

Automorphism-Compatible NTT and its Application to Homomorphic Encryption

Charanjit S. Jutla, Nathan Manohar and Guy Moshkowich

IBM Research

Abstract. For N a power of 2, the negacyclic number-theoretic transform (NTT) maps a degree $N - 1$ polynomial to its evaluations at the N primitive $2N$ th roots of unity. In-place butterfly algorithms such as Cooley–Tukey and Gentleman–Sande compute the negacyclic NTT in $O(N \log N)$ time and output the evaluations in a specific, standard order. We give in-place butterfly algorithms for the negacyclic NTT and its inverse that instead output the evaluations in an *automorphism-compatible* order. These algorithms have the same structure as Cooley–Tukey and Gentleman–Sande, differing only in their precomputed twiddle factors. Our automorphism-compatible butterfly algorithms are applicable to the packed fully homomorphic encryption (FHE) schemes CKKS and BGV/BFV, where the negacyclic NTT is a major component of ciphertext arithmetic. Using our automorphism-compatible butterfly algorithms, Galois automorphisms can be applied to ciphertexts stored in double-CRT format via a simple cyclic shift or adjacent-pair swap of the data. This is contrast to butterfly algorithms currently used that output in the standard order, where applying an automorphism requires performing an arbitrary-looking permutation of the stored data. Our automorphism-compatible butterfly algorithms remove the need for a dedicated automorphism functional unit used in FHE hardware accelerators.

Keywords: Number-theoretic transform · Discrete Fourier transform · Butterfly algorithms · Automorphisms · Fully homomorphic encryption

1 Introduction

The number-theoretic transform (NTT) and its analogue over the complex numbers, the discrete Fourier transform (DFT), are fundamental primitives underlying fast polynomial arithmetic. For N a power of two, the related negacyclic NTT maps a polynomial $m(X) = m_0 + m_1X + \dots + m_{N-1}X^{N-1}$ to its evaluations at the N primitive $2N$ th roots of unity, i.e. the odd powers of a fixed primitive $2N$ th root of unity ω . Computing the NTT (or the DFT if the base field is $\mathbb{C}$ rather than $\mathbb{Z}_q$) or negacyclic NTT can be done in $O(N \log N)$ time using the Cooley-Tukey algorithm [CT65]. This is significantly faster than the $O(N^2)$ time required for standard matrix-vector multiplication, which could be used to compute the NTT as a linear transform. Moreover, the Cooley-Tukey algorithm is an *in-place butterfly algorithm*, meaning it can be evaluated with $O(1)$ auxiliary storage, which is crucial for performance.

A defining feature of these in-place butterfly algorithms is that they impose a specific ordering on their input or output. The Cooley-Tukey algorithm recurses via decimation in time, meaning that at each level of recursion, the input is split into even and odd indices. Thus, the input to the Cooley-Tukey algorithm needs to be stored in a bit-reversed manner, and the output is in the standard order. Alternatively, the negacyclic NTT can be computed using the Gentleman-Sande algorithm [GS66], which is also an in-place butterfly algorithm, but recurses via decimation in frequency, meaning that at each level of recursion,

the input is split into the first half and the second half. Thus, the input to the Gentleman-Sande algorithm is in the standard order, and the output is in bit-reversed order. When we refer to the standard order of the input $m(X)$, we mean the order $m_0, m_1, \dots, m_{N-1}$, and for the output, the standard order refers to $m(\omega), m(\omega^3), \dots, m(\omega^{2^N-1})$ for a fixed primitive $2N$ th root of unity ω (in a finite field). This bit-reversal is not an artifact of these two particular algorithms but is intrinsic: a radix-2 in-place butterfly transform, consisting solely of butterfly operations with $O(1)$ auxiliary storage, admits a matrix factorization in which the bit-reversal permutation appears as a factor that cannot be absorbed into the butterfly stages [Loa92]; thus, such an algorithm cannot have *both* its input and its output in the standard order.

