$ where ω is a principal 8th root of unity, adapted from [ACC⁺22]. Computational flow goes from left to right.

2.4 Nussbaumer Fast Fourier Transform

Nussbaumer fast Fourier transform manufactures algebraic structures suitable for NTT-based algebra homomorphisms, and the resulting transformation requires only additions and subtractions [Nus80]. For simplicity, we illustrate the idea when the polynomial modulus is a power-of-two cyclotomic polynomial.

The goal is to design fast transformations for the ring $R[x]/\langle x^n + 1\rangle$ with only additions and subtractions in R where $n = 2^k$. Conceptually, we chop a size-n polynomial into $\Theta\left(\sqrt{n}\right)$ polynomials of size $\Theta\left(\sqrt{n}\right)$ while manufacturing principal roots of unity by zero-padding. Formally, we choose an $n' = 2^{\left\lfloor \frac{\log n}{2} \right\rfloor}$ and rewrite $R[x]/\langle x^n + 1\rangle$ as $\mathcal{R}[x]\Big/\Big\langle x^{n'} - y\Big\rangle$ where $\mathcal{R} = R[y]\Big/\Big\langle y^{\frac{n}{n'}} + 1\Big\rangle$. Since the x-degree of a product of two polynomials with coefficient ring $\mathcal{R}$ is bounded by $2n' - 2$, one can choose a polynomial modulus with x-degree larger than $2n' - 2$ for efficient computation. We replace the relation $x^{n'} \sim y$ with $x^{2n'} \sim 1$ by zero-padding. Since $y^{\frac{n}{n'}} = -1$ in $\mathcal{R}$ by definition, y is a principal $\frac{2n}{n'}$-th root of unity supporting a size-$\frac{2n}{n'}$ radix-2 cyclic Cooley–Tukey FFT over $\mathcal{R}$. By the choice of n', $\frac{2n}{n'} \geq 2n'$ and $\mathcal{R}[x]\Big/\Big\langle x^{2n'} - 1\Big\rangle$ splits into polynomial rings of the form $\mathcal{R}[x]/\langle x - y^i\rangle$ with additions, subtractions, and multiplications by powers of y in $\mathcal{R}$. Since multiplications by powers of y amount to negacyclic shifts, the entire transformation only requires additions and subtractions in R. If we apply the idea recursively, then the computing task is converted into R^h where h grows proportionally to $n \log_2 n$. In this work, we apply the idea only once and switch to different approaches for the small-dimensional computing task.

2.5 Toeplitz Matrix-Vector Product

In lattice-based cryptography, the Toeplitz matrix-vector product (TMVP) was applied to Saber and NTRU on Cortex-M4 [IKPC20, IKPC22], and NTRU and NTRU Prime on Cortex-A72 [Hwa24a, CCHY24]. Conceptually, for an algebra homomorphism multiplying polynomials in $R[x]$, its module-theoretic dual implements a Toeplitz matrix-vector product with the same algebraic complexity [Fid73, Win80, CCHY24]. We call a square matrix Toeplitz if elements belonging to the same diagonal are the same, and a matrix-vector product TMVP if the matrix is a Toeplitz matrix. Prominent examples are polynomial multiplications modulo $x^n - \zeta$ for a $\zeta \in R$. Suppose we have an algebra homomorphism f computing the product $\boldsymbol{ab} = f^{-1}\left(f(\boldsymbol{a})f(\boldsymbol{b})\right)$ of two size-n polynomials $\boldsymbol{a}, \boldsymbol{b}$ in $R[x]$, then $f^* \circ \left(\boldsymbol{b}' \mapsto f\left(\boldsymbol{a}\right)\boldsymbol{b}'\right)^* \circ \left(f^{-1}\right)^* \left(\boldsymbol{b}\right)$ computes the reversal of an $n \times n$ TMVP where * is the module-theoretic dualization, corresponding to matrix transposition in terms of matrix manipulation.

Asymmetric nature of TMVP. The observation relevant to us is the asymmetric nature of TMVP, which was first introduced under the name "asymmetric multiplication" in [BHK+22b][4]: In practice, the dual of a module homomorphism usually results in the same algebraic complexity. If f^{-1} is much more expensive than f, dualizing the entire process and applying $\left(f^{-1}\right)^*$ to the most frequently used operand is more advantageous. For example, suppose we want to compute polynomial products $\boldsymbol{a}_0\boldsymbol{b}$ and $\boldsymbol{a}_1\boldsymbol{b}$ whose reversals are Toeplitz matrix-vector products. If we proceed symmetrically, we need three (fast) f's and two expensive f^{-1}'s. We can instead dualize the maps and apply $\left(f^{-1}\right)^*$ to $\boldsymbol{b}$. The remaining operations are equivalent to applying four f's, replacing an expensive f^{-1} by an f^*.

A running example with Karatsuba. We illustrate the idea with Karatsuba [KO62] shown in [Win80]. Suppose we have an algebra homomorphism computing $(a_0 + a_1x)(b_0 + b_1x) = a_0b_0 + (a_0b_1 + a_1b_0)x + a_1b_1x^2$ by first applying the forward map $a_0 + a_1x \mapsto (a_0, a_0 + a_1, a_1)$ to both operands. We then multiply the resulting images and apply the "inversion map" $(c_0, c_1, c_2) \mapsto c_0 + (c_1 - c_0 - c_2)x + c_2x^2$. This gives us the desired product $a_0b_0 + (a_0b_1 + a_1b_0)x + a_1b_1x^2$. We illustrate how to turn the above computation into a computation implementing the Toeplitz matrix-vector product $\begin{pmatrix} c_1 & c_2 \\ c_0 & c_1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}$. For the vector, we compute $(a_0, a_0 + a_1, a_1)$ same as before. As for the Toeplitz matrix, we extract the elements c_0, c_1, c_2 and apply the dual (or transpose in the matrix view) of the "inversion map". This gives us $(c_0 - c_1, c_1, c_2 - c_1)$. We then point-multiply them and receive $((c_0 - c_1)a_0, c_1(a_0 + a_1), (c_2 - c_1)a_1)$. Finally, we apply the dual of $(b_0, b_1) \mapsto (b_0, b_0 + b_1, b_1)$ and receive $(c_0a_0 + c_1a_1, c_1a_0 + c_2a_1)$, which is the reversal of the desired result.

Naming convention of TMVP. For an algebra homomorphism f multiplying polynomials in $R[x]$, we denote the resulting Toeplitz matrix-vector product as Toeplitz-f. Furthermore, due to the asymmetric nature of TMVP, we denote Hom-V for the module homomorphism applied to the vector operand and Hom-M for the module homomorphism applied to the Toeplitz matrix operand. For the resulting small dimensional TMVP, we denote it as BiHom since it is a bilinear map that is rarely a ring multiplication. Finally, we denote Hom-I for the homomorphism mapping the resulting small dimensional products to the desired result and call it an interpolation.

2.6 Dilithium

Dilithium [ABD+20a] is a digital signature based on "Fiat-Shamir with aborts" [Lyu09]. The security of Dilithium relies on the Module Small Integer Solutions and the Module Learning with Errors problems. The module is a $k \times \ell$ matrix over the polynomial ring $R_q := \mathbb{Z}_q[x]/\langle x^{256} + 1\rangle$ where $q = 2^{23} - 2^{13} + 1$ is a prime and $(k, \ell) = (4, 4), (6, 5), (8, 7)$, depending on the security level. See Table 1 for an overview of parameter sets.

Table 1: Dilithium parameters [ABD+20a] relevant to this work.

Parameter set	NIST security level	k	ℓ	η	τ	# rep.
`dilithium2`	II	4	4	2	39	4.25
`dilithium3`	III	6	5	4	49	5.1
`dilithium5`	V	8	7	2	60	3.85

During the key generation, we sample a $k \times \ell$ matrix $\hat{A}$ over the image of a negacyclic NTT defined on $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$ and two vectors s_1 and s_2 of polynomials with coefficients drawn from $\{-\eta, \dots, 0, \dots, \eta\}$. We then compute the matrix-vector product As_1 and add s_2 to the result. Finally, we round the result and hash (cf. Algorithm 3). In the signature

[4] See [Hwa22, Section 8.3.2] for their relations.

generation, we compute a matrix-vector product where the vector is sampled with a nonce (κ in Algorithm 5) as one of the parameters. We then compute a challenge polynomial c with exactly τ ± 1's and $256 - \tau$ 0's from the product and test if it meets the security requirements. During the testing, we compute the vector of products cs_1, cs_2, and ct_0 where s_1 and s_2 are the same as key generation and t_0 a polynomial with coefficients in $\left\{-2^{12}, \dots, 0, \dots, 2^{12}-1\right\}$ (t_0 is the lower part of the rounding in the key generation). See Algorithm 5 for the details. If any tests fail, we increment the nonce and restart the signature generation. In Table 1, we list the expected number of iterations for generating a desired signature. In the signature verification, we compute a matrix-vector product (cf. Algorithm 4).

This work optimizes the polynomial arithmetic. We outline the target operations of this paper in Algorithms $3-5$. Operations in blue are covered by this work and operations in purple are covered by [HAZ+24] and this work. The polynomial arithmetic improvement of [HAZ+24] does *not* apply to the operations in blue in Algorithms $3-5$.

Algorithm 3 Dilithium key generation.

```
Output: $sk = (r, K, tr, s_1, s_2, t_0)$
Output: $pk = (r, t_1)$
$r \leftarrow \{0,1\}^{256}$
$K \leftarrow \{0,1\}^{256}$
$(s_1, s_2) \leftarrow S_\eta^\ell \times S_\eta^k$
$\hat{A} \in R_q^{k \times \ell} \leftarrow \texttt{ExpandA}(r)$
$\hat{s}_1 \leftarrow \mathsf{NTT}(s_1)$
$t \leftarrow \mathsf{NTT}^{-1}\left(\hat{A}\hat{s}_1\right) + s_2$
$(t_1, t_0) \leftarrow \texttt{Power2Round}(t)$
$tr \in \{0,1\}^{256} \leftarrow \mathcal{H}(r||\mathbf{t}_1)$
```

Algorithm 4 Dilithium verification.

```
Input: $pk = (r, t_1), M \in \{0,1\}^*$
Input: $\sigma = (z, h, \tilde{c})$
Output: Valid or Invalid
$\hat{A} \in R_q^{k \times \ell} \leftarrow \texttt{ExpandA}(r)$
$\hat{z} \leftarrow \mathsf{NTT}(z)$
$\mu \in \{0,1\}^{384} \leftarrow \mathcal{H}(\mathcal{H}(r||t_1)||M)$
$c \leftarrow \mathcal{H}_B(\tilde{c})$
$w_1' \leftarrow \mathsf{NTT}^{-1}\left(\hat{A}\hat{z}\right) - 2^d ct_1$
$w_1' \leftarrow \texttt{UseHint}(h, w_1')$
if $\tilde{c} \neq \mathcal{H}(\mu||w_1')$ or
    $\|z\|_\infty \geq \gamma_1 - \beta$ or
    $\#$ 1's in $h \leq \omega$ then
    return Invalid
end if
```

2.7 Cortex-M3

Cortex-M3 implements the instruction set architecture Armv7-M and is heavily used in industry, including NXP general purpose microcontrollers[5], Infineon microcontrollers[6], and more. We briefly describe the relevant instructions in Armv7-M [ARM21b] and their timing on Cortex-M3 [ARM10a]. `add` adds up two 32-bit values and `sub` subtracts them. `adc` and `sbc` add and subtract the values with carry. `lsl` and `lsr` logically shift a 32-bit value left and right by the specified constant/register value. `asr` performs an arithmetic right-shift. `ubfx` extracts certain consecutive bits and unsigned-extends the result to a 32-bit value. `sbfx` signed-extends the result to a 32-bit value. Each of the above instructions takes one cycle (we exclude the instruction timing involving PC operands). `mul` multiplies two 32-bit values, `mla` accumulates the product to the accumulator, and `mls` subtracts the product from the accumulator. `mul` takes one cycle and `mla/mls` takes two cycles. `{u, s}mull` computes the 64-bit unsigned/signed product of two 32-bit values, and `{u, s}mlal` accumulates the product to an accumulator. `{u, s}{mul, mla}l` takes 3 to 7 cycles and is input-dependent [ARM10a, Table 18-1]. On Cortex-M4, a close relative of Cortex-M3, all the arithmetic instructions take one cycle. See Table 2 for the instruction timings of

[5] https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus:GENERAL-PURPOSE-MCUS.

[6] https://www.infineon.com/cms/en/product/microcontroller/.

Algorithm 5 Dilithium signature generation.

```
**Input:** $sk = (r, K, tr, s_1, s_2, t_0), M \in \{0,1\}^*$
**Output:** Signature $\sigma = (z, h, \tilde{c})$
$\hat{A} \in R_q^{k \times \ell} := \texttt{ExpandA}(r)$; $\mu \in \{0,1\}^{512} \leftarrow \mathcal{H}(tr||M)$; $r' \in \{0,1\}^{512} \leftarrow \mathcal{H}(K||\mu)$; $\kappa \leftarrow 0$
$\hat{s}_1 \leftarrow \mathsf{NTT}(s_1)$; $\hat{s}_2 \leftarrow \mathsf{NTT}(s_2)$; $\hat{t}_0 \leftarrow \mathsf{NTT}(t_0)$
$(z, h) \leftarrow \perp$
while $(z, h) = \perp$ do
    $y \in S_{\gamma_1 - 1}^{\ell} \leftarrow \texttt{ExpandMask}(r', \kappa)$
    $\hat{y} \leftarrow \mathsf{NTT}(y)$; $w \leftarrow \mathsf{NTT}^{-1}\left(\hat{A}\hat{y}\right)$
    $w_1 \leftarrow \texttt{HighBits}(w)$; $\tilde{c} \in \{0,1\}^{256} \leftarrow \mathcal{H}(\mu||w_1)$; $c \leftarrow \mathcal{H}_B(\tilde{c})$
    $\hat{c} \leftarrow \mathsf{NTT}(c)$; $z \leftarrow y + \mathsf{NTT}^{-1}(\hat{c}\hat{s}_1)$; $r \leftarrow \mathsf{NTT}^{-1}(\hat{c}\hat{s}_2)$
    $r_0 \leftarrow \texttt{LowBits}(w - r)$
    if $||z||_\infty \geq \gamma_1 - \beta$ or $||r_0||_\infty \geq \gamma_2 - \beta$ then
        $(z, h) = \perp$
    else
        $h \leftarrow \mathsf{NTT}^{-1}\left(\hat{c}\hat{t}_0\right)$
        $h \leftarrow \texttt{MakeHint}(-h, w - r + h))$
        if $\|ct_0\|_\infty \geq \gamma_2$ or # 1's in $h > \omega$ then
            $(z, h) = \perp$
        end if
    end if
    $\kappa \leftarrow \kappa + 1$
end while
```

the relevant arithmetic instructions on Cortex-M3 and Cortex-M4.

Constant-time concerns of Cortex-M3 long multiplications. The variable run-time of the long multiplications (`smull`, `smlal`, `umull`, `umlal`) on Cortex-M3 is a critical issue for computing on secret data. In the literature, [GOPT10] showed that variable-time multiplication instructions led to straightforward timing side-channel attacks. A workaround is to emulate the long multiplications with multi-limb arithmetic [GKS21], resulting in a significant performance penalty.

Table 2: Summary of instruction timings on Cortex-M3 and Cortex-M4 where inputs are 32-bit registers.

Instruction	Cycle	
	Cortex-M3	Cortex-M4
`add/adc/sub/sbc/lsl/lsr/asr/ubfx/sbfx/mul`	1	1
`mla/mls`	2	1
`smull/smlal/umull/umlal`	3−7	1

3 Algorithm Designs

3.1 Modular Multiplications

We first go through our implementations of modular multiplications. Convention-wise, we call a multiplication modular multiplication if it computes a number equivalent to the product modulo an odd number with the same arithmetic precision. Otherwise, we call it a plain multiplication.

