Code Generation of Faster Formally Verified NTT with Plantard Reduction

Donnie Y. Xu[1⋆], Rajeev Gore[1], Amin Sakzad[1][0000-0003-4569-3384], Ron Steinfeld[1][0000-0003-1745-4183], and Raymond K. Zhao[2][0000-0003-1257-9147]

[1] Monash University, Melbourne, Australia
yxuu0182@student.monash.edu, rajeev.gore@monash.edu, amin.sakzad@monash.edu, ron.steinfeld@monash.edu
[2] ExeQuantum, Melbourne, Australia
raymond@exequantum.com

Abstract. We present a formally verified implementation of the ML-KEM Number-Theoretic Transform (NTT) based on Plantard arithmetic, produced via a code generator that targets ML-KEM, ML-DSA, and FN-DSA from a single parameter triple. The generator embeds a static bound analyzer that places modular reductions at code-generation time without runtime branching, eliminating per-scheme manual tuning while preserving constant-time guarantees. Each generation produces structurally identical implementations in two backends: portable C, and Jasmin for formal verification. To establish end-to-end correctness, we contribute a parametric formalization of Plantard arithmetic in EasyCrypt and a layer-by-layer program-equivalence proof connecting the extracted Jasmin ML-KEM NTT to the abstract specification of formosa-mlkem; the existing algebraic chain is reused unchanged to extend correctness down to the mathematical NTT definition. Benchmarks across three schemes show that the generated code outperforms reference C by 1.5×–1.8× on the forward NTT and 1.7×–2.5× on the inverse, and outperforms the formally verified formosa-mlkem Jasmin baseline by 1.26× and 2.19× on ML-KEM. We believe our techniques generalize to other lattice-arithmetic primitives requiring both performance and formal verification.

Keywords: Number Theoretic Transform · Plantard Reduction · Lazy Reduction · Formal Verification · ML-KEM · Lattice-based Cryptography

1 Introduction

Shor's algorithm [32] run on a sufficiently scalable quantum computer can efficiently solve the integer factorization and discrete logarithm problems underlying

⋆ The first author is the main contributor; the remaining authors contributed equally and are listed alphabetically by surname.

widely deployed public-key cryptography. In response, the National Institute of Standards and Technology (NIST) organized a competition to select and standardize cryptographic schemes that can resist quantum attacks. To date, NIST has standardized ML-KEM [27], ML-DSA [26], and SLH-DSA [28], with FN-DSA (derived from Falcon [16]) and HQC [24] to follow. However, implementing them efficiently, securely, and correctly becomes another significant concern.

Given the scale of data these schemes will protect and the frequency at which they will be used, their implementations are highly sensitive to efficiency. Among the NIST PQC competition winners, three are lattice-based, while SLH-DSA is hash-based and HQC is code-based. These schemes heavily rely on matrix multiplications where each entry is a polynomial ring. Their efficiency is therefore dominated by the speed of polynomial multiplication. The Number-Theoretic Transform (NTT) [30] is a common method that can speed up polynomial modular multiplication and is used by the lattice-based NIST PQC standards.

Within each NTT implementation, modular reductions are frequently applied to avoid overflow and maintain bounds for subsequent operations. This suggests that efficient modular reduction is a natural optimization target. For this purpose, various modular reduction techniques have been developed, including Montgomery reduction [25], Barrett reduction [7], and more recent Plantard arithmetic [29,19,20]. Montgomery and Barrett reductions are widely deployed in lattice-based cryptographic implementations, while Plantard modular arithmetic (hereafter, *arithmetic* refers to *modular arithmetic*) has only recently been adopted. However, due to the complex nature of NTT with embedded modular reductions, ensuring that optimizations are correctly applied is difficult, and most of them require hand-tuning. Cryptographic implementations also require high assurance, while traditional software testing cannot handle all edge cases or hidden bugs, and exhaustively testing all inputs is infeasible.

Formal verification is a solution to address the assurance issue, and can be used to formally prove that an implementation correctly behaves as mathematically specified for all inputs through induction. Almeida et al. [3] from Formosa-Crypto developed an end-to-end formal verification framework and proof of ML-KEM in their own toolchain—Jasmin [2] and EasyCrypt [8], using Montgomery and Barrett modular arithmetic. They follow a lazy-reduction approach in reference implementation that does not apply any modular reduction after addition and subtraction. Trieu [33], building on Fiat-Crypto, developed a Rocq-based framework for parametric NTT verification that is sufficiently generic for future usage. Although they considered lazy-reduction formalization in their verification pipeline, they explicitly deferred it as future work along with Plantard arithmetic formalization.

A separate line of work integrates generation and verification. Masuda and Kameyama [22] developed a standalone verification pipeline based on a Domain-Specific Language (DSL), a custom language that allows the design of a framework tailored to a specific use. Here, they provided a single framework that handles both code generation and verification of NTT functional correctness. Their DSL's tagless-final style [11] for code generation also naturally enables convenient

interval analysis. By using this method, their pipeline targets NewHope's [1] NTT with Montgomery and Barrett reductions, and applies detailed interval analysis that derives an aggressive lazy Barrett reduction.

Plantard arithmetic [29,19,20] has recently emerged as the fastest modular reduction for lattice-based NTT on scalar and embedded targets, and recent work has begun to verify Plantard-based implementations [12,5]. These efforts verify hand-written implementations at fixed parameters. However, no reusable Plantard formalization is available in the Jasmin/EasyCrypt toolchain, nor has Plantard been used as the basis of a code generator for existing cryptographic primitives. Several factors pose challenges for Plantard arithmetic formalization and its integration into existing NTT verification pipelines. The tunable variable that controls the asymmetric input and output ranges makes preconditions and postconditions harder to establish than the symmetric ranges of Montgomery and Barrett reductions. Therefore, Plantard's correctness proof is more sophisticated than its Montgomery and Barrett counterparts, posing challenges for the proper reasoning in a proof assistant.

Scalar implementations are the portability baseline for these schemes. Although vectorized backends provide the best absolute performance, they must be written and verified per architecture, whereas a single scalar implementation compiles everywhere. Considering software capabilities and hardware architecture, only SSE2 is guaranteed on x86-64, while AVX2 sits two microarchitecture levels above this baseline [23], so Linux distributions building for generic x86-64 cannot assume it is available. Vectorized paths are moreover often selected at compile time rather than at run time. For instance, `mlkem-native` [21] assumes by default that a feature enabled at build time is available on the host, and requires an explicit override for runtime detection, so a build for generic x86-64 contains only the scalar path even on hardware that supports AVX2. Plantard arithmetic is well suited to this setting, since its advantage over Montgomery lies in requiring one fewer multiplication per butterfly, and it targets the small word-sized moduli of lattice schemes rather than the large moduli of RSA and elliptic-curve cryptography. Our generator therefore emits scalar C and Jasmin, and we benchmark and compare against scalar implementations throughout.

Our Contributions. To address the gaps mentioned above, in this paper, we develop a code generator that produces forward and inverse NTT routines using Plantard arithmetic in C and Jasmin backends. We further translate the Plantard correctness proof into an EasyCrypt module with abstracted parameters that facilitate its use in other contexts not necessarily related to NTT. Using this Plantard formalization, we formally verify the generated Jasmin ML-KEM NTT routines end-to-end in EasyCrypt against the abstract NTT specification of Formosa-mlkem [3]. Specifically, our contributions are listed as follows:

1. We develop *Lazy-Generator*, a Python-based code generator for NTT implementations. It takes a (modulus, ring dimension, root of unity mod q) parameter triple (q, n, ω) from schemes and outputs forward and inverse NTT routines using Plantard arithmetic in C and Jasmin. Reductions are

automatically placed within the NTT pipeline when overflow is detected at code-generation time through static bound analysis.
2. We formalize Plantard arithmetic as two separate EasyCrypt `theories` (`Plantard16` and `Plantard32`), which contain the abstracted parameters sufficiently generic for reuse across schemes, and are hence of independent interest (i.e. not just for NTT).
3. We then formally verify the generated Jasmin ML-KEM NTT routines end-to-end in EasyCrypt using formalized Plantard arithmetic. In particular, we provide an equivalence proof between the extracted Jasmin output and the abstract NTT specification from Formosa-mlkem [3]. Combined with [3]'s results, this gives the first end-to-end formally verified Jasmin implementations of ML-KEM with Plantard arithmetic.
4. Finally, we benchmark the generated NTT implementations across ML-KEM, ML-DSA and FN-DSA. We observe speedups of 1.47×–1.83× on the forward NTT and 1.66×–2.48× on the inverse NTT over the NTT routines from the NIST PQC reference C implementations, and 1.63× on round-trip NTT over the formally verified Jasmin baseline for ML-KEM NTT using Montgomery arithmetic [3]. To demonstrate the practical utility of the generator, we further deploy the generated NTT inside ML-KEM and ML-DSA. We obtain end-to-end speedups of up to 1.32× for decapsulation (ML-KEM-512), 1.27× for encapsulation, and 1.17× for key generation, over the NIST PQC reference C implementations. For ML-DSA, the improvement is up to 1.26× for signing (ML-DSA-2), 1.16× for verification, and 1.11× for key generation.

Availability. Source code, examples, EasyCrypt proofs, and integrated implementations are available at https://anonymous.4open.science/r/pkc_anonymous_artifact-8F22.

2 Preliminaries

This section establishes the mathematical and algorithmic background required throughout the paper. We first define our notations used in the paper (Section 2.1), then introduce the Number Theoretic Transform and its butterfly operations (Section 2.2). We further review two commonly used modular reduction algorithms, namely Montgomery and Barrett reduction in lattice-based cryptography (Section 2.3), and next interpret the concepts on Plantard reduction and its use in our lazy-reduction strategy (Section 2.4). Finally, we describe the Jasmin and EasyCrypt toolchain (Section 2.5) and summarize the prior formal-verification effort on ML-KEM that our work builds upon (Section 2.6).

2.1 Notations

In this paper, polynomials are written in italic with the indeterminate made explicit (e.g. $a(x)$ or $A(x)$); we denote their corresponding coefficient vectors by

lowercase boldface symbols (e.g. $\boldsymbol{a}$). The word width ℓ denotes the smallest signed integer width $\ell \in \{16, 32, 64\}$ that fits the prime modulus q, and $\hat{n}$ denotes the effective transform size ($\hat{n} = n$ for a complete NTT, or the largest power-of-two divisor of n with $\omega^{\hat{n}} \equiv -1 \pmod{q}$ for incomplete). We use $\lfloor \cdot \rceil$ for rounding to the nearest integer and $\bmod^{\pm}$ for signed modular reduction. We denote $\lfloor X/2^k \rfloor$ as $[X]^k$ for the k high bits of X (truncating the k low bits), and $X \bmod 2^k$ as $[X]_k$ for the low k bits.

2.2 Number Theoretic Transform

Lattice-based schemes operate over the polynomial ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$, where q is a prime modulus and n is a power of two, and their performance is dominated by polynomial multiplication in R_q. One intuitive way to compute the product of two polynomials is through schoolbook multiplication: given two polynomials $a(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$ and $b(x) = b_0 + b_1 x + \cdots + b_{n-1} x^{n-1}$ where $a(x), b(x) \in R_q$, their product yields the polynomial $c(x) = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1}$ where each coefficient of $c(x)$ can be defined as:

$$c_i \;=\; \Big(\sum_{j+k=i} a_j b_k \;-\; \sum_{j+k=n+i} a_j b_k \Big) \bmod q. \tag{1}$$

This approach defined by (1) is the *Negative Wrapped Convolution* of polynomials $a(x)$ and $b(x)$, which incurs a computational complexity of $\mathcal{O}(n^2)$. Therefore, naive schoolbook multiplications can easily lead to significant overhead in modern lattice-based cryptography implementations.