This raises a more basic question about which *orderings* of the roots of unity a butterfly algorithm can accommodate at all. The standard order above evaluates $m(X)$ at $\omega, \omega^3, \omega^5, \dots, \omega^{2^N-1}$, but this is only one of the $N!$ orderings of the primitive $2N$ th roots of unity. Any ordering yields a valid linear transform that can be computed as an $O(N^2)$ matrix-vector product, but only special orderings admit an $O(N \log N)$ in-place butterfly algorithm¹. The first question we address is exactly which orderings these are.

Classifying admissible orderings. We give a complete characterization of the orderings of the primitive $2N$ th roots of unity that admit an in-place butterfly algorithm either of Cooley-Tukey (decimation-in-time) or Gentleman-Sande (decimation-in-frequency) form. Writing an ordering as a bijection $f: [N] \rightarrow 2 * [N] + 1$ so that the j th evaluation point is $\omega^{f(j)}$, we show (Theorem 1 in Section 3.3) that exactly 2^{N-1} of the $N!$ orderings admit such an algorithm, and we give a simple recursive condition that characterizes them. Notably, apart from the difference in whether the input or the output is bit-reversed, the orderings that admit a decimation-in-time butterfly algorithm are exactly the same as those that admit a decimation-in-frequency butterfly algorithm. While it is no surprise that the standard order $f(j) = 2j + 1$ satisfies this condition, the surprising result is that an *automorphism-compatible* order that we describe below also satisfies this condition. For readers familiar with fully homomorphic encryption (FHE), the main upshot is that ciphertexts can be maintained in a format which enables SIMD-rotations of the underlying plaintext by rotations of the ciphertext data, instead of by non-trivial permutations of the data that current implementations require. See Figure 1 for a quick look-ahead at the two types of permutations: (a) standard-order is what is currently implemented, and (b) automorphism-compatible order is what our implementation provides. We now describe FHE and the underlying ciphertext data representations.

Application to Fully Homomorphic Encryption. The choice of ordering turns out to have significant consequences in fully homomorphic encryption (FHE), where the negacyclic NTT is a substantial component of ciphertext arithmetic and a main bottleneck in performance. FHE enables a client to encrypt their data, send it to a cloud server, and have the server compute any function on the client’s data in encrypted form. Since Gentry’s breakthrough in 2009 [Gen09a, Gen09b], FHE has gone through many iterations towards practicality and today, some of the most widely used schemes are BGV/BFV [BGV12, Bra12, FV12] and CKKS [CKKS17] that support SIMD computation. These schemes pack (up to) thousands of plaintext values into each ciphertext and support computations between ciphertext slots using the underlying field’s Galois automorphisms.

Gentry’s breakthrough for FHE came with the realization that lattice based encryption schemes, which are naturally additively homomorphic, can also be made homomorphic under multiplication if the underlying lattices are restricted to ideal lattices in the ring

¹An astute reader may wonder that since any permutation can be performed in $O(N)$ time, why couldn’t one simply apply a post-permutation after the butterfly algorithm without much effect on the time complexity. However, butterfly algorithms offer natural parallelism that can reduce the critical path to essentially $O(\log N)$, whereas arbitrary permutations may not have an easy parallelization and/or a hardware implementation. We discuss these practical issues in more detail in Section 5.

of integers of a number field. Subsequent schemes follow this idea of using ring elements where the ring is an order of a number field. These ring elements are naturally represented as polynomials modulo the defining polynomial of the number field; for instance, for power of two-cyclotomic number fields, this defining polynomial is $X^N + 1$ (where N is a power of two). All such schemes obtain their security by adding noise to the lattice elements *and* further restricting the polynomials to be defined modulo an integer (see, for instance [Reg09]).