3.1.1 Integer Approximation: Revisited

We generalize the notion of integer approximations. For a function $\llbracket\rrbracket : \mathbb{R} \to \mathbb{Z}$, we call it an integer approximation if $\exists \delta \in \mathbb{R}_{>0}, \forall r \in \mathbb{R}, |r - \llbracket r \rrbracket| \leq \delta$. When δ is known, we call $\llbracket\rrbracket$ a δ-integer-approximation. The generalizations of $\mathrm{mod}^{\llbracket\rrbracket}$ and $\left|\mathrm{mod}^{\llbracket\rrbracket} q\right|$ are defined in the same way.

3.1.2 Generalizing Barrett Multiplication

Let $a \in \left[-\frac{\mathtt{R}}{2}, \frac{\mathtt{R}}{2}\right), b \in \left[-\frac{q}{2}, \frac{q}{2}\right)$ be integers. Recall that Barrett multiplication computes a representative of $ab \bmod q$ as $ab - \left\llbracket \frac{a \llbracket \frac{b\mathtt{R}}{q} \rrbracket_0}{\mathtt{R}} \right\rrbracket_1 q$ for integer approximations $\llbracket\rrbracket_0, \llbracket\rrbracket_1$. If we choose $\llbracket\rrbracket_0 = \llbracket\rrbracket_1 = \lfloor\rceil$, we have the standard Barrett multiplication. If we choose $\llbracket\rrbracket_0 = \lfloor\rceil$ and $\llbracket\rrbracket_1 = \lfloor\rfloor$, we compute a representative of $ab \bmod {}^{\pm}q$ with absolute value bounded by $1.75q$ since the images of $\lfloor\rceil$ and $\lfloor\rfloor$ differ by at most 1, implying an increase in absolute value of at most q. We call the choice $(\llbracket\rrbracket_0, \llbracket\rrbracket_1) = (\lfloor\rceil, \lfloor\rfloor)$ the floor variant of Barrett multiplication. We show that careful choices of $\llbracket\rrbracket_1$ are of practical importance. We choose $\llbracket\rrbracket_0 = \lfloor\rceil$ and $\llbracket\rrbracket_1 = \llbracket\rrbracket_b$ the following integer approximation:

$$\forall r \in \mathbb{R}, \llbracket r \rrbracket_b := \left\lfloor \frac{a_l b_h}{\sqrt{\mathtt{R}}} \right\rfloor + \left\lfloor \frac{a_h b_l}{\sqrt{\mathtt{R}}} \right\rfloor + a_h b_h$$

where $a_l + a_h\sqrt{\mathtt{R}} = \frac{r\mathtt{R}}{\llbracket \frac{b\mathtt{R}}{q} \rrbracket_0}$, $b_l + b_h\sqrt{\mathtt{R}} = \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket_0$ and $a_l, b_l \in [0, \sqrt{\mathtt{R}})$ and call the resulting Barrett multiplication the approximate variant. We first prove $|\lfloor r \rceil - \llbracket r \rrbracket_b| \leq 3$ as follows.

Proof.

$$\begin{aligned}
& |\lfloor r \rceil - \llbracket r \rrbracket_b| \\
= \; & \left| \left\lfloor \frac{\left(a_h b_h + \frac{1}{2}\right)\mathtt{R} + (a_l b_h + a_h b_l)\sqrt{\mathtt{R}} + a_l b_l}{\mathtt{R}} \right\rfloor - \left(a_h b_h + \left\lfloor \frac{a_l b_h}{\sqrt{\mathtt{R}}} \right\rfloor + \left\lfloor \frac{a_h b_l}{\sqrt{\mathtt{R}}} \right\rfloor \right) \right| \\
= \; & \left| \left\lfloor \frac{1}{2} + \left(\frac{a_l b_h}{\sqrt{\mathtt{R}}} - \left\lfloor \frac{a_l b_h}{\sqrt{\mathtt{R}}} \right\rfloor \right) + \left(\frac{a_h b_l}{\sqrt{\mathtt{R}}} - \left\lfloor \frac{a_h b_l}{\sqrt{\mathtt{R}}} \right\rfloor \right) + \frac{a_l b_l}{\mathtt{R}} \right\rfloor \right| \\
\leq \; & \left| \left\lfloor \frac{1}{2} + \frac{\sqrt{\mathtt{R}} - 1}{\sqrt{\mathtt{R}}} + \frac{\sqrt{\mathtt{R}} - 1}{\sqrt{\mathtt{R}}} + \frac{\left(\sqrt{\mathtt{R}} - 1\right)^2}{\mathtt{R}} \right\rfloor \right| \\
= \; & \left| \left\lfloor \frac{1}{2} + \frac{3\mathtt{R} - 4\sqrt{\mathtt{R}} + 1}{\mathtt{R}} \right\rfloor \right| \\
= \; & 3.
\end{aligned}$$

□

Since $|\lfloor r \rceil - \llbracket r \rrbracket_b| \leq 3$, we have

$$\left| ab - \left\llbracket \frac{a \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket_0}{\mathtt{R}} \right\rrbracket_b q \right| \leq \left| ab - \left\lfloor \frac{a \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket_0}{\mathtt{R}} \right\rceil q \right| + 3q \leq \frac{q}{2} \left(7 + \frac{|a|}{\mathtt{R}} \right).$$

Therefore, computing with $ab - \left\llbracket \frac{a \llbracket \frac{b\mathtt{R}}{q} \rrbracket_0}{\mathtt{R}} \right\rrbracket_b q$ is tolerable as long as $\frac{q}{2}\left(7 + \frac{|a|}{\mathtt{R}}\right) < \frac{\mathtt{R}}{2}$. In Dilithium, this is the case since $q < 2^{23}$ and $\mathtt{R} = 2^{32}$. The benefit is that $\llbracket\rrbracket_1$ is faster than

$\lfloor\rceil$ if we have to emulate them with $\frac{\log_2 \mathtt{R}}{2} = \frac{\log_2 \mathtt{R}}{2} \times \frac{\log_2 \mathtt{R}}{2}$ low multiplication instructions[7]. Furthermore, we can also choose $\llbracket\rrbracket_1 =_b \llbracket\rrbracket$ as follows:

$$\forall r \in \mathbb{R}, {}_b\llbracket r \rrbracket := \left\lfloor \frac{a_l b_h + \frac{\sqrt{\mathtt{R}}}{2}}{\sqrt{\mathtt{R}}} \right\rfloor + \left\lfloor \frac{a_h b_l}{\sqrt{\mathtt{R}}} \right\rfloor + a_h b_h$$

and find $\left| \lfloor r \rceil - {}_b\llbracket r \rrbracket \right| \leq 2$. We call the resulting Barrett multiplication the half-approximate variant. See Table 3 for an overview of the variants of Barrett multiplication.

Table 3: Overview of the variants of Barrett multiplications. Upper bounds stand for the upper bounds of the absolute values of the results.

Name	$\llbracket\rrbracket_0$	$\llbracket\rrbracket_1$	Upper bound	Upper bound when $\lvert a\rvert \leq \frac{\mathtt{R}}{2}$
Standard	$\lfloor\rceil$	$\lfloor\rceil$	$\frac{q}{2}\left(1 + \frac{\lvert a\rvert}{\mathtt{R}}\right)$	$0.75q$
Floor	$\lfloor\rceil$	$\lfloor\rfloor$	$\frac{q}{2}\left(3 + \frac{\lvert a\rvert}{\mathtt{R}}\right)$	$1.75q$
Half-approximate	$\lfloor\rceil$	${}_b\llbracket\rrbracket$	$\frac{q}{2}\left(5 + \frac{\lvert a\rvert}{\mathtt{R}}\right)$	$2.75q$
Approximate	$\lfloor\rceil$	$\llbracket\rrbracket_b$	$\frac{q}{2}\left(7 + \frac{\lvert a\rvert}{\mathtt{R}}\right)$	$3.75q$

3.2 Transformations

We go through the transformations in this section. Section 3.2.1 goes through the multiplication-based NTT/iNTT and Section 3.2.2 goes through the fast homomorphism modulo $\mathbb{Z}_{2^k}$.

3.2.1 Multiplication-Based NTT/iNTT

This paper presents two classes of multiplication-based NTT/iNTT. The first is the NTT and iNTT mandated by Dilithium, which we call Dilithium NTT and iNTT. The second one is the 16-bit NTT/iNTT for the multiplication by the challenge polynomial in the signature generation of Dilithium.

Dilithium NTT/iNTT. For the Dilithium NTT/iNTT, since one is asked to compute a certain transformation, there are not many implementation choices. The main focus is on choosing an efficient modular multiplication. We choose Barrett multiplication for the Dilithium NTT/iNTT throughout this paper. Two variants come to our mind – the floor variant avoiding the addition of $\frac{\mathtt{R}}{2}$ and the approximate variant avoiding the computation of the low parts in the high multiplication.

Correctness of the Barrett-based NTT/iNTT. We argue that our Barrett-based NTT/iNTT is correct as there are no overflows throughout the computation. Our argument follows the traditional range analysis. Suppose the result of a modular multiplication has an absolute value bounded by θq for a positive real number θ. In Dilithium NTT, since each layer of butterflies increases the absolute values of the coefficients by at most θq and there are eight layers of butterflies, the resulting values have absolute values bounded by $(8\theta + 1)\, q$. As long as $(8\theta + 1)\, q < \frac{\mathtt{R}}{2}$, there are no overflows. This implies that any choices of $\theta \leq 31.90 < \frac{\frac{\mathtt{R}}{2q} - 1}{8}$ are sufficient for $q = 2^{23} - 2^{13} + 1$ and $\mathtt{R} = 2^{32}$. We have $\theta \leq 0.75$

[7] In the reference manual of Armv7-M [ARM21b], multiplication instructions are sometimes denoted as `w0 = w1 x w2` multiplication instructions for several combinations of bit sizes `w0` for the output, `w1` for the first input, and `w2` for the second input. We follow the same convention.

for the standard one, $\theta \leq 1.75$ for the floor variant, $\theta \leq 2.75$ for the half-approximate variant, and $\theta \leq 3.75$ for the approximate variant based on Table 3. Table 4 summarizes the upper bounds of the modulus q for various choices of θ's. Recently, there have been continuous efforts to formally verify the implementation correctness of critical subroutines, including [HLS+22] with CryptoLine and [ABB+23] with EasyCrypt. In principle, the ideas of [HLS+22] and [ABB+23] work, but we believe formal verification is out of the scope of this work.

Table 4: Relations between the quality θ of a modular multiplication and the modulus q for a 7-layer radix-2 Cooley–Tukey FFT. The approaches of modular multiplication are sorted in the increasing order of the computational efficiency on Cortex-M3.

	θ	Upper bound of q	Upper bound of $\log_2 q$
Standard	0.75	306 783 378	28.1926
Floor	1.75	143 165 576	27.0931
Half-approx.	2.75	93 368 854	26.4763
Approximate	3.75	69 273 666	26.0458

Experimental analysis. Instead of formal verification, we provide some experimental analyses for the range of the intermediate coefficients throughout the computation of Dilithium NTT with the approximate variant of Barrett multiplication. Table 5 illustrates the upper bounds of the absolute values of the intermediate coefficients layer by layer.

Table 5: Upper bounds of the absolute values of intermediate coefficients throughout an NTT computation with random inputs.

	Input	Output	Output ($\log_2$-scale)
Layer 0	8 363 648	27 234 418	24.70
Layer 1	27 234 418	41 063 339	25.30
Layer 2	41 063 339	49 941 104	25.58
Layer 3	49 941 104	61 183 326	25.87
Layer 4	61 183 326	73 057 383	26.13
Layer 5	73 057 383	75 903 955	26.18
Layer 6	75 903 955	88 663 707	26.41
Layer 7	88 663 707	93 546 600	26.48

Polynomial multiplications with 16-bit arithmetic. Recall that in Dilithium, c is a polynomial with τ ±1's and $256 - \tau$ 0's, and s_1 and s_2 are vectors of polynomials with coefficients in $\{-\eta, \dots, 0, \dots, \eta\}$, the vectors of products cs_1 and cs_2 can be computed over a sufficiently large modulus bounding the maximum possible coefficient in the results [AHKS22]. [AHKS22] applied the idea to Dilithium on Cortex-M4, and showed that Fermat number transform with the coefficient ring $\mathbb{Z}_{257}$ is the fastest approach for the security levels II and V. For the security level III, since η is too large, they chose $\mathbb{Z}_{769}$ as the coefficient ring and resorted to standard 16-bit NTT. In prior Cortex-M3 work [HAZ+24] exploiting the same idea of small NTT, they argued the benefit of code sharing between multiple parameter sets, while the whole point of their paper is about optimization for speed. We implement FNT on Cortex-M3 with butterflies from [BHK+22a] and a refined variant of [HZZ+24, HAZ+24]'s Plantard reduction based on [AMOT22] and outperform their small NTT implementation for security levels II and V in speed as we will see in Section 5.2.

3.2.2 The Fast Homomorphism Modulo Powers of Two

This section describes our polynomial multiplier for $\mathbb{Z}_{2^k}[x]/\langle x^{256}+1\rangle$ with $k = 0, 1, \ldots, 24$ operating over 32-bit registers. In the signature of Dilithium, we also need to multiply the challenge polynomial c by t_0 whose elements reside in $\left\{-2^{12}, \ldots, 0, \ldots, 2^{12}-1\right\}$. Following the previous paragraph, one can compute the product ct_0 modulo a q with $q \geq 491520 \geq 2 \cdot \tau \cdot 2^{12}$. We compute with the modulus $q = 2^{19} > 491520$.

Nussbaumer FFT in theory. Notice that $\mathbb{Z}_{2^k}[x]/\langle x^{256}+1\rangle$ admits substructures isomorphic to $\mathbb{Z}_{2^k}[y]\Big/\left\langle y^{2^{8-h}}+1\right\rangle$ via $x^{2^h} \sim y$ for some h's. We introduce the equivalence $x^{16} \sim y$ (so $y^{16} \sim -1$) and replace it with $x^{32} \sim 1$. If we regard the 2-indeterminate polynomial ring as a polynomial ring in x, we find that y is a principal 32-nd root of unity since $y^{16} = -1$ (cf. Section 2.3). Therefore, $x^{32}-1$ splits into $\prod_i \left(x - y^i\right)$. We then apply size-32 cyclic Cooley–Tukey FFT in x with y as the principal 32-nd root of unity.