As a special case of the Discrete Fourier Transform (DFT) that operates over finite fields, the Number Theoretic Transform (NTT) provides an efficient method to compute polynomial multiplications. By transforming polynomial representations into point-value representations [18], the NTT enables faster and cheaper point-wise multiplications rather than *Negative Wrapped Convolution* over two polynomials. Like DFT, NTT is driven by a twiddle factor ω, where ω is a primitive n-th root of unity in $\mathbb{Z}_q$ that satisfies $\omega^n \equiv 1 \pmod{q}$ and also $\omega^i \not\equiv 1 \pmod{q}$ for $0 < i < n$. The existence of such a root requires n to be a power of two, q to be prime, and $q \equiv 1 \pmod{n}$. Let $a(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$ be a polynomial, and let $\boldsymbol{a} = (a_0, \ldots, a_{n-1})$ be its coefficient vector of $a(x)$, where each coefficient $a_i \in \mathbb{Z}_q$. The forward NTT of vector $\boldsymbol{a}$, denoted by $\mathrm{NTT}(\boldsymbol{a})$, is the NTT-domain vector $\hat{\boldsymbol{a}} = (\hat{a}_0, \ldots, \hat{a}_{n-1})$, such that: $\hat{a}_j = \sum_{i=0}^{n-1} a_i \omega^{ij} \pmod{q}$, for $j = 0, 1, \ldots, n-1$. The inverse NTT, denoted as $\mathrm{INTT}(\hat{\boldsymbol{a}})$, transforms NTT-domain vector $\hat{\boldsymbol{a}}$ back to coefficient vector $\boldsymbol{a}$, where $a_i = n^{-1} \sum_{j=0}^{n-1} \hat{a}_j \omega^{-ij} \pmod{q}$, for $i = 0, 1, \ldots, n-1$. Therefore, we have the NTT round-trip $\boldsymbol{a} = \mathrm{INTT}(\mathrm{NTT}(\boldsymbol{a}))$. Consequently, the NTT enables efficient polynomial multiplication: point-wise multiplication in the NTT domain corresponds to polynomial multiplication in the original ring. Given polynomials A, B and their corresponding coefficient vectors $\boldsymbol{a}, \boldsymbol{b}$, to compute their product $C = A \cdot B$ with coefficient vector $\boldsymbol{c}$, one can compute

$$\boldsymbol{c} = \mathrm{INTT}(\mathrm{NTT}(\boldsymbol{a}) \odot \mathrm{NTT}(\boldsymbol{b})). \tag{2}$$

Polynomial multiplications can therefore be computed by multiplying two coefficient vectors in the NTT domain with a complexity of $\mathcal{O}(n)$. Provided that forward NTT and inverse NTT can be implemented in $\mathcal{O}(n \log n)$ time, the overall cost is reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ [13,30].

Efficient NTT implementations rely on the Chinese Remainder Theorem (CRT) factorization of R_q, which decomposes the polynomial ring into a product of smaller component rings. For lattice-based schemes operating over $R_q = \mathbb{Z}_q[X]/(X^n+1)$, we extend the primitive n-th root of unity ω to a primitive $2n$-th root of unity $\psi \in \mathbb{Z}_q$, satisfying $\psi^{2n} \equiv 1 \pmod{q}$ and $\psi^2 = \omega$. The existence of ψ now requires $q \equiv 1 \pmod{2n}$, and by construction, $\psi^n \equiv -1 \pmod{q}$. The polynomial ring can therefore be decomposed as $X^n + 1 = X^n - \psi^n = (X^{n/2} - \psi^{n/2})(X^{n/2} + \psi^{n/2}) = (X^{n/2} - \psi^{n/2})(X^{n/2} - \psi^{n/2+n}) = \ldots$, where each factor $X^m - \psi^l$ (for some m and l) can be recursively split into $X^{m/2} - \psi^{l/2}$ and $X^{m/2} - \psi^{l/2+n}$. We refer to each step of this recursive factorization as a *layer*. Through this recursive factorization, we obtain the complete factorization $X^n + 1 = \prod_{i=0}^{n-1}(X - \psi^{2i+1})$, which by CRT yields the ring isomorphism:

$$\mathbb{Z}_q[X]/(X^n+1) \;\cong\; \prod_{i=0}^{n-1} \mathbb{Z}_q[X]/(X - \psi^{2i+1}). \tag{3}$$

The implementation of each CRT split in the recursive factorization is achieved by *butterfly* operations. The forward NTT utilizes the *Cooley-Tukey (CT) butterfly* [13], the ring isomorphism

$$\phi\colon \mathbb{Z}_q[X]/(X^m - \psi^l) \;\to\; \mathbb{Z}_q[X]/(X^{m/2} - \psi^{l/2}) \;\times\; \mathbb{Z}_q[X]/(X^{m/2} - \psi^{l/2+n}). \tag{4}$$

Any polynomial $a(x) \in \mathbb{Z}_q[X]/(X^m - \psi^l)$ in degree-m form can be split into its low and high halves as $A(x)$ and $B(x)$:

$$a(x) = \underbrace{(a_0 + \cdots + a_{m/2-1}x^{m/2-1})}_{A(x)} + x^{m/2} \cdot \underbrace{(a_{m/2} + \cdots + a_{m-1}x^{m/2-1})}_{B(x)},$$

so $a(x) = A(x) + x^{m/2}B(x)$. Applied to this split form, the map ϕ is given by

$$A(x) + x^{m/2}B(x) \;\longmapsto\; \big(A(x) + \psi^{l/2}B(x),\; A(x) - \psi^{l/2}B(x)\big) \pmod{q}, \tag{5}$$

where the two output components are the *plus arm* $(A(x) + \psi^{l/2}B(x))$ and the *minus arm* $(A(x) - \psi^{l/2}B(x))$. Similarly, the inverse NTT employs the dual *Gentleman-Sande (GS) butterfly* [17], which inverts (5). Given a pair $(A'(x), B'(x))$ in the product ring on the right of (4), it recovers the parent representation

$$(A'(x), B'(x)) \;\longmapsto\; \big(\underbrace{\tfrac{1}{2}(A'(x) + B'(x))}_{A(x)},\; \underbrace{\tfrac{1}{2}\psi^{-l/2}(A'(x) - B'(x))}_{B(x)}\big) \pmod{q}, \tag{6}$$

yielding the polynomial $A(x) + x^{m/2}B(x)$ in $\mathbb{Z}_q[X]/(X^m - \psi^l)$. In real-world NTT applications, the accumulated factor $\frac{1}{2}$ across all layers can be absorbed

into a single final scaling by N^{-1}, where $N = 2^L = \hat{n}$ is the transform size. We drop these factors from the butterfly definition and use the simplified form

$$(A'(x), B'(x)) \longmapsto \big((A'(x) + B'(x)),\ \psi^{-l/2}(A'(x) - B'(x))\big) \pmod{q}, \quad (7)$$

where the two output components are the *plus arm* $(A'(x) + B'(x))$ and the *minus arm* $\psi^{-l/2}(A'(x) - B'(x))$ similar to CT butterfly.

For some lattice schemes, the recursion in (3) is stopped before reaching $\mathbb{Z}_q$ arithmetic. For example, ML-KEM uses modulus $q = 3329$, ring dimension $n = 256$, and operates over $R_q = \mathbb{Z}_q[X]/(X^{256} + 1)$. A primitive $2n$-th root of unity does not exist in $\mathbb{Z}_q$ since $q \not\equiv 1 \pmod{2n}$, yielding the incomplete decomposition:

$$\mathbb{Z}_q[X]/(X^{256} + 1) \ \cong \ \prod_{i=0}^{127} \mathbb{Z}_q[X]/(X^2 - \psi^{2i+1}). \quad (8)$$

The point-wise multiplication in (2) is now replaced by 128 multiplications over these quadratic rings, each computed by a *base multiplication* (basemul): given $(a_0 + a_1X), (b_0 + b_1X) \in \mathbb{Z}_q[X]/(X^2 - \psi^{2i+1})$, their product can be derived by

$$(a_0b_0 + a_1b_1\psi^{2i+1}) + (a_0b_1 + a_1b_0)X \pmod{q}. \quad (9)$$

In this paper, both the forward and inverse transforms refer to this incomplete variant.

2.3 Modular Arithmetic for Lattice-Based Cryptography

To keep NTT operations implemented within word boundaries, multiplications in both butterfly operations ($\psi^{l/2}B$ and $\psi^{-l/2}(A'-B')$) require a modular reduction to prevent coefficients from growing rapidly through subsequent additions and subtractions. Traditional implementation through recursive trivial division is less efficient for frequent use in NTT. Therefore, the cost of modular reduction dominates the overall NTT efficiency.

Montgomery reduction [25] Montgomery reduction replaces trial division with cheap bit-shift operations to compute the result $r = a \cdot R^{-1} \bmod q$ for a chosen radix $R = 2^\ell$ coprime to q. To cancel the factor of R^{-1} in the result, the field elements can be represented in *Montgomery form* $a' = aR \bmod q$. The product of *Montgomery multiplication* between two factors in *Montgomery form* aR, bR is then computed via *Montgomery reduction* as $(aR \cdot bR)R^{-1} \bmod q$, which remains in *Montgomery form*, avoiding frequent conversion. This property allows twiddle factors ψ^k to be precomputed once in *Montgomery form* as a table, so that the multiplication in each butterfly operation ($\psi^k \cdot B$ and $\psi^{-l/2}(A'-B')$) introduces a factor of R that the subsequent Montgomery reduction cancels, transforming the coefficient back to standard form.

Algorithm 1 Original Plantard Multiplication

Input: Unsigned integers $a, b \in [0, q]$, $q < \frac{2^\ell}{\varphi}$, $\varphi = \frac{1+\sqrt{5}}{2}$, $q' \equiv q^{-1} \bmod 2^{2\ell}$, where ℓ is the machine word size
Output: $r \equiv ab(-2^{-2\ell}) \bmod q$ where $r \in [0, q]$
1: $r \leftarrow \left[\left([\,[abq']_{2\ell}\,]^\ell + 1 \right) \cdot q \right]^\ell$
2: **return** r

Barrett reduction [7] Barrett reduction alternatively leverages the floor approximation on the quotient $\lfloor a/q \rfloor$ using a precomputed value $\mu = \lfloor 2^{2\ell}/q \rfloor$, so that $a \bmod q$ can be computed as $a - \lfloor a \cdot \mu / 2^{2\ell} \rfloor \cdot q$ by cheap bit-shifts. Note that the output of *Barrett reduction* is in standard form.

The reference implementations of ML-KEM [10], ML-DSA [6], and FN-DSA [16] all use Montgomery and Barrett reduction in their NTT pipelines. Montgomery handles multiplications on butterfly units with properly encoded twiddle factor and Barrett is more generally used since it returns standard form. There also exist some variants of two reductions introduced above; however, both Montgomery and Barrett reductions suffer from the issue with loose output bound ($[0, 2q)$ for Montgomery and $[0, 3q)$ for Barrett), which adds extra instructions for bound correction. Furthermore, they were originally designed for large moduli of RSA and elliptic-curve cryptography. Modern lattice-based schemes operate over much smaller word-sized moduli, typically 16–64 bits.

Plantard Reduction To address the problems discussed, *Plantard reduction* [29] was specifically designed for smaller word-sized modular arithmetic, performing the entire reduction within a single 2ℓ-bit word using only multiplications and bit-shifts. Algorithm 1 presents the original Plantard multiplication, which produces a result $r = ab \cdot (-2^{-2\ell}) \bmod q$ in $[0, q]$ for unsigned inputs in $a, b \in [0, q]$.

Note that the output carries an extra factor of $-2^{-2\ell}$, which we refer to as the *Plantard factor*, analogous to Montgomery's R^{-1}, which can be canceled by precomputed constants, in the same fashion as Montgomery. Moreover, the output bound of $[0, q]$ is substantially tighter than Montgomery's $[0, 2q)$ and Barrett's $[0, 3q)$, eliminating the subsequent bound correction steps. However, Algorithm 1 only accepts unsigned input and produces outputs in $[0, q]$, constraining its capability in NTT pipelines that produce signed intermediate values.

The extension of Plantard arithmetic to signed integers was first proposed by Aoki et al. [4]. Huang et al. [19] further introduced a signed variant and added a tunable factor of 2^α (see Algorithm 2), which enlarges the input range to $[-q2^\alpha, q2^\alpha]$ while maintaining the tight signed output bound $(-\frac{q}{2}, \frac{q}{2})$.