Focusing on BGV/BFV and CKKS, these schemes have ciphertexts of the form $\text{ct} = (c_0, c_1) \in \mathcal{R}_Q^2$, where $\mathcal{R} := \mathbb{Z}[X]/(X^N + 1)$ is a cyclotomic ring of integers with ring dimension $N = 2^n$ a power of two, $Q \in \mathbb{Z}^+$ is a ciphertext modulus, and $\mathcal{R}_Q := \mathcal{R}/Q\mathcal{R}$. Each ciphertext can encrypt either a single element, a modular integer for BGV or a complex number for CKKS, or following [SV14], a polynomial $m(X)$ itself. The latter idea of encrypting a polynomial, as we will see, allows SIMD homomorphic operations. The plaintext, after an appropriate encoding, is usually added to one or both of the ciphertext components. This, clearly makes the scheme linearly homomorphic, and homomorphic multiplications are also ensured by (tensor) multiplication of ciphertexts. Since the ciphertext components are members of the polynomial ring, more precisely $\mathcal{R}_Q$, multiplication of two such elements coming from two different ciphertexts, leads to multiplication of the plaintext (encoded) polynomials. At a conceptual level, if the plaintext encoding involved mapping a vector of plaintexts into a polynomial by a discrete Fourier transform, the multiplication of such polynomial encodings will then equivalently be a SIMD multiplication of the plaintext vectors (by the famous Fourier convolution theorem). We refer the reader to Figure 3 in Section 4.7 for a commutative diagram making this observation more formal.

Since polynomial multiplication is a key component of such FHE schemes, for improved efficiency, Q is chosen to be a product of ℓ primes q_i so that one can use the residue number system (RNS) [Gar59] to represent an element of $\mathcal{R}_Q$ by an element of $\mathcal{R}_{q_0} \times \dots \times \mathcal{R}_{q_{\ell-1}}$ (employing the Chinese remainder theorem). The element of $\mathcal{R}_{q_i}$ is referred to as the i th limb. Within each limb, by choosing primes $q_i \equiv 1 \pmod{2N}$, one can guarantee that all the primitive $2N$ th roots of unity of $\mathbb{Z}_{q_i}$ lie in the field itself and are the odd powers of a chosen primitive $2N$ th root of unity ω , so that the negacyclic NTT maps an element $b(X) \in \mathcal{R}_{q_i}$ to its evaluations $(b(\omega), b(\omega^3), \dots, b(\omega^{2N-1})) \in \mathbb{Z}_{q_i}^N$. This representation of an element of $\mathcal{R}_Q$ is referred to as the double-CRT representation, and, by using it, addition and multiplication over $\mathcal{R}_Q$ can be performed more efficiently by simple addition/multiplication of elements of the various $\mathbb{Z}_{q_i}$'s, where the bit-length of $q_i \ll$ the bit-length of Q . Thus, the ability to efficiently compute NTTs is critical for FHE performance.

Crucially, most homomorphic operations are performed in a SIMD manner on the ciphertext data in double-CRT format and, thus, are agnostic to the order in which the evaluation points are stored. Even the operations that require leaving the evaluation domain (such as the modular reduction on the coefficients of $b(X)$ needed during key-switching, which necessitates computing an inverse NTT) operate on each coefficient in a SIMD manner, making the order the coefficients are stored irrelevant. However, there is one crucial homomorphic operation, ciphertext rotation, that depends on the order of stored data in double-CRT format. As discussed earlier, in CKKS and BGV/BFV, a single ciphertext can encrypt many plaintext values, which are then operated on homomorphically in a SIMD manner. This is referred to as ciphertext packing and is crucial for improved throughput. In more detail, in CKKS², a vector of (up to) $N/2$ complex numbers is mapped to a polynomial $m(X) \in \mathbb{Z}[X]/(X^N + 1)$ via the canonical embedding [CKKS17]. By applying an automorphism in the Galois group $\text{Gal}(\mathbb{Q}(\zeta)/\mathbb{Q})$, where ζ is a primitive $2N$ th root of unity in $\mathbb{C}$, one can rotate the plaintext slots, allowing for interaction

²CKKS and BGV/BFV have different plaintext encodings and for the purposes of this discussion, we will focus on CKKS, although our results also extend to BGV/BFV.