Nussbaumer FFT in practice. In practice, replacing $x^{16} \sim y$ by $x^{32} \sim 1$ amounts to zero padding that is merged with the FFT computation. As for the FFT-based polynomial multiplication, we have to multiply the inverse of the transformation size 32 at the end. Since 32 is not invertible in $\mathbb{Z}_{2^k}$, we replace the coefficient ring $\mathbb{Z}_{2^k}$ by $\mathbb{Z}_{2^{k+5}}$ for adjoining divisions by 32[8]. The resulting polynomial ring is permuted due to Cooley–Tukey FFT. We summarize the transformations so far as follows:

$$\frac{\mathbb{Z}_{2^k}[x]}{\langle x^{256}+1\rangle} \cong \frac{\left(\frac{\mathbb{Z}_{2^k}[y]}{\langle y^{16}+1\rangle}\right)[x]}{\langle x^{16}-y\rangle} \hookrightarrow \frac{\left(\frac{\mathbb{Z}_{2^{k+5}}[y]}{\langle y^{16}+1\rangle}\right)[x]}{\langle x^{32}-1\rangle} \cong \prod_{i_0,\ldots,i_4=0,1} \frac{\left(\frac{\mathbb{Z}_{2^{k+5}}[y]}{\langle y^{16}+1\rangle}\right)[x]}{\left\langle x - y^{\sum_j i_j 2^j}\right\rangle}.$$

Analyzing the number of multiplications. For simplicity, we assume $n \geq 2$ is a power of two with exponent a power of two. In the literature, a size-n Nussbaumer FFT for $R[x]/\langle x^n+1\rangle$ requires $\Theta\left(n \lg n \max\left(\lg\lg n, 1\right)\right)$ additions/subtractions and results in $\frac{n}{2}\lg n$ size-2 polynomials. If we multiply two size-n polynomials with Nussbaumer FFT, we need $\Theta\left(n \lg n \max\left(\lg\lg n, 1\right)\right)$ operations in the coefficient ring. In practice, we need to revise the analysis of the number of multiplications for a concrete analysis. Let's say we recurse until the problem size is smaller than or equal to a platform-dependent power-of-two constant $t \geq 2$ with exponent a power of two and switch to asymptotically slower approaches, such as the schoolbook, Karatsuba, and Toom–Cook, with t^α operations where $1 < \alpha < 2$ is a constant. The switches to asymptotically slower approaches are typically employed in practice [ACC+21, CHK+21, NG21, ACC+22, BBCT22, AHY22, BHK+22b, BMK+22, Hwa24a]. We revise the number of multiplications in Cooley–Tukey and Nussbaumer FFTs for polynomial multiplications as follows:

- Cooley–Tukey FFT: For the transformation, we need $\frac{n \lg n}{2 \lg t}$ multiplications for each, and there are three transformations, resulting in $\frac{3n \lg n}{2 \lg t}$ multiplications. Furthermore, we also have $\frac{n}{t}$ size-t polynomial multiplications with each requiring t^α multiplications. In total, we need $\frac{3n \lg n}{2 \lg t} + nt^{\alpha-1}$ multiplications with Cooley–Tukey FFT.

[8] By correctness, the last division by 32 is applied to the 32-multiple of the desired result. Replacing $\mathbb{Z}_{2^k}$ by $\mathbb{Z}_{2^{k+5}}$ suffices for computing the 32-multiple. This can be justified by the localization of $\mathbb{Z}$-algebra [Jac12, Section 7] at non-zero elements.

- Nussbaumer FFT: We don't need multiplications for the transformation, and we have $\frac{n \lg n}{t \lg t}$ size-t polynomial multiplications with each requiring t^α multiplications. Therefore, we need $\frac{nt^{\alpha-1} \lg n}{\lg t}$ multiplications with Nussbaumer FFT.

We compare the factors of the dominating term $n \lg n$: we have $\frac{3}{2 \lg t}$ in Cooley–Tukey and $\frac{t^{\alpha-1}}{\lg t}$ in Nussbaumer. See Table 6 for a summary. Since t is typically between 4 to 16 by experiments [CHK+21, BBCT22][9], Nussbaumer amounts to a much larger number of multiplications. We will later see the numerical justification of our revision in Section 5.2.

Table 6: Overview of the arithmetic cost of Cooley–Tukey and Nussbaumer FFTs for multiplying two size-n polynomials with the threshold t. Transformation cost refers to the number of corresponding arithmetic for transforming the large-dimensional polynomial multiplication into several small-dimensional polynomial multiplications. There are three rows in the transformation cost: (i) the number of additions/subtractions, (ii) the number of multiplications, and (iii) the number of small-dimensional polynomial multiplications after the transformation. Polynomial multiplication cost refers to the number of corresponding arithmetic implementing the large-dimensional polynomial multiplication. We only present the number of multiplications and the dominating term for the polynomial multiplication cost.

	Cooley-Tukey	Nussbaumer
Transformation cost		
# of add./sub.	$\frac{1}{2 \lg t} \cdot n \lg n$	0
# of mul.	$\frac{1}{\lg t} \cdot n \lg n$	$\Theta\left(n \lg n \max\left(\lg \log_t n, 1\right)\right)$
# of small dim. polymul.	$\frac{n}{t}$	$\frac{1}{t \lg t} \cdot n \lg n$
Polynomial multiplication cost		
# of mul.	$\frac{3n \lg n}{2 \lg t} + nt^{\alpha-1}$	$\frac{nt^{\alpha-1} \lg n}{\lg t}$
Dominating term	$\frac{3n \lg n}{2 \lg t}$	$\frac{nt^{\alpha-1} \lg n}{\lg t}$

The remaining computing task are 32 polynomial multiplications in $\mathbb{Z}_{2^{k+5}}[y] / \left\langle y^{16}+1 \right\rangle$. We implement it with a Toeplitz matrix-vector product built upon Toom-4.

Toeplitz matrix-vector product from Toom-4 in theory. For simplicity, we describe the Toom-4 [Too63] corresponding to the 4-way split of TMVP used in this work. Suppose we want to multiply two size-4 polynomials $\sum_{i=0}^{3} a_i x^i$ and $\sum_{i=0}^{3} b_i x^i$ in $R[x]$. We apply Toom-4 with the point set $\left\{0, \pm 1, \pm 2, \frac{1}{2}, \infty\right\}$. This amounts to applying the following

[9] One should not confuse the determination of t in polynomial multiplication with the designs of Dilithium and Kyber NTT/iNTT. Kyber [ABD+20b] is a key encapsulation mechanism selected by the NIST Post-Quantum Cryptography Standardization. In Dilithium and Kyber NTTs/iNTTs, the designers chose $t = 1, 2$ and mandated one of the operands to be sampled from the transformed domain. Therefore, they are multiplying polynomials in a transformed domain instead of a large-dimensional polynomial ring as in Nussbaumer.

matrix to all the operands with the standard basis[10]:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 2 & 4 & 8 \\ 1 & -2 & 4 & -8 \\ 8 & 4 & 2 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

After 7 multiplications in R, we apply the following inversion map to the 7-tuple:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 1 & 2 & 4 & 8 & 16 & 32 & 64 \\ 1 & -2 & 4 & -8 & 16 & -32 & 64 \\ 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}^{-1}.$$

This gives us the desired product $\left(\sum_{i=0}^{3} a_i x^i\right)\left(\sum_{i=0}^{3} b_i x^i\right)$. The 4-way Toeplitz matrix-vector product used in this work is obtained via a series of dualizations from the module-theoretic point of view. See [CCHY24, Section 4.5] for a constructive correctness proof.

Toeplitz matrix-vector product from Toom-4 in practice. In practice, we need to figure out how to adjoin divisions by powers of two while applying the transformation matrix. Conceptually, we extract all divisions by powers of two, move them to the end of the computation, and implement them with shifts. By the correctness of the Toeplitz matrix-vector product, we will always shift out zeros at the end of the computation. One can show that the requirement of divisions by powers of two is the same as in Toom-4. Since Toom-4 with the point set $\left\{0, \pm 1, \pm 2, \frac{1}{2}, \infty\right\}$ requires the divisions by 8, we replace the coefficient ring $\mathbb{Z}_{2^{k+5}}$ by $\mathbb{Z}_{2^{k+8}}$. This explains why our polynomial multiplier only works for $k = 0, 1, \ldots, 24$ while operating entirely on 32-bit registers.

4 Implementations

4.1 Barrett Multiplications

We implement two variants of Barrett multiplications: the first one is the constant-time approximate variant of Barrett multiplication; the second one is the variable-time floor variant of Barrett multiplication. See Table 7 for an overview of the total instruction timings.

Constant-time Barrett multiplication. For the constant-time Barrett multiplication, we compute $ab - \left\llbracket \frac{a \left\lfloor \frac{2^{32} b}{q} \right\rceil}{2^{32}} \right\rrbracket_b q$ as a representative of $ab \bmod q$ where $\left\lfloor \frac{2^{32} b}{q} \right\rceil$ is precomputed. We first compute $\left\llbracket \frac{a \left\lfloor \frac{2^{32} b}{q} \right\rceil}{2^{32}} \right\rrbracket_b$ with Algorithm 6 and then compute the difference of the products ab and $\left\llbracket \frac{a \left\lfloor \frac{2^{32} b}{q} \right\rceil}{2^{32}} \right\rrbracket_b q$. See Algorithm 7 for an illustration. Since our constant-time

[10] The standard basis representing size-n polynomials is $\left\{1, x, x^2, \ldots, x^{n-1}\right\}$, and the standard basis for a product of polynomial rings is their juxtaposition.

Table 7: Overview of multiplication operations with 32-bit input values on Cortex-M3. The cycles are obtained by summing up the instruction timings from the manual [ARM10a].

Plain multiplication		
Multiplication operation	Work	Cycle
Long (variable-time)	[ARM10a]	3 − 7
Long (constant-time, non-generic)	[GKS21]	11
Long (constant-time)	[GKS21]	12
Modular multiplication (Constant-Time)		
Montgomery multiplication	[GKS21]	23
Barrett multiplication (approximate)	This work	12 (1.92)
Modular multiplication (Variable-Time)		
Montgomery multiplication	[GKS21]	9 − 16
Barrett multiplication (floor)	This work	6 − 8 (1.13 − 2.67)

Barrett multiplication performs comparably to the constant-time long multiplication, it outperforms any modular multiplication calling a constant-time long multiplication followed by a reduction subroutine with non-negligible cost on Cortex-M3. This explains why our constant-time Barrett multiplication performs 1.92 times faster than the constant-time Montgomery multiplication by [GKS21] with the same input constraints (we assume the inputs are 32-bit numbers since such comparisons are more orthogonal for 32-bit arithmetic whereas [GKS21] assumed the inputs are 16-bit numbers while replacing the memory load operations for words with doubly many load operations for halfwards).

Algorithm 6 Implementation of `mulhi_split` on Cortex-M3.

Inputs: `alo` $= a_l$, `ahi` $= a_h$, `blo` $= b_l$, `bhi` $= b_h$.
Outputs: `acchi` $= a_h b_h + \left\lfloor \frac{a_l b_h}{2^{16}} \right\rfloor + \left\lfloor \frac{a_h b_l}{2^{16}} \right\rfloor$.

```
mul  acchi,   ahi,    bhi                      ▷ acchi $= a_h b_h$.
mul accmid,   alo,    bhi                      ▷ accmid $= a_l b_h$.
add  acchi, acchi, accmid, asr #16             ▷ acchi $= a_h b_h + \left\lfloor \frac{a_l b_h}{2^{16}} \right\rfloor$.
mul accmid,   ahi,    blo                      ▷ accmid $= a_h b_l$.
add  acchi, acchi, accmid, asr #16             ▷ acchi $= a_h b_h + \left\lfloor \frac{a_l b_h}{2^{16}} \right\rfloor + \left\lfloor \frac{a_h b_l}{2^{16}} \right\rfloor$.
```

Variable-time Barrett multiplication. For the variable-time Barrett multiplication, we compute $ab - \left\lfloor \frac{a \left\lfloor \frac{2^{32} b}{q} \right\rceil}{2^{32}} \right\rfloor q$ as shown in Algorithm 8. Compared to the assembly-optimized variable-time Montgomery multiplication as shown in Algorithm 9, Barrett multiplication turns the `smlal` into an `mls`. Since `smlal` takes 5 to 8 cycles and `mls` takes only two cycles, Barrett multiplication is obviously faster. Overall, our variable-time Barrett multiplication is 1.13 − 2.67 times faster than the variable-time Montgomery multiplication by [GKS21].

4.2 Dilithium NTT/iNTT

We apply our generalized Barrett multiplication to Dilithium NTT/iNTT. Such NTT/iNTT are required in the matrix-vector multiplication and are not covered by [HAZ+24]. To be specific, Lines 5 and 6 in Algorithm 3, Lines 2 and 5 in Algorithm 4, and Line 6 in Algorithm 5.

Algorithm 7 Constant-time Barrett multiplication with 32-bit inputs on Cortex-M3.

```
Inputs: $\texttt{a} = a, \texttt{b} = b, \texttt{bp} = \left\lfloor \frac{2^{32}b}{q} \right\rceil$.
Outputs: $\texttt{c} = ab - \left[\!\left[ \frac{a \left\lfloor \frac{2^{32}b}{q} \right\rceil}{2^{32}} \right]\!\right]_b q$.
mul         c,   a,   b                    ▷ $\texttt{c} = ab \bmod {}^{\pm}2^{32}$.
ubfx       t0,   a,  #0, #16
asr         a,   a, #16                    ▷ $\texttt{t0} + \texttt{a} \cdot 2^{16} = a$.
ubfx      blo,  bp,  #0, #16
asr       bhi,  bp, #16                    ▷ $\texttt{blo} + \texttt{bhi} \cdot 2^{16} = \left\lfloor \frac{2^{32}b}{q} \right\rceil$.
                                           ▷ This splitting can be merged with memory operations.
mulhi_split t1,  a, bhi, t0, blo,  t2      ▷ $\texttt{t1} = \left[\!\left[ \frac{a \left\lfloor \frac{2^{32}b}{q} \right\rceil}{2^{32}} \right]\!\right]_b$.
mls         c, t1,   $q$,   c              ▷ $\texttt{c} = ab - \left[\!\left[ \frac{a \left\lfloor \frac{2^{32}b}{q} \right\rceil}{2^{32}} \right]\!\right]_b q$.
```

Algorithm 8 Variable-time Barrett multiplication on Cortex-M3.

```
Inputs: $\texttt{a} = a, \texttt{b} = b, \texttt{bhi} = \left\lfloor \frac{2^{32}b}{q} \right\rceil$.
Outputs: $\texttt{c} = ab - \left\lfloor \frac{a \left\lfloor \frac{2^{32}b}{q} \right\rceil}{2^{32}} \right\rfloor q$.
smull lo, hi, a, bhi                       ▷ $\texttt{lo} + \texttt{hi} \cdot 2^{32} = a \left\lfloor \frac{2^{32}b}{q} \right\rceil$.
mul    c,  a, b                            ▷ $\texttt{c} = ab \bmod {}^{\pm}2^{32}$.
mls    c, hi, $q$, c                       ▷ $\texttt{c} = ab - \left\lfloor \frac{a \left\lfloor \frac{2^{32}b}{q} \right\rceil}{2^{32}} \right\rfloor q$.
```

Choices of Barrett multiplications for constant/variable-time NTT/iNTT. For the matrix-vector multiplications of Dilithium, one should apply constant-time computation to the ones in key generation and signature generation. As for the matrix-vector multiplication in the signature verification, one can compute it with the fastest approach without any side-channel worrisome since verification is public under the context of digital signature. On Cortex-M3, we compute the matrix-vector multiplications in key generation and signature generation with constant-time NTT/iNTT based on the approximate variant of Barrett multiplication and the one in the signature verification with the fastest approach – variable-time NTT/iNTT based on the floor variant of Barrett multiplication. See Table 8 for a summary of the constant-time requirement and the chosen modular multiplications in the NTTs/iNTTs of the matrix-vector multiplications.

Table 8: Constant-time requirements and modular multiplications in the NTTs/iNTTs of matrix-vector multiplications of key generation, signature generation, and signature verification.