Huang et al. [20] subsequently revisited the correctness analysis of Algorithm 2 without modifying its algorithm structure. They incorporated the $\alpha > 0$ correction identified by Yang et al. [34] (refining the original $\alpha \geq 0$ constraint in [19]) and bound the product of inputs a, b by an aggregated range as $ab \in [q2^\ell - q2^{\ell+\alpha}, 2^\ell - q2^{\ell+\alpha})$, while the output remains in $[-\frac{q+1}{2}, \frac{q}{2})$. For ML-KEM

Algorithm 2 Improved Plantard Multiplication [19]

Input: Two signed integers $a, b \in [-q2^{\alpha}, q2^{\alpha}]$, $q < 2^{\ell-\alpha-1}$, $q' \equiv q^{-1} \bmod^{\pm} 2^{2\ell}$
Output: $r \equiv ab(-2^{-2\ell}) \bmod^{\pm} q$ where $r \in (-\frac{q}{2}, \frac{q}{2})$

1: $r \leftarrow \left[\left(\left[[abq']_{2\ell} \right]^{\ell} + 2^{\alpha} \right) \cdot q \right]^{\ell}$
2: **return** r

with $q = 3329$ and $\ell = 16$, the maximum admissible α is 3, and the input range becomes $a \in [-137q, 230q]$ when $b = b_{max} = 3328$, at least 2.14× larger than the bound proved in [19]. They use this larger input range and achieve a more aggressive lazy reduction in NTT pipeline by removing all reductions from forward transform and most from the inverse. They target Cortex-M3 and SiFive RISC-V, and achieve a speedup at 26.19%/32.57% (NTT/INTT) on Cortex-M3 and 34.76%/55.54% (NTT/INTT) on SiFive RISC-V against Montgomery and Barrett based implementations. We adopt this enlarged range as the input precondition throughout our verification and lazy reduction analysis.

2.4 Plantard NTT and Lazy Reduction

The signed Plantard reduction of Algorithm 2 with enlarged input range [20] can now replace Montgomery as the arithmetic bridge in each butterfly multiplication. Twiddle factors ψ^k are encoded with a factor of $-2^{2\ell}$ in the precomputation phase, and multiplying a coefficient B by a precomputed ψ^k through Plantard multiplication reduction can cancel the encoding, yielding $\psi^k \cdot B \bmod^{\pm} q$ directly in standard form. The tighter output bound $r \in [-\frac{q+1}{2}, \frac{q}{2})$ makes Plantard more effective at eliminating reductions in an NTT pipeline: the bound on each reduced coefficient is at most half the size of those produced by Montgomery (outputs in $[0, 2q)$) or Barrett (outputs in $[0, 3q)$). The naturally centered output also balances the coefficient accumulation across both sides of the signed word range, instead of growing only on the positive side with unsigned settings.

This tighter bound suggests that the reduction on every butterfly addition or subtraction is not always necessary, which we refer to as *lazy reduction* [31]. For a CT-butterfly operation $A \mapsto (A + \psi^k B,\ A - \psi^k B)$, the term $\psi^k B$ is always bounded by Plantard reduction output bounds, and the range for A is determined by the bound produced by the previous layer. We check whether the resulting sum falls within both the signed word range $[-2^{\ell-1}, 2^{\ell-1})$ to avoid overflow, and Plantard's input precondition of $[q2^{\ell} - q2^{\ell+\alpha}, 2^{2\ell} - q2^{\ell+\alpha})$ required for the next butterfly's product. Coefficients are reduced when either constraint is violated.

2.5 Jasmin and EasyCrypt

In response to the needs of high assurance and high efficiency in cryptographic implementations, *Formosa Crypto* was developed, which includes two complementary tools: *Jasmin* [2] and *EasyCrypt* [8].

Jasmin Jasmin is a programming language and compiler designed for high-assurance and high-performance cryptographic implementations. Its source language combines the high-level control flows (loops, structured types) and abstractions (functions, arrays) with low-level access to instructions and registers, allowing programmers to write structured code while retaining control over its assembly code. The Jasmin compiler is formally verified in *Coq* [14] (now called Rocq), preserving source semantics down to the generated assembly.

EasyCrypt EasyCrypt is an interactive proof assistant designed for the formal verification of cryptographic constructions. Its library supports various tactics for both deterministic and probabilistic programs across program logics, including Hoare logic for program equivalence proofs. `jasmin2ec` serves as a significant extraction process to bridge the two frameworks by producing EasyCrypt model from Jasmin function exports, allowing the extracted procedure to be proved equivalent to an abstract specification in EasyCrypt.

2.6 Prior Formal Verification of NTT

The closest predecessor to our work is the *formosa-mlkem* project [3], which provides formally verified end-to-end implementations of ML-KEM in Jasmin and EasyCrypt, including NTT.

Their work demonstrates a pipeline that writes the implementation in Jasmin, extracts an EasyCrypt model via `jasmin2ec`, and develops EasyCrypt specification referring to these extracted models syntactically close to the FIPS-203 pseudocode to demonstrate functional correctness and equivalence. Specifically for NTT, its correctness is established by a chain of program equivalences from the extracted code down to the abstract specification: $\mathsf{NTT}_{\mathsf{Jasmin}} \Leftrightarrow \mathsf{NTT}_{\mathsf{Fq}} \Leftrightarrow \mathsf{NTT}_{\mathsf{opt}} \Leftrightarrow \mathsf{NTT}_{\mathsf{bsrev}} \Leftrightarrow \mathsf{NTT}_{\mathsf{naive}} \Leftrightarrow \mathsf{ntt_spec}$.

$\mathsf{NTT}_{\mathsf{Jasmin}}$ contains the raw NTT model in EasyCrypt extracted from Jasmin. $\mathsf{NTT}_{\mathsf{Fq}}$ replaces each procedure call (e.g. Montgomery multiplication, Barrett reduction) in $\mathsf{NTT}_{\mathsf{Jasmin}}$ by its mathematical specification over $\mathbb{F}_q$. $\mathsf{NTT}_{\mathsf{opt}}$ decomposes the NTT nested-loop structure into several small functions, enabling modular reasoning. $\mathsf{NTT}_{\mathsf{bsrev}}$ replaces the incremental counter used to access array of roots of unity by the bit-reversal of the value computed directly from loop indices, and substitutes the innermost loop index variable by its final value where it is used after the loop. $\mathsf{NTT}_{\mathsf{naive}}$ reorders the two innermost loops with respect to $\mathsf{NTT}_{\mathsf{bsrev}}$, and replaces the array accesses to roots of unity by direct computation of the desired root. $\mathsf{ntt_spec}$ is the abstract mathematical specification of the NTT, expressed directly as a polynomial evaluation map. Each stage decomposes one layer further of implementation detail until it is structurally identical to the naive mathematical definition of NTT, which is itself an instance of the DFT over $\mathbb{F}_q$.

The formosa-mlkem development formalizes Montgomery reduction as a standalone EasyCrypt theory, imported into the $\mathbb{F}_q$ theory and used throughout the NTT correctness proof chain. This theory establishes the algebraic correctness of signed Montgomery reduction (`SREDC`) based on pen-and-paper proof,

encoding both the functional equivalence to multiplication by R^{-1} modulo q and the reasoning from input range to output range. The NTT correctness proof relies on this theory as an arithmetic bridge in butterfly operation. Every call to the Jasmin Montgomery procedure `__fqmul` is replaced by `SREDC` via a Hoare-logic specification (`fqmul_corr` in [3, Section 5.1]). Therefore, the NTT proof reasoning can be performed entirely at $\mathbb{F}_q$ level.

More recently, Trieu [33] demonstrated another end-to-end parametric NTT specification in Rocq based on Fiat-Crypto code generation. Their formalization (Section 2) abstracts the coefficient ring R, a primitive root of unity ζ and the recursion depth r, which is sufficient to support both complete and incomplete NTT transforms. Therefore, the formalization can be reused across schemes such as ML-KEM and ML-DSA by instantiating the abstract parameters with scheme-specific values. The specification recursively decomposes the polynomial ring using Chinese Remainder Theorem (CRT) (Theorem 1 of [33]), and its correctness can be ensured by the ring isomorphisms derived at each decomposition. The verified C implementations are then extracted through the Bedrock2 backend of Fiat-Crypto [15]. The embedded signed Barrett and Montgomery arithmetic are formally proved correct as the underlying reduction layer (Section 4.2 in [33]). However, they explicitly deferred the Plantard arithmetic and lazy reduction as future work and haven't formalized them in their current codebase (Section 7 in [33]).

3 Generating Plantard NTT with Lazy Reduction

To address the gaps discussed in Section 2.6, in this section, we present *Lazy-Generator*, an automated Python-based NTT code generator that produces forward and inverse NTT pipelines based on Plantard arithmetic. We will begin with a high-level overview of the generator framework (Section 3.1). We then interpret its two core components: the *bound analyzer* (Section 3.2), which tracks the coefficient growth and avoids overflow via bound analysis, and the *code generator* (Section 3.3) that produces NTT code in two backends (C and Jasmin).

3.1 Overview

We introduce *Lazy-Generator*[3], a Python-based code generator for forward and inverse NTT routines. It targets small-moduli NTT implementations such as those commonly used in lattice-based cryptography, using Plantard arithmetic with lazy reduction. This tool is fully automated and requires minimal parameter description as input. The complete pipeline is shown in Figure 1, with each component described in turn in the remainder of this section.

We notice that shaping a well-structured NTT only requires modulus q and ring dimension n, and a primitive root ω. We therefore built a front-end terminal-based interface that takes the parameter triple (q, n, ω) as input and validates

[3] Source code and the relevant artifact are available at https://anonymous.4open.science/r/pkc_anonymous_artifact-8F22

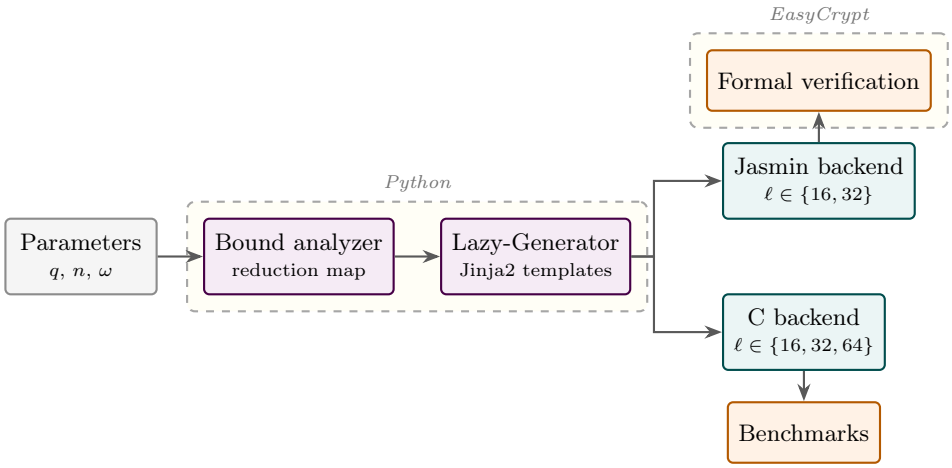

Fig. 1. Pipeline for NTT generator. The triple (q, n, ω) feeds to Bound analyzer, which produces a per-layer reduction map driving the Jinja2 templates. The templates dispatch on word size $\ell \in \{16, 32, 64\}$ to a Jasmin or C backend; Jasmin supports $\ell \in \{16, 32\}$ while $\ell = 64$ requires `__int128` and is C-only. The Jasmin backend is extracted to EasyCrypt via `jasmin2ec` for formal verification (Sec. 4), while the C backend is compiled and used to obtain the cycle-count benchmarks reported in Sec. 5. Dashed boxes indicate the language or proof framework each component lives in.

that it can produce a well-defined NTT: q must be prime, n is a power of two, and ω either an n-th or $2n$-th root of unity modulo q. The front-end pre-validation process automatically detects the resulting NTT type: $\omega^n \equiv -1 \pmod{q}$ yields a complete NTT, while $\omega^n \equiv 1 \pmod{q}$ yields an incomplete NTT. From the concrete value of q, we could trivially determine the minimum word size $\ell \in \{16, 32, 64\}$ satisfying $q < 2^{\ell-1}$.

Recall Huang et al. [20] showed that Plantard reduction takes two signed integers a, b whose product ab needs to satisfy the asymmetric range: $ab \in \left[q2^{\ell} - q2^{\ell+\alpha},\ 2^{2\ell} - q2^{\ell+\alpha} \right)$, where α is a positive integer ($\alpha > 0$), and free to be chosen in implementation. Note that the width of the interval above is fixed by ℓ and q alone without α, which gives $2^{2\ell} - q \cdot 2^{\ell}$. What α controls is its *position* on the real line. As α grows, the envelope gradually shifts left, exposing increasingly more negative magnitude at the cost of positive range. Since the modulus q is bounded by $q < 2^{\ell-\alpha-1}$, we get $\alpha < \ell - \log_2 q - 1$. Therefore, there exists a maximum value $\alpha_{max} = \ell - \lceil \log_2 q \rceil - 1$ that can position the envelope as symmetrically around zero as the precondition allows.

With the parameter triple validated and the derived parameters ℓ, $\alpha_{\max}$ in place, the NTT layer structure is sufficiently explicit to be specified in two further stages: the *bound analyzer* (Section 3.2) determines the layer-by-layer specification of the NTT and inserts the necessary reductions to maintain safe intermediate value, and the *code generator* (Section 3.3) derives the remaining

backend-specific parameters and renders them through the corresponding C or Jasmin template for eventual code output.

3.2 Bound analyzer

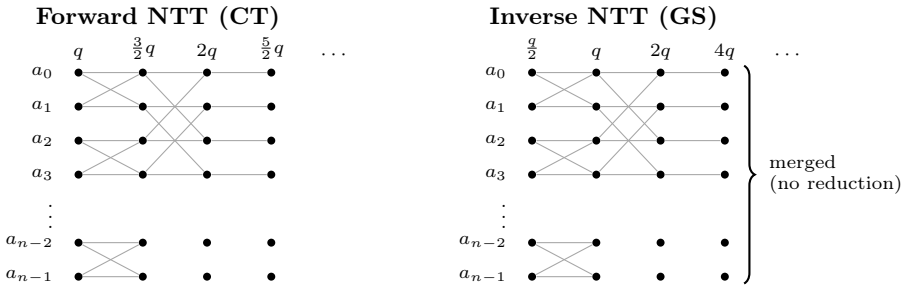

Fig. 2. Coefficient bound growth across NTT and INTT layers under Plantard arithmetic. CT butterflies (left) grow the bound by $q/2$ per layer; GS butterflies (right) double it. Each merged-layer block is the maximal sequence of layers between successive modular reductions, determined statically by our bound analyzer.

Placing modular reductions in an NTT implementation is an optimization problem under a strict correctness constraint. The correctness requires every intermediate coefficient to stay within the signed word bound and Plantard's input bound to ensure the next layer's correctness. Omitting reductions makes the implementation faster, but the bounds need careful maintenance. There exist multiple optimized reference implementations that can resolve this problem manually. Huang et al. provided lazy-reduction strategies for ML-KEM on Cortex-M3 and RISC-V layer by layer [20]. Becker et al. [9] manually fuse Montgomery and Barrett reductions into a custom "Barrett multiplication" operation, with reduction placement tuned across schemes based on informal bound analysis. Therefore, the resulting effort does not transfer between schemes. The manual tuning is required for each new modulus, word size, and merging strategy (e.g. bound analysis for ML-DSA differs from ML-KEM's, and FN-DSA's again differs).

Our *bound analyzer* is a core embedded component of the proposed generator. Given a parameter set $(q, n, \omega, \ell, \alpha)$, it determines the position to insert the reduction in a generated NTT routine. The result is a per-layer reduction map, which will be subsequently consumed by the code generator. Reduction points are determined entirely at code-generation time and depend only on the parameter set given, resulting in structurally identical C and Jasmin NTT code. The generated source therefore avoids branches on coefficient values, which is a standard requirement to prevent cryptographic applications from side-channel attacks. For the Jasmin backend this property carries to the compiled program; for the C backend it holds of the source, and its preservation in the binary depends on the compiler (Section 6).

Coefficient Growth Model The proposed bound analyzer tracks the worst-case absolute value of any coefficient that the layer produces. We denote the bound derived after layer i as B_i for $i = 1, \ldots, L$ where L is the total number of layers in the generated NTT, with B_0 denoting the bound on the input to the first butterfly layer. Recall the Cooley–Tukey (CT) butterfly computation: $(a, b) \mapsto (a + \omega b,\ a - \omega b)$, where ω is a precomputed twiddle factor. The product ωb is always passed through Plantard multiplication reduction; its value is therefore always bounded by Plantard reduction postcondition, that is, $\omega b \in [-\frac{q+1}{2}, \frac{q}{2})$. For the odd modulus q used in most of the schemes considered, this interval contains exactly the integers $\{-\frac{q+1}{2}, \ldots, \frac{q-1}{2}\}$, equivalently $|\omega b| \leq \frac{q-1}{2} < \frac{q}{2}$. Therefore, given input bounds $|a| \leq B_{i-1}$ and $|b| \leq B_{i-1}$, the butterfly plus arm's output satisfies $|a + \omega b| \leq |a| + |\omega b| < B_{i-1} + \frac{q}{2}$, and similarly for $|a - \omega b|$. The post-layer bound after layer i is $B_i^{\mathrm{CT}} = B_{i-1} + \frac{q}{2}$. After i layers without reduction, $B_i = B_0 + i \cdot \frac{q}{2}$. The bound therefore grows linearly with the layer index i. Figure 2 (left) shows the linear CT growth across several layers.

The Gentleman–Sande (GS) butterfly utilizes the simplified form as interpreted in Section 2.2: $(a, b) \mapsto (a + b,\ \omega(a - b))$, where ω is again a precomputed twiddle factor. The two arms of the GS butterfly operate asymmetrically: the plus arm $a + b$ is a sum of two unreduced inputs; given input bounds $|a|, |b| \leq B_{i-1}$, we have $|a + b| \leq |a| + |b| \leq 2B_{i-1}$. The bound is therefore doubled after each computation. The minus arm $\omega(a - b)$ passes through Plantard multiplication reduction; its bound is determined by the same postcondition $|\omega(a - b)| < \frac{q}{2}$, independent of B_{i-1}. The bound on the minus arm thus never accumulates, since it is reduced at every layer. Therefore, the plus arm dominates, so the bound after layer i is $B_i^{\mathrm{GS}} = 2B_{i-1}$. After i layers without reduction, $B_i = 2^i \cdot B_0$, which grows exponentially with the layer index i. The right panel of Figure 2 shows the analogous GS exponential growth. The inverse NTT therefore exhausts the word-size range faster than the forward NTT. Whether this actually forces a reduction depends on the concrete parameters: the bound must cross $2^{\ell-1} - 1$ within the L layers available, which is a function of q and ℓ rather than of the growth rate alone.

Reduction Insertion The analyzer encodes the bound analysis result as an array $[\texttt{plus_arm}, \texttt{minus_arm}]$ with $\texttt{plus_arm}, \texttt{minus_arm} \in \{0, 1\}$. The analyzer flags the corresponding entry as 1 on the butterfly arm whose post-butterfly bound B_i would exceed the signed ℓ-bit storage range: $B_i > 2^{\ell-1} - 1$. Note that for CT butterfly, both arms grow symmetrically, so flags only appear as either $[1, 1]$ or $[0, 0]$. For GS butterfly, the minus arm is already reduced by Plantard multiplication reduction (Section 3.2), so the analyzer only considers the plus arm and outputs either $[1, 0]$ or $[0, 0]$.

Section 2.2 establishes that every Plantard multiplication in the NTT pipeline takes the form $c \cdot \omega$, where c is a coefficient and ω is the precomputed twiddle factor. In implementation, $c \cdot \omega$ corresponds to the term $\psi^{l/2} B(x)$ shared by both arms of CT butterfly (5), and the minus arm $\psi^{-l/2}(A'(x) - B'(x))$ in GS butterfly (6). In such case, the analyzer necessarily ensures the product satisfies

Plantard's input precondition at every layer, while keeping coefficients within the signed ℓ-bit storage range. Checking both conditions separately at every butterfly would double the analyzer's work. We show in Lemma 1 that this is unnecessary. Here the bound we discussed for ω is already in the worst-case, subsuming its signed centered representation, that is $|\omega| \leq \frac{q-1}{2}$. There is therefore no need to discuss signed and unsigned settings separately.

Lemma 1 (Storage subsumption). *Let $q \geq 1$, $\alpha > 0$ be integers and ℓ a positive integer with $q < 2^{\ell-\alpha-1}$. For any integers c, ω with $|c| \leq 2^{\ell-1} - 1$ and $|\omega| \leq q - 1$, the product $c \cdot \omega$ lies within Plantard's input range $\left[q \cdot 2^{\ell} - q \cdot 2^{\ell+\alpha},\ 2^{2\ell} - q \cdot 2^{\ell+\alpha}\right)$.*

Proof. The preconditions stated on $|c|$ and $|\omega|$ give the worst-case product bound

$$|c \cdot \omega| \leq (2^{\ell-1} - 1)(q - 1) < 2^{\ell-1} \cdot q. \tag{10}$$

We then show the product $c \cdot \omega$ lies within the input range on both sides.
Upper bound. We require $c \cdot \omega < 2^{2\ell} - q \cdot 2^{\ell+\alpha}$, equivalently $2^{2\ell} - q \cdot 2^{\ell+\alpha} - c \cdot \omega > 0$. Using (10): $\left(2^{2\ell} - q \cdot 2^{\ell+\alpha}\right) - c \cdot \omega > 2^{2\ell} - q \cdot 2^{\ell+\alpha} - 2^{\ell-1} \cdot q = 2^{2\ell} - q \cdot \left(2^{\ell+\alpha} + 2^{\ell-1}\right)$. By Plantard's modulus precondition $q < 2^{\ell-\alpha-1}$: $2^{2\ell} - q \cdot \left(2^{\ell+\alpha} + 2^{\ell-1}\right) > 2^{2\ell} - 2^{\ell-\alpha-1} \cdot \left(2^{\ell+\alpha} + 2^{\ell-1}\right) = 2^{2\ell-1} - 2^{2\ell-\alpha-2}$. For any $\alpha > 0$, $2^{2\ell-\alpha-2} \leq 2^{2\ell-3} < 2^{2\ell-2}$. Therefore, $\left(2^{2\ell} - q \cdot 2^{\ell+\alpha}\right) - c \cdot \omega > 2^{2\ell-2} > 0$, establishing the upper bound.
Lower bound. We require $c \cdot \omega - (q \cdot 2^{\ell} - q \cdot 2^{\ell+\alpha}) \geq 0$. Using (10): $c \cdot \omega - \left(q \cdot 2^{\ell} - q \cdot 2^{\ell+\alpha}\right) > -2^{\ell-1} \cdot q - q \cdot 2^{\ell} + q \cdot 2^{\ell+\alpha} = q \cdot \left(2^{\ell+\alpha} - 2^{\ell} - 2^{\ell-1}\right)$. The factor in bracket satisfies $2^{\ell+\alpha} - 2^{\ell} - 2^{\ell-1} = 2^{\ell} \cdot \left(2^{\alpha} - \frac{3}{2}\right)$. For any $\alpha > 0$, $2^{\alpha} \geq 2 > \frac{3}{2}$, so the factor is positive. Therefore, $c \cdot \omega - \left(q \cdot 2^{\ell} - q \cdot 2^{\ell+\alpha}\right) > 0$, establishing the lower bound. □

The analyzer thus only needs to track the signed ℓ-bit storage range instead of checking the Plantard input precondition separately.

The Analyzer Algorithm We combine the techniques demonstrated above into the aggregated analyzer procedures: AnalyzeNTT for forward NTT (CT butterflies) and AnalyzeINTT for inverse NTT (GS butterflies). Both procedures operate recursively over L layers, where L can be computed entirely through input triple (q, n, ω). If $\omega^n \equiv -1 \pmod{q}$, the transform is complete and the effective transform size is $\hat{n} = n$. In contrast, if $\omega^n \equiv 1$, the transform is incomplete, and $\hat{n}$ is the largest power-of-2 divisor of n satisfying $\omega^{\hat{n}} \equiv -1 \pmod{q}$. In both cases, $L = \log_2 \hat{n}$. Each procedure inherits the coefficient growth model, and produces the reduction map, flagging each layer whenever an overflow on the storage range $2^{\ell-1} - 1$ occurs. The two procedures are structurally identical but differ in their growth recurrence and reduction pattern.