	Constant-timeness requir.	Approach (Cortex-M3)
Key generation	✓	Constant-time approx. Barrett
Signature generation	✓	Constant-time approx. Barrett
Signature verification	✗	Variable-time floor Barrett

Algorithm 9 Variable-time Montgomery multiplication on Cortex-M3 [GKS21, ACC⁺21]. [GKS21] and [ACC⁺21] independently proposed the same computation. [GKS21] applied the idea to Cortex-M3 and Cortex-M4 and [ACC⁺21] applied the idea to Cortex-M4.

```
Inputs: a = $a$, b = $b$.
Outputs: hi = $\frac{ab+\left(-abq^{-1} \bmod {}^{\pm}2^{32}\right)q}{2^{32}}$.
smull lo, hi, a, b                                ▷ lo + hi $\cdot 2^{32} = ab$.
mul   lo, lo, $-q^{-1} \bmod {}^{\pm}2^{32}$      ▷ lo $= -abq^{-1} \bmod {}^{\pm}2^{32}$.
smlal lo, hi, lo, $q$                             ▷ hi $= \frac{ab+\left(-abq^{-1} \bmod {}^{\pm}2^{32}\right)q}{2^{32}}$.
```

Analyzing the register pressure of layer-merging. Layer-merging is a common memory optimization strategy for software implementations. Conceptually, we load a series of coefficients defining multiple layers of isomorphisms, compute the isomorphisms, and store the results in memory. For the radix-2 NTT/iNTT, the goal is to compute l layers of radix-2 butterflies. We formally analyze the register pressure enabling an l-layer merging for the constant-time and variable-time NTT/iNTT. For an l-layer merge in an NTT/iNTT, we must load 2^l coefficients and $2^l - 1$ twiddle factors. For the variable-time radix-2 NTT/iNTT, since we need 2^l registers for the coefficients and $2^l - 1$ registers for the twiddle factors, we can merge up to two layers. On the other hand, we need 2^l registers for the coefficients and $3\left(2^l - 1\right)$ registers for the twiddle factors in our constant-time radix-2 NTT. Therefore, there is no layer-merging in our constant-time radix-2 NTT. As for the iNTT, we apply a similar layer-merging technique from [ACC⁺22] by observing that some twiddle factors are 1's.

4.3 Fast Homomorphism Modulo Powers of Two

For our fast homomorphism for $\mathbb{Z}_{2^k}[x]/\left\langle x^{256}+1\right\rangle$, we briefly outline the implementation challenges for the Nussbaumer implementing $\mathbb{Z}_{2^k}[x]/\left\langle x^{256}+1\right\rangle \hookrightarrow \mathcal{R}'[x]/\left\langle x^{32}-1\right\rangle \cong (\mathcal{R}')^{32}$ where $\mathcal{R}' = \mathbb{Z}_{2^{k+5}}[y]/\left\langle y^{16}+1\right\rangle$. As for the TMVP for $\mathcal{R}'$, we find it straightforward for implementation, so we skip the description.

Layer merging in Nussbaumer. Recall that in Nussbaumer, twiddle factor multiplications are negacyclic shifts, and when we move to the next layer, there are doubly many negacyclic shifts where two registers are involved in each butterfly. This implies a factor of 4 blow-up, and we need 2^{2l-1} registers for an l-layer merge. Therefore, we can only merge at most two layers on Cortex-M3.

Optimizing the layer merging during pre- and post-processing in Nussbaumer. We further implement the following optimizations to reduce the memory access during the replacement of $x^{16} \sim y$ by $x^{32} \sim 1$. We skip the explicit replacement of relations and the initial butterflies, and modify the memory load in the follow-up butterflies accordingly. For the converse replacement (replacing $x^{32} \sim 1$ by $x^{16} \sim y$), we also merge it with the last series of butterflies.

Memory consumption. Finally, we outline the memory consumption of the fast homomorphism modulo powers of two. In the beginning, we map each polynomial in $\mathbb{Z}_{2^k}[x]/\left\langle x^{256}+1\right\rangle$ to 32 polynomials in $\mathbb{Z}_{2^{k+5}}[y]/\left\langle y^{16}+1\right\rangle$ with Nussbaumer. For each polynomial in $\mathbb{Z}_{2^{k+5}}[y]/\left\langle y^{16}+1\right\rangle$, we map it to seven 4×4 Toeplitz matrices over $\mathbb{Z}_{2^{k+8}}$ for one of the operands and seven size-4 vectors over $\mathbb{Z}_{2^{k+8}}$ for the other operand. For each Toeplitz matrix, we only need to store the first row and the first column, but we allocate 8 words for smooth engineering [IKPC20]. We need $4 \cdot 8 \cdot 7 \cdot 32 = 7168$ bytes for

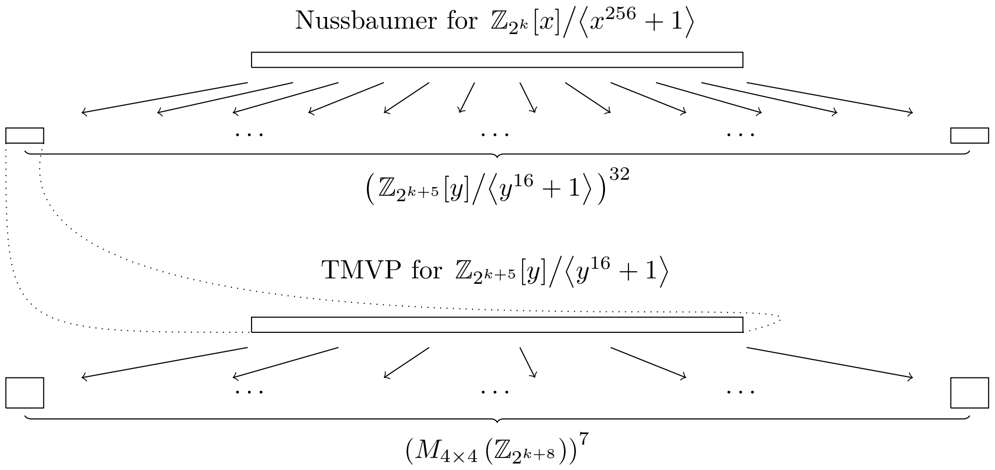

Nussbaumer for $\mathbb{Z}_{2^k}[x]/\langle x^{256}+1\rangle$
$\left(\mathbb{Z}_{2^{k+5}}[y]/\langle y^{16}+1\rangle\right)^{32}$
TMVP for $\mathbb{Z}_{2^{k+5}}[y]/\langle y^{16}+1\rangle$
$\left(M_{4\times 4}\left(\mathbb{Z}_{2^{k+8}}\right)\right)^{7}$

Figure 2: Overview of our fast homomorphism modulo powers of two. There are two phases: (i) Nussbaumer for $\mathbb{Z}_{2^k}[x]/\langle x^{256}+1\rangle$ and (ii) TMVP for $\mathbb{Z}_{2^{k+5}}[y]/\langle y^{16}+1\rangle$. Nussbaumer maps $\mathbb{Z}_{2^k}[x]/\langle x^{256}+1\rangle$ to $\left(\mathbb{Z}_{2^{k+5}}[y]/\langle y^{16}+1\rangle\right)^{32}$. As for TMVP for $\mathbb{Z}_{2^{k+5}}[y]/\langle y^{16}+1\rangle$, we illustrate the matrix part – each polynomial in $\mathbb{Z}_{2^{k+5}}[y]/\langle y^{16}+1\rangle$ is expanded into seven 4×4 Toeplitz matrices. For a Toeplitz matrix, we only store the first row and column explicitly [IKPC20].

the Toeplitz part and $4\cdot 4\cdot 7\cdot 32 = 3584$ bytes for the vector part. See Figure 2 for an illustration for the Toeplitz part. In addition, we also need two buffers of 512 words each for the Nussbaumer while saving memory operations with out-of-place computation. In total, we need $7168 + 3584 + 4\cdot 512\cdot 2 = 14848$ bytes of memory where the buffers for Nussbaumer are shared while multiplying two polynomials with our fast homomorphism modulo powers of two.

5 Results

5.1 Benchmarking Environment

We benchmark our Armv7-M implementations on a `nucleo-f207zg` board containing a `stm32f207zg` core with 128 KiB of SRAM and 1 MB of flash memory. According to [STM20, Sections 3.2 and 3.6], `stm32f207zg` provides access to SRAM and flash memory with 0 wait state up to the frequency 120 MHz. Nevertheless, we follow the literature [ACC+22] and benchmark at a frequency 30 MHz for consistency. We compile our code with the cross-compiler `arm-none-eabi-gcc` version `10.3.1`. For fair comparisons, we plug in the improved Keccak permutation by [HAZ+24] and re-bench the implementations from the literature with this new Keccak permutation.

5.2 Performance of Polynomial Multiplications

5.2.1 Dilithium NTT/iNTT

For the Dilithium NTT/iNTT, our constant-time Barrett-based NTT and iNTT are 1.51× and 1.38× faster than prior art with constant-time Montgomery multiplications, and our variable-time Barrett-based NTT and iNTT are 1.21× and 1.10× faster than prior art with variable-time Montgomery multiplications. See Table 9 for a summary.

Table 9: Performance numbers of Dilithium NTT/iNTT on Cortex-M3.

	Constant-time		Variable-time	
	[GKS21]	This work	[GKS21]	This work
NTT	33 025	21 876 (1.51)	19 347	15 985 (1.21)
iNTT	36 609	26 524 (1.38)	21 006	19 067 (1.10)

5.2.2 Polynomial Multiplications with 16-bit Arithmetic Precision

For the 16-bit arithmetic, our NTT, base multiplication, and iNTT are 1.08×, 1.41×, 1.11× faster than prior art by [HAZ+24] with no surprises. See Table 10 for a summary.

Table 10: Performance cycles of polynomial multiplications with 16-bit arithmetic precision on Cortex-M3. The numbers of [ACC+22] in this table were reported in one of the authors' master thesis [Hwa22, Table 9.10]

Work	[ACC+22]	[HAZ+24]	This work
Coefficient ring	$\mathbb{Z}_{3329}$	$\mathbb{Z}_{769}$	$\mathbb{Z}_{257}$
Approach	Montgomery	Plantard	FNT
NTT	8 688 (0.90)	7 830	7 252 (1.08)
Mul.	5 987 (0.67)	3 989	2 835 (1.41)
iNTT	9 553 (0.89)	8 543	7 667 (1.11)

5.2.3 Polynomial Multiplications with 32-bit Arithmetic Precision

For polynomial multiplications with 32-bit arithmetic precision, our polynomial multiplier built upon Nussbaumer FFT and TMVP from Toom-4 is more performant than existing works. We have roughly 1.9 times faster module homomorphism for one of the operands and the interpolation at the expense of slightly slower module homomorphism for the other operand and a slower bilinear map compared to [HAZ+24]. See Table 11 for a summary. We demonstrate the impact of the choices of coefficient rings with numerical evidence from the literature [BBCT22, Hwa24a, HAZ+24] and this work as follows.

Nussbaumer modulo $\mathbb{Z}_q$ for an odd q. Section 3.2.2 explains that Nussbaumer intrinsically requires much more multiplications not reflected in the asymptotic analysis of the runtime than Cooley–Tukey. We compare the Schönhage[11]+Nussbaumer approach by [BBCT22] to the multiplication-based NTT approach by [Hwa24a] for multiplying polynomials in $\mathbb{Z}_{4591}[x]/\langle x^{761} - x - 1\rangle$ of NTRU Prime [BBC+20]. Both approaches are quite complicated, so we skip the descriptions and focus on the number of small-dimensional polynomial multiplications after the transformations. Both approaches amount to size-8 polynomial multiplications over $\mathbb{Z}_{4591}$ and are implemented with AVX2 on Haswell processors with the modular multiplication by [Sei18]. The Schönhage+Nussbaumer by [BBCT22] results in 768 size-8 polynomial multiplications, and the multiplication-based NTT by [Hwa24a] results in 192 size-8 polynomial multiplications. Since each modular multiplication in $\mathbb{Z}_{4591}$ amounts to 3 times the cycles compared to the plain multiplication for the most critical pipeline on Haswell, we would expect a large performance penalty while extending

[11] Schönhage [Sch77] is a similar approach crafting the roots of unity.

Table 11: Performance cycles of polynomial multiplications with 32-bit arithmetic precision on Cortex-M3. The total cycles of polynomial multiplications are obtained by summing up all the rows in the building block, and other rows are obtained by benchmarking.

Work	[ACC+22]	[HAZ+24]	This work
Coefficient ring	$\prod_{i=0,1} \mathbb{Z}_{q_i}$	$\prod_{i=0,1} \mathbb{Z}_{q_i}$	$\mathbb{Z}_{2^{\leq 24}}$
Approach	Montgomery	Plantard	Nussbaumer
Building block			
NTT/Hom-M	16 774 (0.93)	15 626	15 820 (0.99)
NTT/Hom-V	16 774 (0.93)	15 626	8 259 (1.89)
Mul./BiHom	11 933 (0.68)	8 061	11 217 (0.72)
iNTT/Hom-I	23 721 (0.88)	20 772	10 960 (1.90)
Polynomial multiplication			
Total cycles	69 202 (0.87)	60 085	46 256 (1.30)
Ratio of mul./BiHom over total cycles	17.24%	13.42%	24.25%
Memory (bytes)	1 536	2 048*	14 848

* In [HAZ+24, Section 4.2.3], the authors claimed to achieve an improvement of stack usage to one-third of Algorithm 5 in [ACC+22]. Their statement is invalid. Firstly, Algorithm 5 of [ACC+22] computes the masked product with 48-bit precision, and a scientifically valid way is to compare to Algorithm 4 of [ACC+22], which computes the unmasked product with 32-bit precision, the computation what [HAZ+24, Section 4.2.3] refers to. Secondly, they claimed that 1024 bytes are used by the polynomial multiplication by sharing the memory usage of one of the operands with something else and removing its memory attribution, whereas [ACC+22] reported memory usage with all the operands included and also reused the memory for something else in the implementation. We follow [ACC+22]'s comparison methodology.

the polynomial rings with Nussbaumer and Schönhage. In fact, the cycles of the small-dimensional polynomial multiplications in the Schönhage+Nussbaumer FFT approach by [BBCT22] takes 55.24% of the polynomial multiplication cycles, whereas the cycles of the small-dimensional polynomial multiplications in the multiplication-based NTT approach by [Hwa24a] takes only 23.98% of the polynomial multiplication cycles, and the majority of the improvement cycles of [Hwa24a] over [BBCT22] comes from the reduction of small-dimensional polynomial multiplications. In summary, when multiplications in the coefficient ring are slow, Nussbaumer FFT is not a good choice.

Nussbaumer modulo $\mathbb{Z}_{2^k}$. Things become quite different when the coefficient ring takes the form $\mathbb{Z}_{2^k}$. For the Nussbaumer approach, the small-dimensional polynomial multiplication takes only 24.25% of the overall polynomial multiplication. As for the multiplication-based NTT approach, since the $\mathbb{Z}_{2^k}$ does not straightforwardly support an NTT, one has to resort to the multi-moduli approach based on the residue number system when there is no efficient 32-bit modular multiplication, such as Cortex-M3 [ACC+22, HAZ+24] and AVX2 [CHK+21]. See Table 12 for a summary of the performance impact of the choices of coefficient rings in Nussbaumer.

Notes on the engineering effort. For the engineering effort of the Nussbaumer and Toeplitz-TC over $\mathbb{Z}_{2^k}$, we developed the assembly programs with the aid of the artifact in C associated to [Hwa24b]. According to our development log, only six days were spent from parameter selection to hand-written assembly-optimized implementation without any scripting support. We later developed the C programs for the transformations to facilitate future development. Please find the C programs in the artifact.

Table 12: Summary of the performance impact of the choices of coefficient rings in Nussbaumer and similar approaches versus other approaches. We extracted the small-dimensional polynomial multiplications in the AVX2 programs by [BBCT22, Hwa24a] and benchmarked with the same setting of [Hwa24a] on the same Haswell processor after contacting the author of [Hwa24a].