Note that the post-reduction state on line 7 of Algorithm 4 is set to $q - 1$ rather than $\lfloor q/2 \rfloor$, because the reduction on the minus arm always happens after the plus arm: the reduction on layer i is immediately followed by the doubling in

Algorithm 3 AnalyzeNTT

Input: q, ℓ, L
Output: $rmap \in \{0,1\}^{L\times 2}$

$B \leftarrow q$
$S \leftarrow 2^{\ell-1} - 1$
for $i \leftarrow 1$ **to** L **do**
  $B \leftarrow B + \lfloor q/2 \rfloor$ ▷ CT growth
  if $B > S$ **then**
    $rmap[i] \leftarrow [1, 1]$
    $B \leftarrow \lfloor q/2 \rfloor$ ▷ post-reduction
  else
    $rmap[i] \leftarrow [0, 0]$
  end if
end for
return $rmap$

Algorithm 4 AnalyzeINTT

Input: q, ℓ, L
Output: $rmap \in \{0,1\}^{L\times 2}$

$B \leftarrow q$
$S \leftarrow 2^{\ell-1} - 1$
for $i \leftarrow 1$ **to** L **do**
  $B \leftarrow 2B$ ▷ GS growth
  if $B > S$ **then**
    $rmap[i] \leftarrow [1, 0]$
    $B \leftarrow q - 1$ ▷ post-reduction
  else
    $rmap[i] \leftarrow [0, 0]$
  end if
end for
return $rmap$

Fig. 3. The bound analyzer. Both procedures take the modulus q, word size ℓ, and layer count L; both return a per-layer reduction map. Differences between left and right algorithms: CT growth is additive ($+\lfloor q/2 \rfloor$), GS growth is multiplicative ($\times 2$); CT reduces both arms when triggered ($[1, 1]$), GS reduces only the plus arm ($[1, 0]$) since the minus arm passes through Plantard's multiplication and is already in $(-q/2, q/2)$.

layer $i+1$. Setting $B \leftarrow q-1$ can precisely capture this post-doubling magnitude $2\lfloor q/2 \rfloor$ in advance, so that the next layer's growth step $B \leftarrow 2B$ correctly reflects the worst-case bound entering layer $i+2$ rather than causing overflow at layer $i+1$. This one-layer prediction keeps the storage check aligned with the actual point at which a Plantard precondition violation could arise.

Specifically, for ML-KEM forward NTT ($q = 3329$, $\ell = 16$, $L = 7$), we assume the coefficient bound is tuned and starts at $B_0 = q$, growing by $\lfloor q/2 \rfloor = 1664$ per layer and reaching at most $9 \cdot \lfloor q/2 \rfloor \approx 4.5q \approx 14981$ after seven layers forward NTT. The resulting bound remains well below the storage limit $2^{15} - 1 = 32767$. So the forward NTT requires no reduction according to the analyzer. For the inverse NTT in ML-KEM, the analyzer inserts a single reduction at layer 3 to avoid the following overflow at the plus arm in layer 4. Both analysis results align with Huang et al.'s hand-derived ML-KEM NTT strategy [20]. The same analyzer can be instantiated with different parameter tuples from multiple lattice-based schemes' NTTs, including ML-DSA, and FN-DSA without modification. The reduction map it produces is consumed by the code generator (Section 3.3), which generates the corresponding NTT and INTT implementations.

3.3 Code Generator

Our code generator is the core component that derives remaining parameters needed for code generation and consumes the reduction map produced by ana-

Algorithm 5 Forward NTT (Cooley–Tukey)

Input: Polynomial $a = (a_0, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$; twiddle table $\zeta[0..\hat{n}-1]$ in bit-reversed order; bottom-layer degree d
Output: a overwritten with its NTT

$k \leftarrow 0$
for $len = n/2,\ n/4,\ \ldots,\ d$ **do** ▷ NTT Layer
  for $start = 0$ **to** $n-1$ **step** $2 \cdot len$ **do**
    $k \leftarrow k + 1$
    $\omega \leftarrow \zeta[k]$
    for $j = start$ **to** $start + len - 1$ **do**
      $t \leftarrow \omega \cdot a[j + len] \bmod q$ ▷ CT butterfly
      $a[j + len] \leftarrow a[j] - t$ ▷ Minus arm: insert `plantard_mul_reduction`
      $a[j] \leftarrow a[j] + t$ ▷ Plus arm: insert `plantard_mul_reduction`
    end for
  end for
end for

lyzer (Sec. 3.2). Utilizing these parameters, code generator emits per-layer specialized NTT/INTT procedures targeting two backends: portable C and Jasmin [2] for verifiable deployment. These backends are produced from separate Jinja2 templates `ntt_template_c.jinja2` for C and `ntt_template.jinja2` for Jasmin sharing an identical structure differing in target platform syntax. The generated routines therefore share control-flow structure across targets.

Although the analyzer determines the reduction map, the templates still require several further generation-time constants. Most of the remaining parameters can be determined trivially: the storage tier $\ell \in \{16, 32, 64\}$ is the smallest word size with $q < 2^{\ell-1}$, the Plantard parameter is taken at its maximum $\alpha = \ell - \lceil \log_2 q \rceil - 1$,[4] and $q^{-1} \bmod 2^{2\ell}$ is the precomputed inverse used by Algorithm 2 from Section 2.3. The one derivation that materially shapes the emitted code is the number of layers, which follows from the NTT type.

Recall from Section 2.2 that NTT includes two types: complete NTT and incomplete NTT. Both of them are supported by our generator and type is determined by ω. If $\omega^n \equiv -1 \pmod q$ the transform is complete and the effective transform size is $\hat{n} = n$; otherwise $\omega^n \equiv 1$, the transform is incomplete, and $\hat{n}$ is the largest power-of-two divisor of n for which $\omega^{\hat{n}} \equiv -1$, leaving residue polynomials of degree $d = n/\hat{n}$ at the bottom layer. In either case the number of layers is $L = \log_2 \hat{n}$, which resolves the loop count for the template. The same $\hat{n}$ fixes the size of the twiddle table. Each entry pre-encodes $\omega^{\mathrm{br}(i)}$ into Plantard form via $\zeta_i = \big(\big(\omega^{\mathrm{br}(i)} \cdot (-2^{2\ell}) \bmod q\big) \cdot q^{-1}\big) \bmod^{\pm} 2^{2\ell}$, where $x \bmod^{\pm} m$ centres a residue in $(-m/2, m/2]$. Given the coefficient a_i is always within word size bound $(-2^{\ell-1}, 2^{\ell-1}]$, the runtime product $a_i \cdot \zeta_i$ directly produces the Plantard inner term $[a_i \omega_i q']_{2\ell}$, which is exactly what `plantard_mul_reduce` consumes

[4] Equivalently $\alpha = \ell - \lfloor \log_2 q \rfloor - 2$, which is the form used in the implementation; the two expressions (notice the change from ceiling to floor around $\log_2 q$) match because q is prime and therefore not a power of two, so $\lceil \log_2 q \rceil = \lfloor \log_2 q \rfloor + 1$.

Algorithm 6 Inverse NTT (Gentleman–Sande)

Input: NTT-domain polynomial a; twiddle table $\zeta[0..\hat{n}-1]$; bottom-layer degree d; scaling constant $\hat{n}^{-1} \bmod q$
Output: a overwritten with its inverse NTT

1: $k \leftarrow \hat{n} - 1$
2: **for** $len = d,\ 2d,\ \ldots,\ n/2$ **do** ▷ INTT Layer
3: **for** $start = 0$ **to** $n - 1$ **step** $2 \cdot len$ **do**
4: $\omega \leftarrow -\zeta[k];\ k \leftarrow k - 1$
5: **for** $j = start$ **to** $start + len - 1$ **do**
6: $t \leftarrow a[j] + a[j + len]$ ▷ Plus arm: insert `plantard_mul_reduction`
7: $a[j + len] \leftarrow \omega \cdot (a[j] - a[j + len]) \bmod q$ ▷ Minus arm, always reduced
8: $a[j] \leftarrow t$ ▷ GS butterfly
9: **end for**
10: **end for**
11: **end for**
12: **for** $j = 0$ **to** $n - 1$ **do** ▷ final scaling
13: $a[j] \leftarrow \hat{n}^{-1} \cdot a[j] \bmod q$
14: **end for**

as input. The analyzer's storage check on coefficient a_i subsumes Plantard's precondition (Section 3.2), so no further bound check is required. The inverse-NTT scaling constant $\hat{n}^{-1} \bmod q$ is encoded the same way.

We refer to the loop count as the NTT and INTT layer count, shown on line 2 of Algorithms 5 and 6. In our template, the outer `for` loop (line 2 in Algorithms 5 and 6) is unrolled at code-generation time into L specialized procedures $\mathsf{ntt}_1, \ldots, \mathsf{ntt}_L$(and likewise $\mathsf{intt}_1, \ldots, \mathsf{intt}_L$ for the inverse NTT). Each procedure embeds its layer-specific *len* (line 2 in both algorithms) and k (line 1 and line 4 in both algorithms) as compile-time constants, and the flagged arm in reduction map emitted by analyzer is realized as an explicit `plantard_mul_reduction`; unflagged arms remain unreduced (see line 8, 9 in Algorithm 5, and line 6 in Algorithm 6). Since these decisions are handled within code generation, the generated procedures contain no `if`-statement on the layer index at run-time, which follows the constant-time cryptographic implementation rules.

Together with the bound analyzer of Section 3.2, the code generator forms a complete pipeline: given a parameter triple (q, n, ω), it produces specialized NTT and INTT implementations[5] whose lazy-reduction strategy is derived cautiously by the analyzer's bound tracking methodology. The C backend supports $\ell \in \{16, 32, 64\}$ while Jasmin backend only supports $\ell \in \{16, 32\}$ due to its own compiler constraint, sufficient for ML-KEM, ML-DSA, and FN-DSA. Section 4 then takes the Jasmin output for ML-KEM through formal verification.

[5] The generator produces forward and inverse transforms only; schemes using NTT-based polynomial multiplication must supply a basemul routine for the bottom-layer residue separately.

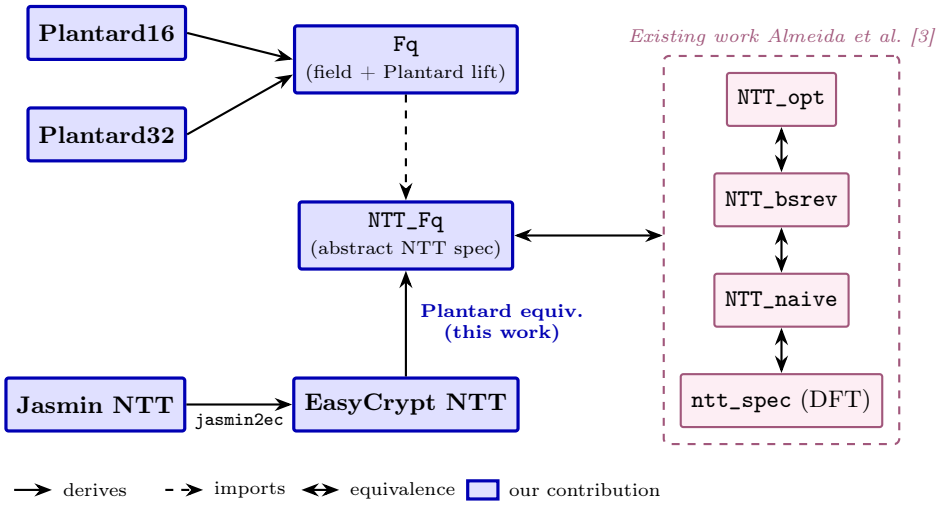

Fig. 4. Verification framework for the Plantard ML-KEM NTT. The left column formalizes Plantard arithmetic at two word widths and lifts it through `Fq.ec` into the abstract NTT specification `NTT_Fq`. The Jasmin implementation is derived by *LazyGenerator* and extracted via `jasmin2ec`, and the program-equivalence proof between this extracted Jasmin NTT and `NTT_Fq` is the core proof of this work. From `NTT_Fq`, the existing chain of Almeida et al. [3] discharges equivalence to the mathematical DFT specification. Bold blue boxes mark our contributions.

4 Formally Verified ML-KEM NTT

We now demonstrate our verification methodology, a machine-checked equivalence between Jasmin NTT routine implementations produced by our generator (Section 3.3) and the abstract NTT layer. We inherit the general verification framework and the abstract NTT specification developed by Almeida et al. [3] as the endpoint. In place of Montgomery, we present our formalization of Plantard arithmetic field operations, and a program-equivalence proof bridging the Jasmin implementation to the abstract specification. Once the NTT equivalence to abstract layer is established, we then combine our work with their algebraic proof chain (Section 5.3 in [3]): `NTT_Fq.NTT.ntt` ⇔ `NTT_opt` ⇔ `NTT_bsrev` ⇔ `NTT_naive` ⇔ `ntt_spec` ⇔ `ntt`, so that we can show our generated NTT is equivalent to its mathematical definition.

Figure 4 summarizes the verification framework. Our contributions, marked in bold blue, consist of the Plantard arithmetic theories `Plantard16` and `Plantard32`, their lift into the field theory `Fq`, the Jasmin NTT generated by our toolchain, and the equivalence proof from the extracted EasyCrypt program to the abstract specification defined in `NTT_Fq`. The remaining chain on the right is reused from Almeida et al. [3] unchanged. We first introduce the Plantard formalization (Section 4.1), which establishes the field-level reasoning principles used by the

implementation. We then present the program-equivalence proof (Section 4.2) connecting the extracted Jasmin NTT to the abstract specification in `NTT_Fq`.