	Mul./BiHom	Total	Mul./BiHom over Total
Polymul. in $\mathbb{Z}_{4591}/\langle x^{761} - x - 1\rangle$ of NTRU Prime with AVX2 on Haswell			
Schönhage/Nussbaumer, [BBCT22]	12 960	23 460	55.24%
Mul.-based, [Hwa24a]	2 958	12 336	23.98%
Polymul. of *one of the components* of ct_0 for Dilithium on Cortex-M3			
Mul.-based, [HAZ+24] ($\mathbb{Z}_{3329} \times \mathbb{Z}_{7681}$)	8 061	60 085	13.42%
Nussbaumer ($\mathbb{Z}_{2^{19}}$), this work	11 217	46 256	24.25%

5.3 Performance of Core Polynomial Arithmetic

Matrix-vector multiplications. In Dilithium, the most time-consuming polynomial arithmetic is the matrix-vector multiplications in key generation, signature generation, and signature verification. In the key generation and signature generation, we use constant-time arithmetic; in the signature verification, we use variable-time arithmetic as mentioned in Section 4.2 and Table 8. We plug in our Barrett-based constant/variable-time NTT/iNTT accordingly. We additionally follow a similar idea of accumulating several products prior to applying modular reductions in the literature [CHK+21, Section 3.1]. Compared to the state-of-the-art [GKS21], our variable-time matrix-vector multiplications are $1.36 - 1.41$ times faster, and our constant-time matrix-vector multiplications are $1.71 - 1.77$ times faster (cf. Table 13).

Table 13: Performance cycles of the matrix-vector multiplications for Dilithium on Cortex-M3. The performance numbers of prior work [GKS21] are projections of the lower bounds based on [GKS21, Table 2].

Security level	Variable-time		Constant-time	
	[GKS21]	This work	[GKS21]	This work
II	240k	176k (1.36)	414k	242k (1.71)
III	370k	267k (1.39)	639k	371k (1.72)
V	578k	411k (1.41)	999k	566k (1.77)

Multiplications with the challenge polynomial. The second most time-consuming polynomial arithmetic is cs_1, cs_2, and ct_0 in the rejection loop of signature generation. We apply our FNT to cs_1 and cs_2 in Dilithium II and V, and our implementations are 1.20× faster than [HAZ+24]. We also apply our Nussbaumer with Toeplitz-TC to ct_0 in Dilithium II. In theory, the idea also works for Dilithium III and V, but we can't apply them due to memory usage. Our ct_0 in Dilithium II is 1.31× faster than [HAZ+24]. See Table 14 for a summary.

Table 14: Performance cycles of cs_1, cs_2, ct_0 in the rejection loop of signature generation in Dilithium on Cortex-M3 where the transformation cost of c is excluded since it is shared among several computations. The numbers of [HAZ$^+$24] are projections based on Tables 10 and 11 since their benchmark setup is flawed ([HAZ$^+$24] counted the transformation cost of c twice while reporting numbers of cs_1 and cs_2, but this should be counted only once even if the cost of c is counted). The memory usage of the full signature generation is also included where **S** stands for the signature generation of Dilithium. In `dilithium2`, the memory overhead comes from (i) accumulating long products in the matrix-vector multiplication, (ii) applying FNT with elements stored as 32-bit elements to cs_1 and cs_2, and (iii) Nussbaumer to ct_0. In `dilithium5`, the memory overhead comes from (ii) and (iii).

	`dilithium2`		`dilithium5`	
	[HAZ$^+$24]	This work	[HAZ$^+$24]	This work
cs_1	50 128	41 672 (1.20)	87 724	72 891 (1.20)
cs_2	50 128	41 672 (1.20)	100 256	83 296 (1.20)
ct_0	115 332	87 969 (1.31)	230 664	-
S memory (bytes)	44 972	76 876	57 740	67 436

5.4 Performance of Dilithium

We compare the performance of Dilithium on Cortex-M3 by [HAZ$^+$24]. Before going through the numbers, we would like to point out misuses of variable-time Dilithium NTT in the signature generation of `dilithium5` by [HAZ$^+$24] – they applied the variable-time Dilithium NTT by [GKS21] to the vector operand $\hat{y} = \mathtt{NTT}(y)$ of the matrix-vector multiplication $\hat{A}\hat{y}$ in Line 6 of Algorithm 5. We believe this is an oversight since they only applied the variable-time Dilithium NTT to the signature generation of `dilithium5`, and it doesn't occur in the signature generations of `dilithium2` and `dilithium3`. We replace the variable-time Dilithium NTT by [GKS21] in the signature generation of [HAZ$^+$24]'s work with the constant-time Dilithium NTT by [GKS21] and benchmark their work in this paper.

For the key generation, we reduce the cycles by 12.70%, 9.34%, and 9.65% for the security levels II, III, and V, respectively. For the signature generation, we reduce the cycles by 18.42%, 12.34%, and 8.34% for the security levels II, III, and V, respectively. As for the signature verification, we reduce the cycles by 5.57%, 5.15%, and 5.08% for the security levels II, III, and V, respectively. See Table 15 for a summary of the performance cycles. In addition, we also summarize the stack usage in Table 16. The increases in stack usage come from the optimization of the accumulation of the products in the matrix-vector multiplication, Fermat number transform for cs_1 and cs_2, and Nussbaumer over $\mathbb{Z}_{2^k}$ for ct_0 in `dilithium2`.

6 Discussions

Applications to other Cortex processors. In this paper, we explore the impact of the absences of powerful multiplication instructions on Cortex-M3 for Dilithium. For 32-bit modular multiplications, our findings immediately apply to Cortex processors with similar characteristics such as Cortex-M0, Cortex-M0+, and Cortex-M23 – there is only `muls` instruction for multiplication on these processors. Therefore, one has to resort to software emulation for the long multiplications in Montgomery multiplication. On the other hand, our Barrett multiplication avoids this and will be faster than Montgomery multiplication. For

Table 15: Performance cycles of Dilithium on Cortex-M3. **K** stands for key generation, **S** stands for signature generation, and **V** stands for signature verification.

NIST security level	Work	Operation					
		K		**S**		**V**	
		Cycles	Hash	Cycles	Hash	Cycles	Hash
II	[HAZ+24]	1 764k	1 185k	5 617k	2 173k	1 597k	1 065k
	This work	1 540k	1 123k	4 554k	2 173k	1 508k	1 065k
III	[HAZ+24]	2 944k	2 034k	7 448k	3 399k	2 659k	1 872k
	This work	2 669k	2 034k	6 529k	3 399k	2 522k	1 872k
V	[HAZ+24]	4 923k	3 510k	20 180k	14 195k	4 525k	3 347k
	This work	4 448k	3 510k	18 383k	14 195k	4 295k	3 347k

Table 16: Stack usage in bytes of Dilithium on Cortex-M3. **K** stands for key generation, **S** stands for signature generation, and **V** stands for signature verification.

NIST security level	Work	Operation		
		K	**S**	**V**
		Stack	Stack	Stack
II	[HAZ+24]	8 764	44 972	36 404
	This work	11 804	76 876	36 356
III	[HAZ+24]	9 780	69 020	57 900
	This work	13 844	69 996	57 852
V	[HAZ+24]	11 828	57 740	42 780
	This work	17 940	67 436	42 732

Cortex-M processors implementing powerful scalar long multiplications such as Cortex-M4 and Cortex-M7, the floor variant of Barrett multiplication and Montgomery multiplication perform the same. For high-end processors implementing Armv8.1-M MVE [Mar20] and Armv7-A/Armv8-A Neon [ARM12, ARM21a] vector instruction sets/extensions, one should apply the Barrett multiplication by [BHK+22b] since MVE and Neon implement powerful vector multiplication instructions in the context of Dilithium[12]. See Table 17 for an overview of recommended constant-time 32-bit modular multiplications for Cortex-M processors. This work generalizes the Barrett multiplication by [BHK+22b] and applies the resulting Barrett multiplication to platforms without powerful multiplication instructions.

Modular multiplications on other architectures and more. Generally speaking, our generalization of Barrett multiplication scales well when the arithmetic precision is $\frac{1}{2}$ to $\frac{1}{4}$ of the target precision of shape-independent modular multiplication. For the case of 32-bit modular multiplications, this covers a wide range of legacy machines such 8-bit AVR (cf. Appendix C), 16-bit MSP430, and some 32-bit architectures such as SPARC V7. For the 64-bit modular multiplications, we demonstrate the benefit of Barrett multiplication over Montgomery multiplication with the Armv7E-M adopted by Cortex-M4 and Cortex-M7. Algorithm 10 illustrates 64-bit Barrett multiplication and Algorithm 11 illustrates the 64-bit Montgomery multiplication. Algorithm 10 can be implemented straightforwardly and

[12] Following [BHK+22b], we can implement the vectorized 32-bit Barrett multiplication with instructions `sqrdmulh`, `mul`, and `mla` in Neon. As for MVE, we can implement the vectorized 32-bit Barrett multiplication with instructions `vqrdmulh.s32`, `vmul.s32`, and `vmla.s32` [BHK+22b, BMK+22].

Table 17: Overview of recommended constant-time 32-bit modular multiplications for Cortex-M processors. ISA stands for the instruction set architecture, mod. mul. stands for modular multiplication, Barrett (approx.) stands for the approximate variant of Barrett multiplication, and MVE stands for the M-profile vector extension.

Processor	ISA	Recommended mod. mul.
Cortex-M0	Armv6-M	Barrett (approx.), this work
Cortex-M0+	Armv6-M	Barrett (approx.), this work
Cortex-M3	Armv7-M	Barrett (approx.), this work
Cortex-M4	Armv7E-M	Montgomery, [GKS21, ACC+21]
Cortex-M7	Armv7E-M	Montgomery, [GKS21, ACC+21]
Cortex-M23	Armv8-M (Baseline)	Barrett (approx.), this work
Cortex-M33	Armv8-M (Mainline)	Montgomery, [GKS21, ACC+21]
Cortex-M52	Armv8.1-M (Mainline, MVE)	Barrett, [BHK+22b, BMK+22]
Cortex-M55	Armv8.1-M (Mainline, MVE)	Barrett, [BHK+22b, BMK+22]
Cortex-M85	Armv8.1-M (Mainline, MVE)	Barrett, [BHK+22b, BMK+22]

Algorithm 11 incurs some register pressure issues. Nevertheless, we focus on the arithmetic cost and ignore the register pressure issue of Algorithm 11 (Montgomery multiplication) for simplicity. On Cortex-M4, each of the instructions in Algorithms 10 and 11 takes one cycle based on the reference manuals [ARM10b, ARM10c]. Therefore, we need 9 arithmetic cycles for 64-bit Barrett multiplication and 15 arithmetic cycles for 64-bit Montgomery multiplication. Similarly, if one really wants to implement a 96-bit/128-bit shape-independent modular multiplication when precomputation is free, we believe Barrett multiplication is more favorable than Montgomery multiplication on 64-bit architectures such as `aarch64` and `x86-64`.

Algorithm 10 64-bit Barrett multiplication.

```
Inputs: $\texttt{alo} + 2^{32}\texttt{ahi} = a$.
Inputs: $\texttt{blo} + 2^{32}\texttt{bhi} = b$.
Inputs: $\texttt{bplo} + 2^{32}\texttt{bphi} = \left\lfloor \frac{2^{64}b}{q} \right\rceil$.
Outputs: $\texttt{clo} + 2^{32}\texttt{chi} \equiv ab \pmod{q}$.
umull t, hihi, alo, bphi
umull t, hilo, ahi, bplo
umaal hilo, hihi, ahi, bphi
umull clo, chi, alo, blo
umlal clo, chi, hilo, nqlo
mla chi, alo, bhi, chi
mla chi, ahi, blo, chi
mla chi, hilo, nqhi, chi
mla chi, hihi, nqlo, chi
```

Algorithm 11 64-bit Montgomery multiplication.

```
Inputs: $\texttt{alo} + 2^{32}\texttt{ahi} = a$.
Inputs: $\texttt{blo} + 2^{32}\texttt{bhi} = 2^{64}b \bmod q$.
Outputs: $\texttt{clo} + 2^{32}\texttt{chi} \equiv ab \pmod{q}$.
umull c0, c1, alo, blo
umull c2, c3, alo, bhi
umaal c1, c2, ahi, blo
umaal c2, c3, ahi, bhi
umull t0, t1, c0, qplo
mla t1, c0, qphi, t1
mla t1, c1, qplo, t1
umull h0, h1, t0, qlo
umull h2, h3, t0, qhi
umaal h1, h2, t1, qlo
umaal h2, h3, t1, qhi
adds c0, c0, h0
adcs c1, c1, h1
adcs c2, c2, h2
adc c3, c3, h3
```

NTT-friendly moduli in lattice-based cryptosystems. Our findings are of great importance to the industrial deployment of lattice-based cryptosystems. Even though the timing side-channel issue of 32-bit long multiplications on Cortex-M3 was already known in 2010 [GOPT10], several cryptosystems are designed with 32-bit arithmetic in mind and mandate the use of 32-bit modular arithmetic, including Dilithium in the NIST PQC Standardization initiated in 2016 [ABD+20a] and Raccoon in the NIST PQC Digital Signature Schemes initiated in 2023 [dPEK+23]. Our 32-bit Barrett multiplication comes to the rescue in this case. Furthermore, when people start designing cryptosystems built upon 64-bit prime moduli, our 64-bit Barrett multiplication will also come to the rescue for industrial deployment for legacy 32-bit machines as shown in previous paragraph.

NTT-friendliness of coefficient rings. We also challenge the notion of NTT-friendliness. Generally speaking, NTT is definable over arbitrary rings (also for non-commutative rings) [CK91], but many researchers introduced severely restricted forms of NTTs and some said NTTs over $\mathbb{Z}_{2^k}$ are not supported:

- In page 10 of the specification of Dilithium [ABD+20a]: “The Fast Fourier Transform is also called Number Theory Transform (NTT) in this case where the ground field is a finite field”.
- In page 23 of the specification of Saber [DKRV20]: “The use of two-power moduli makes NTT-like polynomial multiplication not natively supported.”

Our work brings attention to the performance of NTT-like polynomial multiplications over $\mathbb{Z}_{2^k}$, refreshing public understanding of asymptotically faster algorithms.

Performance figure of homomorphisms over $\mathbb{Z}_{r2^w+1}$ and $\mathbb{Z}_{2^k}$. This work shows that Nussbaumer over $\mathbb{Z}_{2^k}$ is the fastest approach when k is far away from the arithmetic precision, as illustrated by our Cortex-M3 implementation. On the other hand, Appendix C shows that if k is too close to the arithmetic precision such that high-dimensional Nussbaumer cannot be deployed, NTTs over NTT-friendly coefficient rings remain to be the fastest approach. Our findings bring attention to cryptographers on the impact of the shape of coefficient rings that one has to take the following into account:

1. Is $\mathbb{Z}_q$ NTT-friendly (so q must be an odd integer)?
2. If q is a power-of-two (so $\mathbb{Z}_q$ is not NTT-friendly), is $\log_2 q$ small enough compared to the arithmetic precision?

If $\mathbb{Z}_q$ is NTT-friendly, then one certainly applies the NTT over NTT-friendly coefficient ring. If $\mathbb{Z}_q$ is not NTT-friendly and q is a power-of-two (an even positive integer), one should apply the Nussbaumer approach if $\log_2 q$ is small enough compared to the arithmetic precision; and if $\log_2 q$ is too large, one should apply the NTT over a newly chosen NTT-friendly coefficient ring.

Applications to Kyber. Kyber [ABD+20b] is a key encapsulation mechanism selected by the NIST Post-Quantum Cryptography Standardization. In Kyber, the coefficient ring is $\mathbb{Z}_{3329}$ and all the elements in $\mathbb{Z}_{3329}$ are stored as halfwords. On Cortex-M3, all the multiplication instructions have 32-bit registers as operands and one can implement the 16-bit long/high multiplications efficiently with the $32 = 32 \times 32$ multiplication instructions `mul`/`mla`/`mls`. Therefore, Barrett multiplication does not translate into an improvement over the state-of-the-art modular multiplication by [AMOT22, HZZ+24]. On the other hand, we do know that Barrett multiplication translates into improvement over other modular multiplications for Kyber on 8-bit and 16-bit machines, such as the legacy 8-bit AVR and the 16-bit MSP430. Please refer to Appendix C.2 for our Kyber’s NTT/iNTT implementations on 8-bit AVR enviornment.

Notes on power consumption and more. In this work, we study the efficiency of Barrett multiplication modulo an odd modulus and Nussbaumer over $\mathbb{Z}_{2^k}$. In pure software optimization works targeting computation time, power consumption roughly scales with the computation time, assuming inputs are random-looking and reporting the power consumption does not convey much information. What is interesting is the power consumption of the computations of special inputs commonly involved in power side-channel attack works. Our Barrett multiplication and Nussbaumer introduce new computational aspects not explored in the literature, and the power-consumption aspects in the power-side-channel sense are left for future exploration.

Acknowledgments

This work is partially supported by the National Research Foundataion of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2022R1C1C1013368, 50%), and is partially supported by the Institute of Information and communications Technology Planning and Evaluation (IITP) Grant by the Korean Government through Ministry of Science and ICT (MSIT) (A study on PQC optimization and security protocol migration to neutralize advanced quantum attacks in Beyond 5G-based next-generation IoT computing environments, 25%) under Grant 2022-00207416, and is partly supported by IITP grant funded by MSIT (No. RS-2021-II210796, Research on Foundational Technologies for 6G Autonomous Security-by-Design to Guarantee Constant Quality of Security, 25%) and Korea Evaluation Institute of Industrial Technology(KEIT) grant funded by the Korea government.

References

[AB74] Ramesh C. Agarwal and Charles S. Burrus. Fast convolution using Fermat number transforms with applications to digital filtering. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 22(2):87–97, 1974. https://ieeexplore.ieee.org/abstract/document/1162555. 5

[ABB+23] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, Antoine Séré, and Pierre-Yves Strub. Formally verifying Kyber Episode IV: Implementation Correctness. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):164–193, 2023. https://tches.iacr.org/index.php/TCHES/article/view/10960. 12

[ABD+20a] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Dilithium. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://pq-crystals.org/dilithium/. 2, 7, 26, 40, 41

[ABD+20b] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Kyber. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://pq-crystals.org/kyber/. 2, 14, 26

[ACC+21] Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben Niederhagen, Cheng-Jhih Shih, Julian Wälde, and Bo-Yin Yang. Polynomial Multiplication in NTRU Prime Comparison of Optimization Strategies on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):217–238, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8733. 2, 13, 18, 25

[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 Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):127–151, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9292. 2, 6, 13, 18, 19, 20, 21, 34, 35, 39, 42, 43

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Dann Sprenkels. 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*, pages 853–871, 2022. https://link.springer.com/chapter/10.1007/978-3-031-09234-3_42. 5, 12

[AHY22] Erdem Alkim, Vincent Hwang, and Bo-Yin Yang. Multi-Parameter Support with NTTs for NTRU and NTRU Prime on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):349–371, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9823. 2, 13

[AMOT22] Daichi Aoki, Kazuhiko Minematsu, Toshihiko Okamura, and Tsuyoshi Takagi. Efficient Word Size Modular Multiplication over Signed Integers. In *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, pages 94–101. IEEE, 2022. https://ieeexplore.ieee.org/document/9974215. 2, 13, 26

[ARM10a] ARM. *Cortex-M3 Technical Reference Manual*, 2010. https://developer.arm.com/documentation/ddi0337/h. 8, 16

[ARM10b] ARM. *Cortex-M4 Technical Reference Manual*, 2010. https://developer.arm.com/documentation/ddi0439/b/. 25

[ARM10c] ARM. *CortexTM-M4 Technical Reference Manual ARM DDI 0439B Errata 01*, 2010. https://developer.arm.com/documentation/ddi0439/be. 25

[ARM12] ARM. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*, 2012. https://developer.arm.com/documentation/ddi0406/latest/. 24

[ARM21a] ARM. *Arm Architecture Reference Manual, Armv8, for Armv8-A architecture profile*, 2021. https://developer.arm.com/documentation/ddi0487/gb/?lang=en. 24

[ARM21b] ARM. *Armv7-M Architecture Refernce Manual*, 2021. https://developer.arm.com/documentation/ddi0403/ed. 8, 11

[Atm16] Atmel. *AVR Instruction Set Manual*, 2016. https://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf. 35

[Bar86] Paul Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *CRYPTO 1986*, LNCS, pages 311–323. SV, 1986. https://link.springer.com/chapter/10.1007/3-540-47721-7_24. 4

[BBC+20] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. NTRU Prime. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://ntruprime.cr.yp.to/. 20

[BBCT22] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, and Nicola Tuveri. OpenSSLNTRU: Faster post-quantum TLS key exchange. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 845–862, 2022. https://www.usenix.org/conference/usenixsecurity22/presentation/bernstein. 2, 13, 14, 20, 21, 22

[BHK+22a] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Lorenz Panny, and Bo-Yin Yang. Efficient Multiplication of Somewhat Small Integers using Number–Theoretic Transforms. *Cryptology ePrine Archive*, 2022. https://eprint.iacr.org/2022/439. 5, 12

[BHK+22b] 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 Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221–244, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9295. 2, 3, 4, 5, 7, 13, 24, 25

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4. In *Progress in Cryptology - AFRICACRYPT 2019*, volume 11627 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2019. https://doi.org/10.1007/978-3-030-23696-0_11. 2

[BMK+22] Hanno Becker, Jose Maria Bermudo Mera, Angshuman Karmakar, Joseph Yiu, and Ingrid Verbauwhedeg. Polynomial multiplication on embedded vector architectures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):482–505, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9305. 2, 13, 24, 25

[CCHY24] Han-Ting Chen, Yi-Hua Chung, Vincent Hwang, and Bo-Yin Yang. Algorithmic Views of Vectorized Polynomial Multipliers – NTRU. In Anupam Chattopadhyay, Shivam Bhasin, Stjepan Picek, and Chester Rebeiro, editors, *Progress in Cryptology – INDOCRYPT 2023*, pages 177–196. Springer Nature Switzerland, 2024. https://link.springer.com/chapter/10.1007/978-3-031-56235-8_9. 2, 6, 15

[CF94] Richard Crandall and Barry Fagin. Discrete Weighted Transforms and Large-integer Arithmetic. *Mathematics of computation*, 62(205):305–324, 1994. https://www.ams.org/journals/mcom/1994-62-205/S0025-5718-1994-1185244-1/?active=current. 5

[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 Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8791. 2, 13, 14, 21, 22

[CK91] David G. Cantor and Erich Kaltofen. On Fast Multiplication of Polynomials over Arbitrary Algebras. *Acta Informatica*, 28(7):693–701, 1991. https://link.springer.com/article/10.1007/BF01178683. 3, 26

[CT65] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965. https://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0178586-1/. 5

[DKRV20] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://www.esat.kuleuven.be/cosic/pqcrypto/saber/. 2, 26, 33, 34, 42, 43

[dPEK$^+$23] Rafael del Pino, Thomas Espitau, Shuichi Katsumata, Mary Maller, Fabrice Mouhartem, Thomas Prest, Mélissa Rossi, and Markku-Juhani Saarinen. Raccoon. Submission to the NIST Post-Quantum Cryptography: Digital Signature Schemes, 2023. https://raccoonfamily.org/. 26

[Fid73] Charles M. Fiduccia. On the Algebraic Complexity of Matrix Multiplication. 1973. https://cr.yp.to/bib/entries.html#1973/fiduccia-matrix. 6

[Für09] Martin Fürer. Faster Integer Multiplication. *SIAM Journal on Computing*, 39(3):979–1005, 2009. https://doi.org/10.1137/070711761. 5

[GKS21] Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact Dilithium Implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):1–24, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8725. 2, 9, 16, 18, 20, 22, 23, 25

[GOPT10] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications. In *Information, Security and Cryptology–ICISC 2009: 12th International Conference, Seoul, Korea, December 2-4, 2009, Revised Selected Papers 12*, pages 176–192. Springer, 2010. 9, 26

[GPW$^+$04] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and rsa on 8-bit cpus. In *Cryptographic Hardware and Embedded Systems-CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings 6*, pages 119–132. Springer, 2004. 37

[HAZ$^+$24] Junhao Huang, Alexandre Adomnicăi, Jipeng Zhang, Wangchen Dai, Yao Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Revisiting Keccak and Dilithium Implementations on ARMv7-M. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2), 2024. 8, 12, 13, 16, 19, 20, 21, 22, 23, 24

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A Modular Analysis of the Fujisaki-Okamoto Transformation. In *Theory of Cryptography Conference*, pages 341–371. Springer, 2017. 34

[HLS$^+$22] Vincent Hwang, Jiaxiang Liu, Gregor Seiler, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verified NTT Multiplications for NISTPQC KEM Lattice Finalists: Kyber, SABER, and NTRU. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 718–750, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9838. 12

[HLY24] Vincent Hwang, Chi-Ting Liu, and Bo-Yin Yang. Algorithmic Views of Vectorized Polynomial Multipliers – NTRU Prime. In Christina Pöpper and Lejla Batina, editors, *Applied Cryptography and Network Security*, pages 24–46. Springer Nature Switzerland, 2024. https://link.springer.com/chapter/10.1007/978-3-031-54773-7_2. 2

[Hwa22] Vincent Bert Hwang. Case Studies on Implementing Number–Theoretic Transforms with Armv7-M, Armv7E-M, and Armv8-A. Master's thesis, National Taiwan University, 2022. https://github.com/vincentvbh/NTTs_with_Armv7-M_Armv7E-M_Armv8-A. 7, 20

[Hwa24a] Vincent Hwang. Pushing the Limit of Vectorized Polynomial Multiplication for NTRU Prime. 2024. To appear at ACISP 2024, currently available at https://eprint.iacr.org/2023/604. 2, 6, 13, 20, 21, 22

[Hwa24b] Hwang, Vincent. A Survey of Polynomial Multiplications for Lattice-Based Cryptosystems. 2024. To appear at IACR Communication in Cryptology, currently available at https://eprint.iacr.org/2023/1962. 21

[HZZ$^+$22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray CC Cheung, Çetin Kaya Koç, and Donglong Chen. Improved Plantard Arithmetic for Lattice-based Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):614–636, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9833. 2, 37, 38

[HZZ$^+$24] Junhao Huang, Haosong Zhao, Jipeng Zhang, Wangchen Dai, Lu Zhou, Ray CC Cheung, Cetin Kaya Koc, and Donglong Chen. Yet another Improvement of Plantard Arithmetic for Faster Kyber on Low-end 32-bit IoT Devices. *IEEE Transactions on Information Forensics and Security*, 2024. https://ieeexplore.ieee.org/abstract/document/10453274. 13, 26, 37

[IKPC20] Írem Keskinkurt Paksoy and Murat Cenk. TMVP-based Multiplication for Polynomial Quotient Rings and Application to Saber on ARM Cortex-M4. *Cryptology ePrint Archive*, 2020. https://eprint.iacr.org/2020/1302. 2, 6, 18, 19

[IKPC22] Írem Keskinkurt Paksoy and Murat Cenk. Faster NTRU on ARM Cortex-M4 with TMVP-based multiplication. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(10):4083–4092, 2022. https://ieeexplore.ieee.org/document/9835023. 2, 6

[Jac12] Nathan Jacobson. *Basic Algebra II*. Courier Corporation, 2012. 13

[KO62] A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145(2), pages 293–294, 1962. http://cr.yp.to/bib/1963/karatsuba.html. 7

[KRS19] Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster Multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to Speed up NIST PQC Candidates. In *International Conference on Applied Cryptography and Network Security*, pages 281–301. Springer, 2019. https://link.springer.com/chapter/10.1007/978-3-030-21568-2_14. 2

[LSSR$^+$15] Zhe Liu, Hwajeong Seo, Sujoy Sinha Roy, Johann Großschädl, Howon Kim, and Ingrid Verbauwhede. Efficient ring-lwe encryption on 8-bit avr processors. In *Cryptographic Hardware and Embedded Systems–CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings 17*, pages 663–682. Springer, 2015. 36

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 598–616. Springer, 2009. 7

[Mar20] Jon Marsh. *Arm-Helium-Technology*, 2020. https://github.com/arm-university/Arm-Helium-Technology/blob/main/HeliumTechnology_referencebook.pdf. 24

[MKV20] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):222–244, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8550. 2

[Mon85] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of computation*, 44(170):519–521, 1985. https://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/?active=current. 2

[NG21] Duc Tri Nguyen and Kris Gaj. Fast NEON-based multiplication for lattice-based NIST post-quantum cryptography finalists. In *Post-Quantum Cryptography: 12th International Workshop, PQCrypto 2021, Daejeon, South Korea, July 20–22, 2021, Proceedings*, pages 234–254, 2021. https://link.springer.com/chapter/10.1007/978-3-030-81293-5_13. 2, 13

[NIS] NIST, the US National Institute of Standards and Technology. Post-quantum cryptography standardization project. https://csrc.nist.gov/Projects/post-quantum-cryptography. 27, 29, 30

[Nus80] Henri Nussbaumer. Fast Polynomial Transform Algorithms for Digital Convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(2):205–215, 1980. https://ieeexplore.ieee.org/document/1163372. 6

[ORGF$^+$22] Jheyne N Ortiz, Félix Carvalho Rodrigues, Décio Gazzoni Filho, Caio Teixeira, Julio López, and Ricardo Dahab. Evaluation of CRYSTALS-Kyber and Saber on the ARMv8 architecture. In *Anais do XXII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 372–377. SBC, 2022. https://sol.sbc.org.br/index.php/sbseg/article/view/21681. 2

[Pla21] Thomas Plantard. Efficient word size modular arithmetic. *IEEE Transactions on Emerging Topics in Computing*, 9(3):1506–1518, 2021. https://ieeexplore.ieee.org/document/9416314. 2, 37

[Ret21] RetroDan. *AVR Assembler Site*, 2021. https://avr-asm.tripod.com/avr201.html. 36, 37

[Sch77] Arnold Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Informatica*, 7(4):395–398, 1977. https://link.springer.com/article/10.1007/bf00289470. 20

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. 2018. https://eprint.iacr.org/2018/039. 2, 20, 36, 38

[Sho] Victor Shoup. NTL: a Library for Doing Number Theory. http://www.shoup.net/ntl/. 2, 4

[SKS+21a] Pakize Sanal, Emrah Karagoz, Hwajeong Seo, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Kyber on ARM64: Compact implementations of Kyber on 64-bit ARM Cortex-A processors. In *Security and Privacy in Communication Networks: 17th EAI International Conference, SecureComm 2021, Virtual Event, September 6–9, 2021, Proceedings, Part II*, pages 424–440. Springer, 2021. https://link.springer.com/chapter/10.1007/978-3-030-90022-9_23. 2

[SKS21b] JinGyo Song, YoungBeom Kim, and Seog Chung Seo. Optimization Study of Toom-Cook Algorithm in NIST PQC SABER Utilizing ARM/-NEON Processor. *Journal of the Korea Institute of Information Security & Cryptology*, 31(3):463–471, 2021. https://koreascience.kr/article/JAKO202118350341361.page. 2

[SLP+18] Hwajeong Seo, Zhe Liu, Taehwan Park, Hyeokchan Kwon, Sokjoon Lee, and Howon Kim. Secure number theoretic transform and speed record for ring-lwe encryption on embedded processors. In *Information Security and Cryptology–ICISC 2017: 20th International Conference, Seoul, South Korea, November 29-December 1, 2017*, pages 175–188. Springer, 2018. 38

[SS71] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3-4):281–292, 1971. https://link.springer.com/article/10.1007/BF02242355. 5

[STM20] STMicroelectronics. *STM32F207ZG*, 2020. https://www.st.com/en/microcontrollers-microprocessors/stm32f207zg.html. 19

[Too63] Andrei L. Toom. The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers. *Soviet Mathematics Doklady*, 3:714–716, 1963. http://toomandre.com/my-articles/engmat/MULT-E.PDF. 14

[Win80] Shmuel Winograd. *Arithmetic Complexity of Computations*, volume 33. Society for Industrial and Applied Mathematics, 1980. https://epubs.siam.org/doi/10.1137/1.9781611970364. 6, 7

[YMÖS21] Ferhat Yaman, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. A hardware accelerator for polynomial multiplication operation of crystals-kyber pqc scheme. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1020–1025. IEEE, 2021. 38

[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 *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 304–317, 2022. https://dl.acm.org/doi/abs/10.1145/3564625.3564629. 2

A Saber

Saber [DKRV20] is a lattice-based 3rd round finalist KEM based on Module Learning with Rounding. The module is of rank $l \times l$ over the polynomial ring $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ with $q = 2^{13}$ and $n = 256$. The core polynomial arithmetic is to multiply a matrix of rank $l \times l$ by a vector of l polynomials and inner product of vectors of l polynomials. The coefficients of the vector operand in matrix-vector multiplication and one of the operands in inner product are sampled from the centered binomial distribution with domain $\left\{-\frac{\mu}{2}, \dots, 0, \dots, \frac{\mu}{2}\right\}$. See Table 18 for parameters relevant to this work.