4.1 Plantard formalization

We parameterize ML-KEM's constants and formalize the Plantard arithmetic as a parametric EasyCrypt theory `Plantard16` (Listing 1.1) and `Plantard32`. Since our scope focuses on ML-KEM, and the only difference between `Plantard16` and `Plantard32` is the minimum word size as the theory names suggest, we only discuss the 16-bit Plantard arithmetic here. The theory abstracts over a modulus `q`, its inverse `qinv` modulo `R2`, the tunable Plantard constant exponent `plan_alpha`, and a constant `plan_const`, with axioms encoding their correctness conditions following Huang et al. [20] as follows: the modulus bound `q < 2^(L - plan_alpha - 1)`, oddness of `q`, the correctness of `qinv` as a multiplicative inverse modulo `R2`, and the relation `(plan_const * R2) %% q = (-1) %% q`. The word width `L` is fixed at `16`, and `R = 2^L`, `R2 = 2^(2 * L)` are derived. Both `Plantard16/32` contain only the parameter abstractions rather than concrete values, so they are formalized once and ready for general use.

Listing 1.1. Plantard16 parameters and modulus axiom.

```
op L : int = 16.
op q, qinv, plan_alpha, plan_const : int.
op R  : int = 2^L.
op R2 : int = 2^(2 * L).
axiom q_bnd : 0 < q /\ q < 2^(L - plan_alpha - 1).
axiom qinv  : smod (q * qinv) R2 = 1.
axiom plan_const_def : (plan_const * R2) %% q = (-1) %% q.
```

Note that the `smod` operator in Listing 1.1 denotes *signed modular reduction*: rather than producing the ordinary range $[0, \mathtt{b})$, `smod` corrects it into the centered range $[-\mathtt{b}/2, \mathtt{b}/2)$. The operator and its range bound are defined below:

```
op smod (a b : int) =
  if (b %/ 2 <= a %% b) then a %% b - b else a %% b
  axiomatized by smodE.
lemma smod_bnd a b :
  0 < b => 2 %| b => -b %/ 2 <= smod a b < b %/ 2.
```

We define two separate Plantard reduction operators in the theory corresponding to the two usages in which Plantard reduction appears in the implementation. The single-argument form `PLAN_REDC a` assumes `a` is the product of two inputs and folds in the multiplication by *qinv* internally; this aligns with the naive Plantard reduction without precomputation. The form `PLAN_MULRED ab` assumes the *qinv* term is already embedded within input `ab`, that is $ab = a \cdot \zeta \cdot qinv$, where a is a polynomial coefficient and ζ a precomputed twiddle factor; as in the twiddle-factor case, `qinv` precomputation is merged into the twiddle table rather than performed externally at each butterfly operation (Section 2.3). The two operators are equivalent, captured by `PLAN_MULRED_eq`:

```
op PLAN_REDC   (a  : int) : int =
  let p = ((smod (a * qinv) R2) %/ R + 2^plan_alpha) * q
  in p %/ R.
op PLAN_MULRED (ab : int) : int =
  let p = ((smod ab R2) %/ R + 2^plan_alpha) * q in p %/ R.
lemma PLAN_MULRED_eq (a : int) :
  PLAN_REDC a = PLAN_MULRED (a * qinv).
```

The core correctness lemma `plan_corr` (in `Plantard16` and `Plantard32`) strictly follows the Plantard's definition, and establishes both output bound and field congruence under an input range as the precondition. For an input `a` satisfying $\mathtt{qR} - \mathtt{q} \cdot 2^{\mathtt{L+plan_alpha}} \leq \mathtt{a} < \mathtt{R2} - \mathtt{q} \cdot 2^{\mathtt{L+plan_alpha}}$ and the modulus axiom $\mathtt{q} < 2^{\mathtt{L-plan_alpha}-1}$, the result falls in the range $\left[-\frac{\mathtt{q}+1}{2}, \frac{\mathtt{q}}{2}\right]$ and produces $\mathtt{a} \cdot \mathtt{plan_const} \bmod \mathtt{q}$. Here we combine Huang et al.'s Plantard congruence proof (Theorem 1 in [20]) and Plantard output range proof (proof of Theorem 1.1 in [19]). We transfer these combined pen-and-paper proofs to EasyCrypt with minor modifications for semantic compatibility, primarily to handle EasyCrypt's centered residue convention via `smod` rather than the implicit signed modulo of pen-and-paper. The lemma is stated below:

```
lemma plan_corr (a : int) :
  q * R - q * 2^(L + plan_alpha)
  <= a < R2 - q * 2^(L + plan_alpha) =>
  q < 2^(L - plan_alpha - 1) =>
  -(q + 1) %/ 2 <= PLAN_REDC a <= q %/ 2 /\
  PLAN_REDC a %% q = a * plan_const %% q.
```

Note that we did not give a separate correctness lemma for `PLAN_MULRED`, since both operators are equivalent by `PLAN_MULRED_eq`. The bound is instead given by a standalone lemma `plan_mulred_bnd`, which requires no input precondition.

Finally we lift our Plantard formalization from the abstract operator `PLAN_MULRED` to the Jasmin function `plantard_mul_reduce` at the implementation level. The extracted procedure takes a 16-bit signed coefficient and a 32-bit precomputed twiddle, and returns a 16-bit signed result. We follow the standard EasyCrypt pHL pattern of Almeida et al. [3] and separate this lift process into three lemmas (Listing 1.2): a deterministic `hoare` judgement `plan_mul_corr_h` stating that `plantard_mul_reduce` computes $\mathtt{PLAN_MULRED}\,(\mathtt{a} \cdot \mathtt{b})$ on its inputs, an `islossless` lemma `plan_mul_ll` verifies the function termination, and the top-level `phoare` tactic `plan_mul_corr` combines the two statements above via `conseq` to assert the equivalence with probability 1.

Listing 1.2. Hoare-logic lift of `PLAN_MULRED` to the Jasmin procedure.

```
lemma plan_mul_corr_h _a _b :
  hoare [Jntt.M.plantard_mul_reduce :
    to_sint a = _a /\ to_sint b = _b ==>
    to_sint res = PLAN_MULRED (_a * _b)].

lemma plan_mul_ll : islossless Jntt.M.plantard_mul_reduce.
```

```
lemma plan_mul_corr _a _b :
  phoare [Jntt.M.plantard_mul_reduce :
    W16.to_sint a = _a /\ W32.to_sint b = _b ==>
    W16.to_sint res = PLAN_MULRED (_a * _b)] = 1%r
  proof. by conseq plan_mul_ll (plan_mul_corr_h _a _b). qed.
```

Note that the module `M` in `Jntt.M.plantard_mul_reduce` is the default name produced by `jasmin2ec`, which we keep unchanged after extraction. By lifting the theoretical correctness proof `plan_corr` to the implementation-level equivalence `plan_mul_corr`, every procedure call to `plantard_mul_reduce` in the generated NTT can now be replaced by its abstract specification. This allows the program equivalence proof in Section 4.2 to reason about Plantard multiplications at the algebraic level, lifting the bit-level `W16` and `W32` word arithmetic into integer operations directly.

4.2 Bridging Jasmin to Abstract NTT

We now establish the program equivalence between the Jasmin NTT extracted from our generator and the abstract NTT specification `NTT_Fq.NTT.ntt` inherited from Almeida et al. [3]. We relate the two NTT programs through an `equiv` judgment, where under the shared precondition both sides produce identical results. We first introduce the relevant programs and helper operators (Section 4.2), then we prove the forward (Section 4.2) and inverse (Section 4.2) equivalences sequentially.

Setup and helper operators The abstract specification of NTT is typically the schoolbook NTT without any reduction or encoding. It is defined in `NTT_Fq` theory as module `NTT`, by EasyCrypt's naming convention. We refer to it as `NTT_Fq.NTT.ntt`. The abstract specification is expressed plainly as a triple-nested while loop over an `Fq Array256.t` polynomial without bit-level type like `W16` or `W32`, and with no Plantard arithmetic or explicit reductions (provided in Supp. Material). We omit INTT's abstract specification for brevity as it shares identical structure with a scaling process at the end that multiplies every coefficient by `zetas_inv`.[127], the encoded form of $N^{-1} = 128^{-1} \bmod q$.

Unlike the abstract NTT described above, the Jasmin NTT extracted to EasyCrypt via `jasmin2ec` reflects our layer-unrolling decision driven by the constant-time implementation requirement (Section 3.3), flattening the outermost `while` loop into a sequence of procedure calls. The top-level procedure `Jntt.M.ntt` is shown below.

```
proc ntt(rp : W16.t Array256.t) : W16.t Array256.t = {
  rp <@ ntt_1(rp); rp <@ ntt_2(rp); rp <@ ntt_3(rp);
  rp <@ ntt_4(rp); rp <@ ntt_5(rp); rp <@ ntt_6(rp);
  rp <@ ntt_7(rp);
  return rp;}
```

Each `ntt_i` is a fully unrolled per-layer body containing a double-nested while loop, with its loop length len, zeta index k as compile-time constants, and explicit

`plantard_mul_reduce` procedure calls on the butterfly arms that require reduction. The inverse NTT decomposes the same way into `invntt_1` ... `invntt_7`, with an additional final scaling process.

Before we interpret our equivalence lemma, we introduce several auxiliary operators that appear in the precondition and are used throughout the subsequent proof, all defined below.

```
op lift_array256 (p : W16.t Array256.t) =
  Array256.map (fun x => inFq (W16.to_sint x)) p.

op signed_bound_cxq (coefs : W16.t Array256.t, l u c : int) :
    bool = forall k, l <= k < u => b16 coefs.[k] (c * q).

op signed_bound_hq  (coefs : W16.t Array256.t, l u c : int) :
    bool = forall k, l <= k < u => b16 coefs.[k] (c * hq).
```

The lifting `lift_array256` maps a coefficient array of 16-bit machine words into a vector of field elements by signed interpretation. The `signed_bound_cxq p l u c` operator asserts that every coefficient of `p` between indices l and u lies in the centered range $(-c \cdot q,\, c \cdot q)$. Its variant operator `signed_bound_hq` replaces q with the half-modulus $hq = (q+1)/2$, giving the tighter range $(-c \cdot hq,\, c \cdot hq)$.

Forward NTT equivalence With these helper operators, we can now establish our forward equivalence lemma `ntt_correct_aux` (Listing 1.3). The precondition requires that the input coefficient array `rp` on the Jasmin side, lifted by `lift_array256`, exactly matches the abstract input `r`, and that every Jasmin coefficient lies in $(-q,\, q)$, as expressed by `signed_bound_cxq rp 0 256 1`. The postcondition states that the two outputs are equal under the same lifting.

Listing 1.3. Top-level equivalence between abstract and Jasmin NTT.

```
equiv ntt_correct_aux :
  NTT.ntt ~ Jntt.M.ntt :
    r{1} = lift_array256 rp{2} /\
    zetas{1} = zetas /\
    signed_bound_cxq rp{2} 0 256 1
    ==>
    res{1} = lift_array256 res{2}.
```

Note that the precondition statement `zetas{1} = zetas` refers to two separate zetas: `zetas{1}` is the abstract program's argument (a field-element table in $\mathbb{Z}_q$), while `zetas` on the right is the raw twiddle table from `NTT_Fq`. The Jasmin code uses its Plantard-encoded counterpart `Jntt.zetas` internally, related to the abstract table by the lemma `zetas_corr` (Listing 1.4). Both `zetas{1}` and the right-hand `zetas` are equivalent at the field-level, making the equality hold:

Listing 1.4. Twiddle-table encoding correctness.

```
op array128_plan (p : Fq Array128.t) : W32.t Array128.t =
  Array128.map (fun x =>
    W32.of_int (smod (asint x * (-R2) %% q * qinv) R2)) p.
```

```
lemma zetas_corr : array128_plan zetas = Jntt.zetas.
```

The equivalence proof is handled layer by layer. On the abstract side, the outer `while` loop is unrolled one iteration at a time through `unroll{1} 1; rcondt{1} 1`, and the derived subgoal can be trivially closed by tactic `auto`. On the Jasmin side, `ntt_i` procedure call is inlined by the tactic `inline Jntt.M.ntt_i`, which brings layer i full procedure into proof context. We then apply the tactic `seq 3 8` to isolate these two segments to make them aligned, namely one iteration unrolling corresponds to one procedure call `ntt_i`, separately from the other `ntt_j`. The first layer is where the bound tightens: we go in with `signed_bound_cxq rp{2} 0 256 1` and come out with `signed_bound_hq rp{2} 0 256 3`, since each `plantard_mul_reduce` call pulls coefficients into the half-modulus range. All subsequent layers carry the post-condition `signed_bound_hq rp{2} 0 256 c` accordingly, where c is derived directly from the analyzer of Section 3.2.