Table 18: Saber parameters [DKRV20] relevant to this work.

Parameter set	NIST security level	n	q	l	μ
`LightSaber`	I	256	2^{13}	2	10
`Saber`	III	256	2^{13}	3	8
`FireSaber`	V	256	2^{13}	4	6

In the key generation, we need one matrix-vector multiplication. In encryption, we need one matrix-vector multiplication and one inner product where one of the vectors is shared. In decryption, we only need one inner product. See Algorithms 12, 13, and 14 for an illustration of the construction of Saber's public key encryption (PKE) where operations in blue are the optimization targets in this paper. To instantiate a KEM, Saber employs a variant of Fujisaki-Okamoto transform due to [HHK17]. We need one encryption in the encapsulation and one encryption and one decryption in the decapsulation.

Algorithm 12 Saber PKE Key Generation.

```
Output: $pk = (\mathrm{seed}_A, b), sk = (s)$.
1: $\mathrm{seed}_A \leftarrow \texttt{Sample}_U()$
2: $A \in R_q^{l\times l} \leftarrow \texttt{Expand}(\mathrm{seed}_A)$
3: $s \in R_q^l \leftarrow \texttt{Sample}_B()$
4: $b \leftarrow \texttt{Round}\left(A^T s\right)$
```

Algorithm 14 Saber PKE Decryption.

```
Input: $ct = (c, b'), sk = (s)$.
Output: $m$.
1: $v \leftarrow b'^T s \mod p$
2: $m \leftarrow \texttt{Round}(v - 2^{\epsilon_p - \epsilon_T} c \mod p)$
```

Algorithm 13 Saber PKE Encryption.

```
Input: $m, r, pk = (\mathrm{seed}_A, b)$.
Output: $ct = (c, b')$.
1: $A \in R_q^{l\times l} \leftarrow \texttt{Expand}(\mathrm{seed}_A)$
2: $s' \in R_q^l \leftarrow \texttt{Sample}_B(r)$
3: $b' \leftarrow \texttt{Round}\,(As')$
4: $v' \leftarrow b^T s' \mod p$
5: $c \leftarrow \texttt{Round}(v' - 2^{\epsilon-1} m)$
```

B Saber on Cortex-M3

B.1 Matrix-Vector Multiplication and Inner Product

In Saber, the most time-consuming polynomial arithmetic is the matrix-vector multiplications in key generation and encryption. The second most time-consuming polynomial arithmetic is the inner products used in encryption and decryption. In encryption, since one of the vector operands is shared with the matrix-vector multiplication, the inner product is faster than the one in decryption [ACC+22]. We apply our Nussbaumer with Toom–Cook over $\mathbb{Z}_{2^k}$ to matrix-vector multiplications and inner products in all parameter sets of Saber. Compared to the state-of-the-art assembly implementations based on NTTs over NTT-friendly rings [ACC+22], we obtain $1.42-1.46$ times faster matrix-vector multiplications, $1.47-1.60$ times faster inner products for encryption, and $1.32-1.38$ times faster inner products for decryption (cf. Tabel 19).

B.2 Scheme

We compare our work to the state-of-the-art assembly-optimized implementation of Saber on Cortex-M3 by [ACC+22]. After replacing polynomial arithmetic with Nussbaumer

Table 19: Matrix-vector multiplications and inner products for Saber on Cortex-M3. `MV` stands for matrix-vector multiplication, and `IP` stands for the inner product. For the inner products, there are two numbers: one is used in encryption, and one is used in decryption.

Security level	`MV`		`IP`(Enc/Dec)	
	[ACC+22]	This work	[ACC+22]	This work
I	199k	136k (1.46)	83k/116k	52k (1.60) / 84k (1.38)
III	391k	272k (1.44)	114k/164k	75k (1.52) / 122k (1.34)
V	644k	454k (1.42)	144k/211k	98k (1.47) / 160k (1.32)

followed by Toeplitz-TC4 in Saber, we reduce the key generation and encapsulation cycles by around 15%, and decapsulation by around 17%. See Table 20 for a summary.

Table 20: Performance of Saber on Cortex-M3. **K** stands for key generation, **E** stands for encapsulation, and **D** stands for decapsulation.

NIST security level	Work	Operation					
		K		**E**		**D**	
		Cycles	Hash	Cycles	Hash	Cycles	Hash
I	[ACC+22]	464k	226k	628k	316k	686k	248k
	This work	397k	218k	534k	308k	568k	240k
III	[ACC+22]	853k	411k	1079k	534k	1153k	433k
	This work	743k	396k	918k	518k	962k	417k
V	[ACC+22]	1333k	620k	1608k	765k	1709k	642k
	This work	1131k	595k	1357k	739k	1424k	617k

C 8-bit AVR Implementations

The 8-bit AVR microcontroller architecture employs a straightforward two-stage pipeline. Most of its instructions execute in a single cycle. It is equipped with 32 general-purpose 8-bit registers, designated as `[r0:r31]`. Therefore, basic arithmetic operations, including bit operations, are performed on 8-bit units. We describe the relevant instructions in 8-bit AVR environment (cf. Table 21) [Atm16]. Analogous to the fundamental Cortex-M3, it supports 8-bit operations like `add`, `sub`, `adc`, and `sbc`. `lsl`/`lsr` logically shifts an 8-bit value left/right by one bit. `asr` performs an arithmetic 1-bit right-shift. Each of the above instructions takes one cycle. Excluding the early AVR architectures like the ATtiny series, which possesses byte-sized Static RAM (SRAM), the AVR microcontrollers primarily accommodate multiplication instructions via a dedicated hardware multiplication unit. The product of the multiplication is always returned in `[r0:r1]`. `mul` multiplies two unsigned 8-bit values, while `muls` multiplies two signed 8-bit values. `mulsu` multiplies 8-bit signed and unsigned values. These multiplication instructions take two cycles. Unlike `mul`, which allows all registers as operands, both `muls` and `mulsu` mandate the use of registers within the `[r16:r31]` range as operands.

We benchmark our 8-bit AVR implementation using the `IAR Embedded Workbench`. We simulate them on the `Generic Devices -v6` option with Max 16 MB of SRAM and 8 MB of flash memory. We compile our AVR implementations with the compiler of `IAR Embedded Workbench` version `8.10.1` using `High(speed)` level optimization option. Since

Table 21: Instruction timings on 8-bit AVR where inputs are 8-bit registers.

Instruction	Cycle
`add/adc/sub/sbc/lsl/lsr/asr`	1
`mul/muls/mulsu`	2

8-bit AVR comprise the single-pipeline structure, our simulations provide cycle counts equivalent to the benchmarks. To measure the stack size, we use the linker option (`Enable stack usage analysis`) of `IAR Embedded Workbench`, and the code size is measured through the information in the `.map` file.

C.1 32-bit Modular Multiplication on 8-bit AVR

We implement the full range of 16/32-bit low/high/long multiplications with 8-bit words and build Montgomery (cf. Algorithm 15) and Barrett (cf. Algorithm 16) multiplication with them.

Algorithm 15 Our constant-time Montgomery multiplication for Dilithium on 8-bit AVR adapted from [Sei18].

```
Inputs: $(\texttt{a3}\|\cdots\|\texttt{a0}) = a$, $(\texttt{b3}\|\cdots\|\texttt{b0}) = b$
Outputs: $(\texttt{r7}\|\cdots\|\texttt{r4}) = \frac{ab-\left(abq^{-1} \bmod {}^{\pm}2^{32}\right)q}{2^{32}}$.
muls32x32_64  a0, ..., a3, b0, ..., b3, r0, ..., r8          ▷ $r = ab$.
muls32xQinv_lo32  r0, ..., r3, qiimm, t0, ..., t3     ▷ $t = abq^{-1} \bmod {}^{\pm}2^{32}$.
muls32xQ_hi32  t0, ..., t3, qimm, t4, ..., t7         ▷ $t = \frac{\left(abq^{-1} \bmod {}^{\pm}2^{32}\right)q}{2^{32}}$.
sub r4, t4     sbc r5, t5     sbc r6, t6     sbc r7, t7
                                   ▷ $(\texttt{r7}\|\cdots\|\texttt{r4}) = \frac{ab-\left(abq^{-1} \bmod {}^{\pm}2^{32}\right)q}{2^{32}}$.
```

Algorithm 17 Implementation of `muls32xQ_lo32` for Dilithium on 8-bit AVR.

```
Inputs: $(\texttt{a3}\|\cdots\|\texttt{a0}) = a$
Outputs: $(\texttt{c3}\|\cdots\|\texttt{c0}) = aq \bmod {}^{\pm}2^{32}$
movw c0, a0     movw c2, a2     ldi q, 0xE0     mul a0, q
add c1, r0      adc c2, r1      adc c3, zero    mul a2, q
add c3, r0      mul a1, q       add c2, r0      adc c3, r1
ldi q, 0x7F     mul a0, q       add c2, r0      adc c3, r1
mul a1, q       add c3, r0
```

Table 22 summarizes the performance of plain multiplications (low, high, and long multiplications) and modular multiplications (Montgomery and Barrett multiplications).

For the Barrett multiplication, we call three 16-bit long multiplications (one call to `muls16x16_32` and two calls to `mulsu16x16_32`) and two 32-bit low multiplications (`muls32xQ_lo32` and `muls32x32_lo32`). The generic multipliers are built upon the "Move-and-Add" technique [LSSR+15, Ret21]. The macro `muls16x16_32` multiplies two signed 16-bit values, and macro `mulsu16x16_32` multiplies 16-bit signed and unsigned values. We further optimize the 32-bit multiply-low when $q = 2^{23} - 2^{13} + 1$ is one of the operands by exploiting the fact that the least significant byte of q is 1.

For Montgomery multiplication, we adapt the subtractive version from [Sei18] with one 32-bit low multiplication, one 32-bit high multiplication, and one 32-bit long multiplication. Since the least significant words of q and $q^{-1} \bmod {}^{\pm}2^{32}$ in Dilithium are 1,

Algorithm 16 Constant-time Barrett multiplication for Dilithium on 8-bit AVR.

```
Inputs: $(\texttt{a3}\|\cdots\|\texttt{a0}) = a, (\texttt{b3}\|\cdots\|\texttt{b0}) = b, (\texttt{bp3}\|\cdots\|\texttt{bp0}) = \left\lfloor \frac{2^{32}b}{q} \right\rceil$.
        $((\texttt{a3}\|\texttt{a2}) = a_h,\ (\texttt{a1}\|\texttt{a0}) = a_l,\ (\texttt{bp3}\|\texttt{bp2}) = b_h,\ (\texttt{bp1}\|\texttt{bp0}) = b_l)$
Outputs: $(\texttt{c3}\|\cdots\|\texttt{c0}) = c = ab - \left\llbracket \frac{a\left\lfloor \frac{2^{32}b}{q} \right\rceil}{2^{32}} \right\rrbracket_1 q$.
muls16x16_32    a2, a3, bp2, bp3, c0, ···, c3             ▷ $c = a_h b_h$.
mulsu16x16_32   bp2, bp3, a0, a1, t0, ···, t3             ▷ $t = a_l b_h$.
mov r0, t3      lsl r0      sbc r0, r0                    ▷ $\texttt{r0} = \mathrm{SignExtend}(\texttt{t3}[7:7])$.
add c0, t2      adc c1, t3  adc c2, r0  adc c3, r0        ▷ $c = a_h b_h + \left\lfloor \frac{a_l b_h}{2^{16}} \right\rfloor$.
mulsu16x16_32   a2, a3, bp0, bp1, t0, ···, t3             ▷ $t = a_h b_l$.
mov r0, t3      lsl r0      sbc r0, r0                    ▷ $\texttt{r0} = \mathrm{SignExtend}(\texttt{t3}[7:7])$.
add c0, t2      adc c1, t3  adc c2, r0  adc c3, r0
        ▷ $c = a_h b_h + \left\lfloor \frac{a_l b_h}{2^{16}} \right\rfloor + \left\lfloor \frac{a_h b_l}{2^{16}} \right\rfloor = \left\llbracket \frac{a\left\lfloor \frac{2^{32b}}{q} \right\rceil}{2^{32}} \right\rrbracket_1$.
muls32xQ_lo32   c0, ···, c3, qimm, t0, ···, t3            ▷ $t = \left\llbracket \frac{a\left\lfloor \frac{2^{32}b}{q} \right\rceil}{2^{32}} \right\rrbracket_1 q$
muls32x32_lo32  a0,···,a3, b0,···,b3, c0,···,c3           ▷ $r = ab \bmod^{\pm} 2^{32}$.
sub c0, t0      sbc c1, t1  sbc c2, t2  sbc c3, t3
        ▷ $(\texttt{c3}\|\cdots\|\texttt{c0}) = c = ab - \left\llbracket \frac{a\left\lfloor \frac{2^{32}b}{q} \right\rceil}{2^{32}} \right\rrbracket_1 q$.
```

we optimize `muls32xQinv_lo32` and `muls32xQ_hi32` similarly as in `muls32xQ_lo32` (cf. Algorithm 17). For the 64-bit product ab, we implement `muls32x32_64` with the row-wise multiplication [GPW+04] technique for 32-bit long multiplication.

As shown in Table 22, the Barrett multiplication is $\frac{184}{129} \approx 1.43$ times faster than Montgomery multiplication, and close to the 32-bit long multiplication `muls32x32_64`.

Table 22: Overview of assembly-optimized multiplication operations with 8-bit representation of 32-bit input values on 8-bit AVR. `Q` is the modulus q of Dilithium.

Plain multiplication		
Multiplication operation	Work	Cycle
`mulsu_16x16_32`	[Ret21]	17
`muls_16x16_32`	[Ret21]	18
`muls32xQ_lo32`	This work	23
`muls32x32_lo32`	[Ret21]	36
`muls32xQinv_lo32`	This work	27
`muls32xQ_hi32`	This work	51
`muls32x32_64`	[Ret21]	102
Modular multiplication		
Montgomery multiplication	This work	184
Barrett multiplication	This work	129

C.2 16-bit Modular Multiplication on 8-bit AVR

We will show the efficiency of 16-bit Barrett multiplication on 8-bit AVR. In this section, we denote 16-bit primes as q', for instance, $q' = 3329$ or 7681. Considering the overhead of signed 16-bit multiplication (18 cycles), we implement Solinas multiplication and Plantard multiplication [Pla21, HZZ+22, HZZ+24], signed Montgomery multiplication, and our signed Barrett multiplication. Additionally, we include a comparison with the unsigned LookUp-Table multiplication introduced in [SLP+18]. For $q' = 3329$, we implement the half-approximate variant of Barrett multiplication (cf. Table 3) along with the lazy reduction for NTT/iNTT. The upper bound $\frac{q}{2}\left(5 + \frac{|a|}{\mathtt{R}}\right)$ allows the application of the same lazy reduction strategy as Montgomery multiplication in Kyber's NTT/iNTT.

Table 23 summarizes the performance of various modular multiplications for $q' = 3329$. Our Barrett multiplication achieves approximately 30% performance improvement over Montgomery multiplication and is the fastest in the AVR environment. Even when using the Solinas multiplication to reduce the overhead of 16-bit multiplication, it remains inefficient unless q' is a complete Mersenne prime. For instance, Kyber's $q' = 3329 = 2^{12} - 2^{9} - 2^{8} + 1$ requires significant additional computations, as shown in Table 23. The Plantard multiplication is relevant in environments with 32-bit registers and powerful multiplication instructions such as `smulwb` and `smlabb` . The LookUp-Table multiplication, proposed for unsigned-representation, is less efficient in AVR environments due to the additional stack or flash memory usage, despite the absence of cache memory.

Table 23: Overview of assembly-optimized multiplication 16-bit modular multiplication ($q' = 3329$) on 8-bit AVR

Modular multiplication			
Type	Arithmetic	Work	Cycle
Unsigned	Solinas multiplication [YMÖS21]	This work	94
Signed	Plantard multiplication [HZZ+22]	This work	74
Unsigned	LookUp-Table multiplication [SLP+18]	[SLP+18]	57
Signed	Montgomery multiplication [Sei18]	This work	50
Signed	Barrett multiplication	This work	38

While we do not fully implement Kyber's entire scheme in this paper, we show the effectiveness of Barrett multiplication for 16-bit polynomial multiplication through Kyber's NTT/iNTT performance. Table 24 shows a comparison between Montgomery NTT and Barrett NTT for Kyber on an 8-bit AVR platform. Our Barrett NTT outperforms Montgomery NTT by approximately $\frac{170646}{106450} \approx 1.60\times$ and $\frac{202484}{125337} \approx 1.61\times$ in C language-based NTT and iNTT, also the assembly implementation is approximately $\frac{67067}{60005} \approx 1.12\times$ and $\frac{84337}{64337} \approx 1.31\times$ times faster. As mentioned earlier, the lazy reduction due to Half-approximate eliminates the need for reduction in the middle of the NTT layer. Both implementations differ only in pure arithmetic (Mongomery and Barrett). Furthermore, our Barrett multiplication can be applied to rings with polynomial coefficients of 16 bits or less, including NTTRU.

Table 24: Performance cycles of NTT/iNTT of Kyber on 8-bit AVR.

	C		Assembly	
	Montgomery	Barrett	Montgomery	Barrett
NTT	170 646	106 450 (1.60)	67 067	60 005 (1.12)
iNTT	202 484	125 337 (1.61)	84 461	64 337 (1.31)

C.3 Dilithium

C.3.1 Dilithium NTT

Our C implementations of Barrett NTT and iNTT are $\frac{2709601}{449457} \approx 6.03\times$ and $\frac{3191757}{468207} \approx 6.82\times$ faster than the C implementations of Montgomery ones. The main reason is that for the long multiplication, the AVR compilers (`IAR Embedded Workbench` version `8.10.1` and `AVR-gcc` version `13.1.0`) compile it into a sign extension with a series of `lsl` and `sbc` instructions and a series of `mul` instructions multiplying signed 8-bit words. Although this approach results in much simpler register scheduling due to the flexibility of `mul` operands, the resulting performance is slow. Our C implementation of Barrett Dilithium NTT/iNTT avoids the long multiplications and, hence, is much faster.

As for the assembly implementations, we implement the assembly of Montgomery NTT/iNTT with `muls` for fair comparisons. Since One have to maintain the 64-bit product in the register while applying Montgomery multiplication, which necessitates additional book-keeping operations, assembly-optimized Montgomery NTT/iNTT is slow. On the other hand, our generalization of Barrett multiplication, which avoids the long multiplication, removes the book-keeping operations (our generalization is necessary for this to happen). This is why Barrett NTT and iNTT are 2.38–2.53 faster than Montgomery NTT and iNTT, while the Barrett multiplication is only 1.43 times faster than Montgomery multiplication. See Table 25 for a summary.

Table 25: Performance cycles of NTT/iNTT of Dilithium on 8-bit AVR.

	C		Assembly	
	Montgomery	Barrett	Montgomery	Barrett
NTT	2 709 601	449 457 (6.03)	482 828	202 917 (2.38)
iNTT	3 191 757	468 207 (6.82)	596 740	236 028 (2.53)

C.3.2 Matrix-Vector Product

Table 26 compares the performance of matrix-vector products with assembly-optimized NTTs with Montgomery and Barrett multiplications. We find that the Barrett implementations are 1.87 − 1.89 times faster than the Montgomery implementations.

Table 26: Performance cycles of Dilithium matrix-vector products with assembly on 8-bit AVR.

	II	III	V
Montgomery	5 030 708	6 342 703	8 399 225
Barrett	2 662 661	3 389 428	4 501 286

C.3.3 Multiplication with the Challenge Polynomial

We apply a small NTT to cs_1 and cs_2, the same as the approach for the Cortex-M series. We apply FNT ($q = 257$) for security levels II and V, and 16-bit NTT ($q = 769$) for security level III. Our implementation of cs_1 (cs_2) is 4.74 (4.74), 3.29 (3.30), and 4.76 (4.76) times faster than 32-bit NTT for security levels II, III, and V, respectively.

As for ct_0, ideally, one should apply the two 16-bit NTT approach from [ACC$^+$22]. We show that this approach is 1.44 times faster than the 32-bit NTT approach for all security levels as shown in Table 27. However, we do not apply the two 16-bit NTT approach to the ct_0 in our implementations of the full Dilithium scheme. We conclude that placing the twiddle factor of the 32-bit NTT, FNT (small NTT) for cs_1 (cs_2), and the two 16-bit NTTs (ct_0) all in memory is not a realistic implementation option in an 8-bit AVR board.

Table 27: Performance cycles of the multiplication with the challenge polynomial with assembly on 8-bit AVR. “32-bit” refers to the 32-bit Barrett NTT approach, and “16-bit” refers to the 16-bit FNT/NTT approach for cs_1 and cs_2 and two 16-bit NTT approach for ct_0.

	II		III		V	
	32-bit	16-bit	32-bit	16-bit	32-bit	16-bit
cs_1	2 642 049	557 722	3 251 832	987 687	4 471 398	940 201
cs_2	2 642 049	557 722	3 861 615	1 171 636	5 081 181	1 067 694
ct_0	2 642 049	1 831 234	3 861 615	2 683 698	5 081 181	3 536 162

C.3.4 Scheme

Since the reference implementation consumes a significant amount of SRAM in the 8-bit AVR environment and cannot be simulated, we implement a stack-optimized version based on the reference code [ABD$^+$20a] and designate it as the baseline (denoted as Ref). Table 28 summarizes the performance of Dilithium. Table 29 shows a stack usage of our implementation.

First, we compare the Ref using Montgomery multiplication with our implementation using Barrett multiplication. Both implementations are coded in the C language. For `dilithium2`, we reduce the key generation, signature generation, and signature verification cycles by 35.11%, 60.38%, and 43.43%, respectively. For `dilithium3`, we reduce the key generation, signature generation, and signature verification cycles by 45.09%, 59.20%, and 50.40%, respectively. For `dilithium5`, we reduce the key generation, signature generation, and signature verification cycles by 43.18%, 54.48%, and 47.04%, respectively. The contribution entirely comes from our **C program** implementing our generalization of Barrett multiplication.

Subsequently, we compare our hand-written assembly implementation with the C implementations. For `dilithium2`, we reduce the key generation, signature generation, and signature verification cycles by 39.94%, 64.90%, and 49.25%, respectively. For `dilithium3`, we reduce the key generation, signature generation, and signature verification cycles by 48.85%, 63.03%, and 55.08%, respectively. For `dilithium5`, we reduce the key generation, signature generation, and signature verification cycles by 46.86%, 58.09%, and 51.52%, respectively.

All implementations (Ref** (C), This work (C), and This work (ASM)) of each Dilithium process consume the same stack size, and the code size is about 50 KiB in all implementations. As can be seen from the comparison between C implementations, the 32-bit multiply-long instruction is one of the most significant bottlenecks in the 8-bit AVR environment. Especially, the significant performance improvement in the Dilithium signature generation, dominated by the rejection loop, clearly highlights the benefits of Barrett multiplication. With 16KB of SRAM, the 8-bit AVR board (e.g., ATMEGA1284-AU and AVR128DB48) makes Dilithium security levels II and III practically feasible options.

C.4 Saber

C.4.1 Fast Homomorphisms

Table 30 summarizes the performance of various homomorphisms for multiplication over $\mathbb{Z}_q$ within the range of $2^8 < q < 2^{16}$ in the AVR environment. For an NTT-friendly $\mathbb{Z}_q$,

Table 28: Performance cycles of Dilithium on 8-bit AVR.

	Work	Operation					
		K		**S**		**V**	
		Cycles	Hash	Cycles	Hash	Cycles	Hash
II	Ref** (C)	73 556k	29 825k	166 961k	37 282k	86 860k	28 391k
	This work (C)	47 732k	29 825k	66 156k	37 282k	49 138k	28 391k
	This work (ASM)	44 181k	29 825k	58 600k	37 282k	44 081k	28 391k
III	Ref** (C)	154 028k	53 915k	491 601k	119 877k	169 770k	49 901k
	This work (C)	84 579k	53 915k	200 597k	119 877k	84 213k	49 901k
	This work (ASM)	78 786k	53 915k	181 787k	119 877k	76 267k	49 901k
V	Ref** (C)	255 058k	92 632k	1 091 977k	310 304k	276 570k	89 191k
	This work (C)	144 925k	92 632k	497 054k	310 304k	146 478k	89 191k
	This work (ASM)	135 525k	92 632k	457 611k	310 304k	134 076k	89 191k

** Our stack-optimized implementation based on reference code [ABD+20a].

Table 29: Stack usage in bytes of Dilithium on 8-bit AVR. **K** stands for key generation, **S** stands for signature generation, and **V** stands for signature verification.

NIST security level	Work	Operation		
		K	**S**	**V**
		Stack	Stack	Stack
II	Ref** (C)	9 282	12 059	12 751
	This work (C, ASM)	9 282	12 059	12 751
III	Ref** (C)	11 330	14 107	13 775
	This work (C, ASM)	11 330	14 107	13 775
V	Ref** (C)	13 378	16 155	16 079
	This work (C, ASM)	13 378	16 155	16 079

NTT exhibits the fastest performance. We choose $q = 3329$ for the 16-bit NTT and $q = 257$ for the 16-bit FNT. There are used in Dilithium. For the power-of-two $q = 2^k \leq 2^{16}$ in Saber, we explore three approaches: (i) The 4-way Toom–Cook approach; (ii) The striding 4-way Toom–Cook approach; and (iii) The Nussbaumer approach. For the Nussbaumer approach with 16-bit arithmetic, we have to resort to

$$\frac{\mathbb{Z}_{2^k}[x]}{\langle x^{256}+1\rangle} \to \prod_i \frac{\left(\frac{\mathbb{Z}_{2^{k+3}}}{y^{64}+1}\right)[x]}{\langle x - y^{16i}\rangle}$$

resulting in 8 size-64 polynomial multiplications since $2^{k+3} = 2^{16}$ in the Saber case. In this case, the base multiplication phase is significantly slower than the transformation phase due to the large number of subproblems. We find that striding TC is the fastest, followed by TC and Nussbaumer with 16-bit arithmetic.

C.4.2 Matrix-Vector Multiplication and Inner Product

We apply the 16-bit NTT and striding TC approaches to the matrix-vector multiplications and inner products of Saber. We briefly compare their performance based on the assembly-optimized 16-bit NTTs and striding TC. The matrix-vector multiplications with 16-bit

Table 30: Performance cycles of homomorphisms with 16-bit arithmetic in AVR.

C				
	Striding TC	TC4 [DKRV20]	Nussbaumer	16-bit NTT
Hom/NTT	12 011	45 532	67 379	105 450
Mul.	727 802	822 331	910 161	32 264
Hom-I/iNTT	46 532	65 432	88 069	125 337
ASM				
	Striding TC	16-bit NTT	16-bit FNT	
Hom/NTT	7 016	60 005	47 750	
Mul.	565 511	26 732	21 338	
Hom-I/iNTT	25 175	64 337	58 405	

NTTs are 1.68−2.11 times faster than the ones with striding TC, and the inner products with 16-bit NTTs are 1.79−2.62 times faster than the ones with striding TC. See Table 31 for a summary.

Table 31: Performance cycles of matrix-vector multiplications and inner products in Saber on 8-bit AVR.

NIST level Security	[DKRV20] (C) (TC4)	This work (ASM)† (2 × 16 NTT)	This work (C) (Striding TC)	This work (ASM) (Striding TC)
	MV			
I	3 733k	1 386k	3 193k	2 419k
III	8 400k	2 600k	7 185k	5 443k
V	14 933k	4 160k	12 774k	9 676k
	IP(Enc/Dec)			
I	1 867k	573k / 716k	1 597k	1 209k
III	2 800k	747k / 1 009k	2 395k	1 814k
V	3 733k	920k / 1 303k	3 193k	2 419k

† : Our 2 × 16 NTT implementation based on Strategy A of [ACC+22].

C.4.3 Scheme

Table 32 summarizes the performance of reference implementation and our implementations with striding Toom–Cook and 16-bit NTTs. We compare the reference implementation to the fastest implementation with 16-bit NTTs. For LightSaber, we reduce the key generation, encapsulation, and decapsulation cycles by 18.79%, 15.41%, and 17.95%, respectively. For Saber, we reduce the key generation, encapsulation, and decapsulation cycles by 24.76%, 16.84%, and 19.35%, respectively. For FireSaber, we reduce the key generation, encapsulation, and decapsulation cycles by 28.75%, 17.80%, and 19.50%, respectively.

Table 32: Performance cycles of Saber on 8-bit AVR.

NIST security level	Work	Operation					
		K		**E**		**D**	
		Cycles	Hash	Cycles	Hash	Cycles	Hash
I	[DKRV20] (C)	11 363k	5 736k	16 075k	8 030k	15 816k	6 309k
	This work (ASM)	10 086k	5 736k	14 161k	8 030k	13 368k	6 309k
	This work (ASM)†	9 228k	5 736k	13 596k	8 030k	12 977k	6 309k
III	[DKRV20] (C)	21 828k	10 611k	28 496k	13 766k	28 153k	11 185k
	This work (ASM)	18 959k	10 611k	24 678k	13 766k	23 379k	11 185k
	This work (ASM)†	16 424k	10 611k	23 697k	13 766k	22 706k	11 185k
V	[DKRV20] (C)	35 266k	16 060k	43 553k	19 788k	43 469k	16 633k
	This work (ASM)	30 171k	16 060k	37 197k	19 788k	35 944k	16 633k
	This work (ASM)†	25 126k	16 060k	35 800k	19 788k	34 900k	16 633k

† : Our 2×16 NTT implementation based on Strategy `A` of [ACC$^+$22].