We use the `while` tactic to reason about the inner loop one iteration at a time by stating a series of explicit loop invariants that track which coefficients have been processed and which remain, along with their respective bounds. The EasyCrypt proof engine then generates a preservation obligation, which we discharge by stepping through the butterfly body and proving the invariant is preserved across each iteration. Specifically at the innermost while loop where butterfly operations live, each `plantard_mul_reduce` call at the bit-level is replaced by its abstract specification `PLAN_MULRED` via `ecall{2} (Fq.plan_mul_corr ...)`. The encoded twiddle `Jntt.zetas.[k]` is then rewritten by `rewrite -zetas_corr` back into the field-level abstract twiddle. At this point, the innermost loop proof closes trivially by applying `smt` or `ring`.

Inverse NTT equivalence and final scaling The INTT equivalence proof follows a similar pattern to the forward case of Section 4.2, with a few differences. The abstract specification defines a standalone twiddle table `zetas_inv` for the inverse, whose entries map to the negated forward twiddles in reverse order, i.e., `zetas_inv.[126-k] = -zetas.[k+1]` for `k` $\in [0, 126]$, with the last entry `zetas_inv.[127]` set to the scaling constant $N^{-1} = 128^{-1} \bmod q$. For the Jasmin generated implementation, we reuse the same encoded twiddle table in reverse, negating each entry at usage (`zeta_0 <- -zeta_0`). The proof bridges two twiddle tables through a per-iteration invariant in the innermost `while` loop:

```
zeta_{1}  = NTT_Fq.zetas_inv.[126 - to_uint k{2}] /\
zeta_0{2} = - Jntt.zetas.[to_uint k{2} + 1]
```

which is applied at every `plantard_mul_reduce` call together with the negation identity above. Once the negation is factored out, the remaining proof follows a similar pattern as the forward case, with the exception that GS butterflies replace CT butterflies.

At the end of the seven layers, the final scaling multiplies every coefficient by $N^{-1} = 128^{-1} \bmod q$, which cancels the accumulated factor of 2^{-7} produced by the GS butterflies. We show the correctness of the scaling using the same

pHL pattern as `plan_mul_corr` described in Section 4.1: a deterministic Hoare judgment `ntt_scale_h`, an `islossless` lemma `ntt_scale_ll`, and the top-level `phoare` lemma `ntt_scale_corr` combining the two by `conseq` as listed below.

```
lemma ntt_scale_h (_a : int Array256.t) :
  hoare [Jntt.M.ntt_scale :
    (forall k, 0 <= k < 256 => W16.to_sint rp.[k] = _a.[k])
    ==>
    (forall k, 0 <= k < 256 =>
      W16.to_sint res.[k] =
      PLAN_MULRED (_a.[k] * (-33544352)))].

lemma ntt_scale_ll : islossless Jntt.M.ntt_scale.

lemma ntt_scale_corr (_a : int Array256.t) :
  phoare [Jntt.M.ntt_scale :
    (forall k, 0 <= k < 256 => W16.to_sint rp.[k] = _a.[k])
    ==>
    (forall k, 0 <= k < 256 =>
      W16.to_sint res.[k] =
      PLAN_MULRED (_a.[k] * (-33544352)))] = 1%r.
```

Note that the constant -33544352 is N^{-1} encoded in Plantard form. After applying `plan_mul_corr`, the result reduces to multiplication by N^{-1}. Combining `ntt_correct_aux`, `invntt_correct_aux`, and `ntt_scale_corr`, we establish the full equivalence between the generated Jasmin NTT and the abstract specifications `NTT_Fq.NTT.ntt` and `NTT_Fq.NTT.invntt`. With the existing `NTT_algebra` chain (`NTT_opt` $\Leftrightarrow$ `NTT_bsrev` $\Leftrightarrow$ `NTT_naive` $\Leftrightarrow$ `ntt_spec`) developed by Almeida et al. [3], the generated implementation is therefore equivalent to the mathematical NTT specification over $\mathbb{Z}_q$, automatically establishing the round-trip identity $\texttt{invntt}(\texttt{ntt}(r)) = r$.

5 Benchmarks

In this section, we first describe the benchmark setup and report our measured cycles (Section 5.1). Then we discuss the observed speedups and the reason that causes the variation across schemes (Section 5.2).

5.1 Setup and Results

We now report our benchmark results of generated Jasmin and C implementations compared with existing implementations in C and Jasmin (only formosa-mlkem [3]). We only target reference implementations rather than AVX2, since we aim for a general-purpose NTT produced by code generation rather than an extensively optimized pipeline built by manual tuning.

This scoping also reflects where Plantard arithmetic is strongest. Plantard's advantage over Montgomery only survives on scalar implementations, since it

saves a multiplication per butterfly, and that saving does not survive vectorization. We also hand-tuned AVX2 implementations of the Plantard NTT to confirm this idea. Specifically, since Plantard multiplication by a constant requires the low 2ℓ bits of an $\ell \times 2\ell$ product, for which AVX2 provides no single instruction. Therefore, for $\ell = 16$, we synthesize it by splitting the precomputed twiddle into halves and folding the final scaling into `vpmulhrsw`, giving three multiplications and one addition per butterfly against Montgomery's three multiplications and one subtraction. In this case, we measure Plantard AVX2 at roughly 10% slower than Montgomery AVX2. For $\ell = 32$, as used by ML-DSA, the gap is wider. AVX2 has no multiply accepting a 64-bit operand, so the 32×64 product must be synthesized from several `vpmuludq` instructions with shifts and additions, whereas Montgomery obtains its full product from a single `vpmuldq`. We measure Plantard at roughly 28% slower than Montgomery in this setting. We therefore do not pursue a vectorized backend here.

Our benchmark targets three schemes: ML-KEM, ML-DSA, and FN-DSA-1024. For ML-KEM and ML-DSA, we reused their `test_speed` routines to ensure the test harnesses matched. FN-DSA does not provide an equivalent harness in its codebase, so we extracted the ML-KEM and ML-DSA testing harness to ensure consistency. We also confirmed that our extracted harness is reliable by comparing the results with the original ML-KEM harness. The results demonstrate a negligible difference of approximately 20 cycles.

We conducted our benchmark on an Intel Xeon E-2314 (4 cores, 2.80 GHz base) running Linux. We followed standard benchmark practice that turned off CPU TurboBoost and Hyperthreading to maintain one thread per CPU core. For all C benchmarks, we use `gcc-11.4` with the flag `-O3 -fomit-frame-pointer`, matching the original ML-KEM and ML-DSA `test_speed` setup. We compiled our generated Jasmin program using `Jasmin 2025.6.1` and use `clang-14` to link the output assembly. The flags for `clang-14` are `-O3 -march=native -fomit -frame-pointer -fwrapv`, matching the formosa-mlkem testing setup. Each benchmark runs 10k iterations, recording the CPU cycles using `rdtsc` bracketed by `cpuid` for serialization. We repeat the benchmark three times end-to-end and report the median of the three per-run medians. The median was consistent across runs, differing from the average by under 1% of the cycle count.

Table 1 gives the C and Jasmin benchmark results across the three schemes. The forward NTT improves by 1.83× (ML-KEM), 1.73× (ML-DSA), and 1.47× (FN-DSA-1024). The inverse transform improves by 2.48×, 1.78×, and 1.66× respectively. Combining forward and inverse transform speedups, round-trips land at 2.22×, 1.75×, and 1.56×. This metric can intuitively reflect the end-to-end performance, since polynomial multiplication in lattice-based schemes always invokes one forward NTT and one inverse NTT per operation.

We compare our generated Plantard ML-KEM and ML-DSA NTT/INTT with widely deployed NTT/INTT from `mlkem-native` and `mldsa-native` [21]. We build their software with `OPT=0 AUTO=0` to target their scalar C implementations, and compile both with `gcc-11.4` and `-O3 -fomit-frame-pointer`. Table 2 gives the comparison. For ML-DSA, our generated NTT is faster in both

Table 1. Cycle counts for C and Jasmin implementations of NTT and INTT on Intel Xeon E-2314. Round-trip figures are measured independently rather than derived from the NTT and INTT columns.

Scheme	Backend	Implementation	NTT	INTT	Round-trip	Speedup
ML-KEM	C	Reference [10]	7,078	10,222	17,420	—
		Ours	3,876	4,124	7,830	2.22×
	Jasmin	formosa-mlkem [3]	17,332	19,679	36,815	—
		Ours	13,733	8,978	22,565	1.63×
ML-DSA	C	Reference [6]	6,561	8,813	15,384	—
		Ours	3,796	4,950	8,802	1.75×
FN-DSA-1024	C	Reference [16]	31,006	32,892	64,133	—
		Ours	21,073	19,813	41,118	1.56×

Table 2. ML-KEM and ML-DSA generated NTT/INTT against C implementations of `mlkem-native` and `mldsa-native` on Intel Xeon E-2314. Ratios above 1 favor our implementation.

Scheme	Implementation	NTT	INTT
ML-KEM	`mlkem-native` [21]	3,325	4,848
	Ours	3,876	4,124
	ratio	0.86×	1.18×
ML-DSA	`mldsa-native` [21]	6,318	7,539
	Ours	3,796	4,950
	ratio	1.66×	1.52×

NTT/INTT, by 1.66× on the forward and 1.52× on the inverse. For ML-KEM, INTT is faster by 1.18×, while the forward is 0.86×, the only case in which our implementation is slower. The cause is compiler auto-vectorization. Montgomery reduction operates entirely on 16-bit values, whereas Plantard requires a 32-bit intermediate, and the resulting difference in vector lane count outweighs the fewer multiplications that Plantard performs. We defer the detailed analysis to Section 5.2. Note also that our forward NTT normalizes its output to $(-q/2, q/2)$ while the `native` implementations do not, since we aim for generating general-purpose NTT. We defer the analysis on this exception in Section 5.2.

We also integrated our Plantard NTT and reduction into ML-KEM and ML-DSA, and verified the integration correctness using the NIST Known Answer Test (KAT) vectors. The outputs of our Plantard-integrated builds are identical to the reference implementations bit-for-bit across all test vectors. We also benchmark the entire pipeline of ML-KEM and ML-DSA using the same benchmark setup as above, measuring key generation, encapsulation/signing, and decapsulation/verification across all three security levels of each scheme. The end-to-end speedups range from 1.08× to 1.32×, with decapsulation and signing benefiting most. Table 3 demonstrates the benchmark results.

Table 3. End-to-end cycle counts for ML-KEM and ML-DSA on Intel Xeon E-2314

Scheme	Variant	Implementation	KeyGen	Encaps/Sign	Decaps/Verify
ML-KEM	ML-KEM-512	Reference [10]	107,179	123,329	162,047
		Ours	91,471	97,377	122,499
		speedup	1.17×	1.27×	1.32×
	ML-KEM-768	Reference	179,498	195,799	247,272
		Ours	158,522	163,656	198,825
		speedup	1.13×	1.20×	1.24×
	ML-KEM-1024	Reference	270,352	282,571	347,726
		Ours	241,950	238,976	285,829
		speedup	1.12×	1.18×	1.22×
ML-DSA	ML-DSA-2	Reference [6]	282,324	997,553	309,509
		Ours	255,364	788,805	267,618
		speedup	1.11×	1.26×	1.16×
	ML-DSA-3	Reference	515,602	1,669,946	491,183
		Ours	478,567	1,334,233	433,971
		speedup	1.08×	1.25×	1.13×
	ML-DSA-5	Reference	755,732	2,047,556	792,170
		Ours	701,701	1,651,305	709,543
		speedup	1.08×	1.24×	1.12×

5.2 Performance Analysis

The forward NTT shows speedups of 1.83× (ML-KEM), 1.73× (ML-DSA), and 1.47× (FN-DSA-1024), see Table 1. FN-DSA's speedup is visibly less significant, and we can attribute this to its parameters $(N, q) = (1024, 12289)$. Since the modulus q is comparatively closer to the 16-bit signed word bound, the analyzer is forced to insert reductions very frequently, typically FN-DSA needs two reductions in forward NTT and a reduction every two layers for inverse NTT. Whereas ML-KEM and ML-DSA have much looser storage space, which barely requires reductions: ML-KEM only requires a reduction at layer 3 in the inverse transform, and ML-DSA does not require any reduction at all.

For the inverse NTT, we have the speedups: 2.48× for ML-KEM, 1.78× for ML-DSA, and 1.66× for FN-DSA-1024. Note that the inverse transform obtains larger speedups than the forward transform for ML-KEM and FN-DSA-1024, and a comparable speedup for ML-DSA. The reference C implementations apply more modular reductions in the inverse transform since the GS butterfly's plus arm $A(x) + B(x)$ doubles the coefficient at each layer, which pushes the coefficient close to the storage bound. With Plantard's tighter output bound and analyzer's per-layer tracking, the reductions can be mostly and safely omitted. For the forward transform, the CT butterfly grows the bound only additively, and even Montgomery-based reference implementations can omit most reductions in the forward NTT. Though Plantard's tighter bound can help, there are

fewer reductions left to eliminate. ML-DSA is the exception that confirms this reasoning: its forward transform already requires no reduction at all, so the two directions start from comparable positions and the gap closes.

It is noticeable that our generated forward ML-KEM transform is the only case that is slower than `mlkem-native`, and this is caused by $\ell = 16$. The 16-bit Plantard multiplication reduction specifically requires a 32-bit intermediate for the product from multiplication form of $\ell \times 2\ell$, while Montgomery reduction used by `mlkem-native` operates entirely on 16-bit values, namely $\ell \times \ell$. At `-O3`, `gcc` automatically vectorizes the butterfly loop, and the 16-bit form admits twice as many lanes per vector as the 32-bit one. We also confirm this point by compiling both implementations with `-O3 -fno-tree-vectorize`, which disables the auto-vectorization while retaining every other `-O3` optimization. The outcome then becomes reversed, and our forward transform is 1.77× faster than `mlkem-native`. The inverse transform is unaffected: we remain faster either way, at 1.18× with vectorization and 1.68× without. The deficit is therefore a consequence of how the emitted C meets the vectorizer at this word size, rather than a property of Plantard arithmetic.

We can also observe that the generated Jasmin implementation is considerably slower than the C implementation. This is because the Jasmin compiler does not perform aggressive optimizations such as memory access optimization that the `-O3` flag applies in `gcc` and `clang`. The formosa-mlkem authors report a similar gap, so we are matching their methodology that establishes EasyCrypt formalization on our generated code rather than making a claim about absolute Jasmin performance.

Although we observe substantial NTT-level speedups in Table 1, the corresponding end-to-end improvements in Table 3 are still necessarily limited since NTT is one of several components (e.g. ML-KEM-512, NTT costs 3 876 cycles against over 100,000 cycles for end-to-end key generation). The decapsulation in ML-KEM (1.22×–1.32×) and signing in ML-DSA (1.24×–1.26×) benefit the most from the NTT improvements, since both operations invoke the NTT pipeline more heavily than key generation or verification.

6 Conclusion

In this paper, we presented our code generator with an embedded analyzer component that produces NTT routines in Jasmin and C backends, using Plantard arithmetic with precomputed constants encoded at generation time. The generator is sufficiently generic to handle both complete and incomplete NTT cases across multiple lattice-based schemes such as ML-KEM, ML-DSA, and FN-DSA-1024, given a parameter triple. The generated Jasmin ML-KEM is also formally verified end-to-end using EasyCrypt and the Jasmin extraction tool `jasmin2ec`, against the abstract NTT specification from formosa-mlkem.
We also demonstrated our benchmark results targeting ML-DSA and FN-DSA-1024 on the C backend, and ML-KEM on both Jasmin and C backends. We observe a reasonable speedup against the reference C implementations: 1.5× to

1.8× on the forward NTT and 1.7× to 2.5× on the inverse, across all three target schemes. The inverse transform sees more improvement for ML-KEM and FN-DSA-1024, thanks to Plantard's tighter output bounds, which allow the analyzer to omit most reductions. These NTT-level improvements yield end-to-end speedups of up to 1.32× for ML-KEM and 1.26× for ML-DSA.

Limitations. Currently our formal verification focuses only on the Jasmin ML-KEM output, which relies on the abstract specification of incomplete NTT. For the complete NTT used in ML-DSA and FN-DSA, the proof chain developed by Almeida et al. [3] would still need to be manually adapted, since it depends on several ML-KEM-specific libraries (`Kyber.ec`) whose lemmas and operators are bound to ML-KEM's modulus $q = 3329$. Extending verification to other schemes would require re-developing the field theory and the helper lemmas that rely on the properties of the new modulus. The full equivalence proof would also need to be reconstructed. Our portable C backend also remains unverified. Although the generated source does not include any branch on coefficient values, both its functional correctness and its constant-time behavior depend on the C compiler preserving those properties, which no widely used C compiler guarantees. The generator itself is likewise unverified; it is a Python program whose output we verify, rather than a certified compiler, so a defect in the generator would surface as a failed equivalence proof on the emitted Jasmin rather than as a silent miscompilation. Furthermore, our code generator produces scalar code only. As discussed in Section 5.1, this is a scoping decision rather than an unfinished feature, since Plantard arithmetic is not the right choice for vectorized targets.

Future work. The immediate next step is to take the abstract NTT layer as the verification target and extend the generator to emit the equivalence proof alongside the code. Generalizing the full proof chain of Almeida et al. [3] is not straightforward, since specific lemmas and operators rely on properties of the modulus. Our layer-decomposition proof, however, exhibits repeating patterns across layers, and the per-layer bounds it requires are already computed by the analyzer. This suggests the equivalence to the abstract specification can be largely automated, which would address the call for greater proof automation raised by Almeida et al. [3], although connecting that specification down to the DFT definition would still require a per-modulus algebraic development.

Disclosure of AI use. We acknowledge that the figures and algorithm pseudocodes in this paper were produced with the assistance of generative AI tools (Claude). The authors verified their correctness and conducted manual revisions. All technical content, claims, proofs, and experimental results are the authors' own work.

References

1. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key Exchange — A new hope. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 327–343 (2016)

2. Almeida, J.B., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B., Strub, P.Y.: Jasmin: High-Assurance and High-Speed Cryptography. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1807–1823. ACM, Dallas Texas USA (Oct 2017). https://doi.org/10.1145/3133956.3134078
3. Almeida, J.B., Barbosa, M., Barthe, G., Grégoire, B., Laporte, V., Léchenet, J.C., Oliveira, T., Pacheco, H., Quaresma, M., Schwabe, P., Séré, A., Strub, P.Y.: Formally verifying Kyber Episode IV: Implementation Correctness (2023)
4. Aoki, D., Minematsu, K., Okamura, T., Takagi, T.: Efficient Word Size Modular Multiplication over Signed Integers. In: 2022 IEEE 29th Symposium on Computer Arithmetic (ARITH). pp. 94–101 (Sep 2022). https://doi.org/10.1109/ARITH54963.2022.00026
5. Apple Inc.: corecrypto. https://github.com/apple/corecrypto (2026), accessed: 2026-08-03
6. Bai, S., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Dilithium
7. Barrett, P.: Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In: Conference on the Theory and Application of Cryptographic Techniques. pp. 311–323. Springer (1986)
8. Barthe, G., Grégoire, B., Heraud, S., Béguelin, S.Z.: Computer-Aided Security Proofs for the Working Cryptographer. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Rogaway, P. (eds.) Advances in Cryptology – CRYPTO 2011, vol. 6841, pp. 71–90. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22792-9_5
9. Becker, H., Hwang, V., Kannwischer, M.J., Yang, B.Y., Yang, S.Y.: Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 221–244 (2022). https://doi.org/10.46586/tches.v2022.i1.221-244
10. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS – Kyber: A CCA-secure module-lattice-based KEM (2017)
11. Carette, J., Kiselyov, O., Shan, C.C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. Journal of Functional Programming **19**(5), 509–543 (Sep 2009). https://doi.org/10.1017/S0956796809007205
12. Chiu, C.M., Liu, J., Tsai, M.H., Shi, X., Wang, B.Y., Yang, B.Y.: Algebraic Linear Analysis for Number Theoretic Transform in Lattice-Based Cryptography. IACR Transactions on Cryptographic Hardware and Embedded Systems **2025**(3), 668–692 (Jun 2025). https://doi.org/10.46586/tches.v2025.i3.668-692
13. Cooley, J.W., Tukey, J.W.: An Algorithm for the Machine Calculation of Complex Fourier Series. Mathematics of Computation **19**(90), 297–301 (1965). https://doi.org/10.2307/2003354
14. Coquand, T., Huet, G.: The calculus of constructions. Ph.D. thesis, INRIA (1986)
15. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 1202–1219 (May 2019). https://doi.org/10.1109/SP.2019.00005
16. Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU

17. Gentleman, W.M., Sande, G.: Fast Fourier Transforms: For fun and profit. In: Proceedings of the November 7-10, 1966, Fall Joint Computer Conference on XX - AFIPS '66 (Fall). p. 563. ACM Press, San Francisco, California (1966). https://doi.org/10.1145/1464291.1464352
18. Göttert, N., Feller, T., Schneider, M., Buchmann, J., Huss, S.: On the Design of Hardware Building Blocks for Modern Lattice-Based Encryption Schemes. In: Prouff, E., Schaumont, P. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2012. pp. 512–529. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33027-8_30
19. Huang, J., Zhang, J., Zhao, H., Liu, Z., Cheung, R.C.C., Koç, Ç.K., Chen, D.: Improved Plantard Arithmetic for Lattice-based Cryptography. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 614–636 (Aug 2022). https://doi.org/10.46586/tches.v2022.i4.614-636
20. Huang, J., Zhao, H., Zhang, J., Dai, W., Zhou, L., Cheung, R.C.C., Koç, Ç.K., Chen, D.: Yet Another Improvement of Plantard Arithmetic for Faster Kyber on Low-End 32-bit IoT Devices. IEEE Transactions on Information Forensics and Security **19**, 3800–3813 (2024). https://doi.org/10.1109/TIFS.2024.3371369
21. Kannwischer, M.J.: Keynote: mlkem-native & mldsa-native: Open-source high-speed high-assurance pqc. In: Proceedings of the 2025 Cloud Computing Security Workshop (CCSW '25) (2025). https://doi.org/10.1145/3733812.3765539
22. Masuda, M., Kameyama, Y.: Unified Program Generation and Verification: A Case Study on Number-Theoretic Transform. In: Hanus, M., Igarashi, A. (eds.) Functional and Logic Programming, vol. 13215, pp. 133–151. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-99461-7_8
23. Matz, M., Hubička, J., Jaeger, A., Mitchell, M.: System V application binary interface: AMD64 architecture processor supplement, version 1.0. https://gitlab.com/x86-psABIs/x86-64-ABI (2025), table 3.1, Micro-Architecture Levels. Accessed 2026-08-14
24. Melchor, C.A., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Gaborit, P., Persichetti, E., Zémor, G., Bourges, I.: Hamming quasi-cyclic (hqc). NIST PQC Round **2**(4), 13 (2018)
25. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of computation **44**(170), 519–521 (1985)
26. National Institute of Standards and Technology (US): Module-lattice-based digital signature standard. Tech. Rep. NIST FIPS 204, National Institute of Standards and Technology (U.S.), Washington, D.C. (Aug 2024). https://doi.org/10.6028/NIST.FIPS.204
27. National Institute of Standards and Technology (US): Module-lattice-based key-encapsulation mechanism standard. Tech. Rep. NIST FIPS 203, National Institute of Standards and Technology (U.S.), Washington, D.C. (Aug 2024). https://doi.org/10.6028/NIST.FIPS.203
28. National Institute of Standards and Technology (US): Stateless hash-based digital signature standard. Tech. Rep. NIST FIPS 205, National Institute of Standards and Technology (U.S.), Washington, D.C. (Aug 2024). https://doi.org/10.6028/NIST.FIPS.205
29. Plantard, T.: Efficient word size modular arithmetic. IEEE Transactions on Emerging Topics in Computing **9**(3), 1506–1518 (2021)
30. Pollard, J.M.: The fast fourier transform in a finite field. Mathematics of computation **25**(114), 365–374 (1971)
31. Seiler, G.: Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography (2018)

32. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Review **41**(2), 303–332 (1999). https://doi.org/10.1137/S0036144598347011
33. Trieu, A.: Formally Verified Number-Theoretic Transform. IACR Communications in Cryptology **2**(4), cc2–3–15 (Jan 2026). https://doi.org/10.62056/ahbn-4tw9
34. Yang, Y., Jia, Y., Xu, G.: On modular algorithms and butterfly operations in number theoretic transform. Cryptology ePrint Archive (2024)