between the plaintext values stored in different slots during homomorphic computation. Such rotation operations are crucial for efficiently evaluating many functions and for bootstrapping, which requires many rotations. The Galois group is generated by two automorphisms, $\zeta \rightarrow \zeta^5$ and $\zeta \rightarrow \zeta^{-1}$. If the plaintext slots are ordered such that a polynomial $m(X)$ encodes the values $m(\zeta), m(\zeta^5), m(\zeta^{5^2}), \dots, m(\zeta^{5^{N/2-1}})$, then applying the automorphism $\zeta \rightarrow \zeta^5$ results in the polynomial $m(X^5)$, which encodes the values $m(\zeta^5), m(\zeta^{5^2}), \dots, m(\zeta^{5^{N/2-1}}), m(\zeta)$. Thus, applying this automorphism has rotated the plaintext slots by 1. It turns out that one can similarly apply these automorphisms to the ciphertext polynomials, allowing one to homomorphically rotate the plaintext slots. However, when the ciphertext polynomials are stored in double-CRT format under the standard order, the evaluations are stored at the points³ $\omega, \omega^3, \dots, \omega^{2N-1}$ in this order, and to apply the $\omega \rightarrow \omega^5$ automorphism one must perform a permutation of all the data. As we discuss below, this permutation is a substantial cost in FHE hardware accelerators.

1.1 Our Contributions

We make the following contributions. First, as previously mentioned, we characterize exactly which orderings of the primitive $2N$ th roots of unity admit an in-place butterfly algorithm. We show (Theorem 1 of Section 3.3) that precisely 2^{N-1} of the $N!$ orderings admit an in-place butterfly algorithm, and we give a simple recursive condition characterizing them.

Second, we give in-place butterfly algorithms for negacyclic NTT and inverse NTT that operate on the primitive $2N$ th roots of unity in the order $\omega, \omega^{-1}, \omega^5, \omega^{-5}, \omega^{5^2}, \omega^{-5^2}, \dots, \omega^{-5^{N/2-1}}$. These algorithms are “automorphism-compatible” in the sense that any automorphism of the Galois group can be applied by simple data movement (in the double-CRT format), as opposed to the “arbitrary” looking permutation of the data previously required. For example, $\omega \rightarrow \omega^{5^j}$ can be applied by left rotating the (double-CRT format) data by $2j$ and $\omega \rightarrow \omega^{-1}$ can be applied by pairwise swapping adjacent data. Figures 1 and 2 show the difference in data movement required to perform rotation and conjugation, respectively, when the data is stored in the standard order (as in prior work) vs. the automorphism-compatible order. Our algorithms take the same structure as the Cooley-Tukey and Gentleman-Sande algorithms, differing only in their precomputed twiddle factors, so that they can be used in FHE in place of the standard algorithms while reusing the same hardware.

Third, we describe the application of these algorithms to FHE. Using them, ciphertext rotations can be performed via simple data movement, which removes the need in FHE hardware accelerators for a dedicated automorphism functional unit and frees the on-chip data-movement budget it would otherwise consume, all while adding no cost to any other operation. Finally, we implement our algorithms for negacyclic NTT and inverse NTT and use them in a CKKS implementation, demonstrating that our algorithms work in practice.

1.2 Related Work

Butterfly algorithms in the automorphism-compatible order. The algorithmic work most related to our new algorithm is a butterfly algorithm due to Chen, Chillotti, and Song [CCS19] for homomorphic evaluation of the CKKS encoding and decoding linear-transforms that are used in bootstrapping (CoeffsToSlots and SlotsToCoeffs). Their transform, however, is the CKKS *canonical-embedding* transform, which evaluates only at the $N/2$ roots $\zeta, \zeta^5, \dots, \zeta^{5^{N/2-1}}$ (the positive powers of 5). This suffices for the embedding of plaintexts, as the plaintexts in CKKS are assumed to come in conjugate pairs⁴. It can

³Note, ζ is a primitive $2N$ th root of unity in $\mathbb{C}$, whereas ω is a primitive $2N$ th root in a finite field.

⁴The conjugate pairs assumption on the plaintext complex data is one way to assure that the embeddings can then be mapped to $\mathcal{R}_Q$ which has N coefficients in $\mathbb{Z}_Q$.

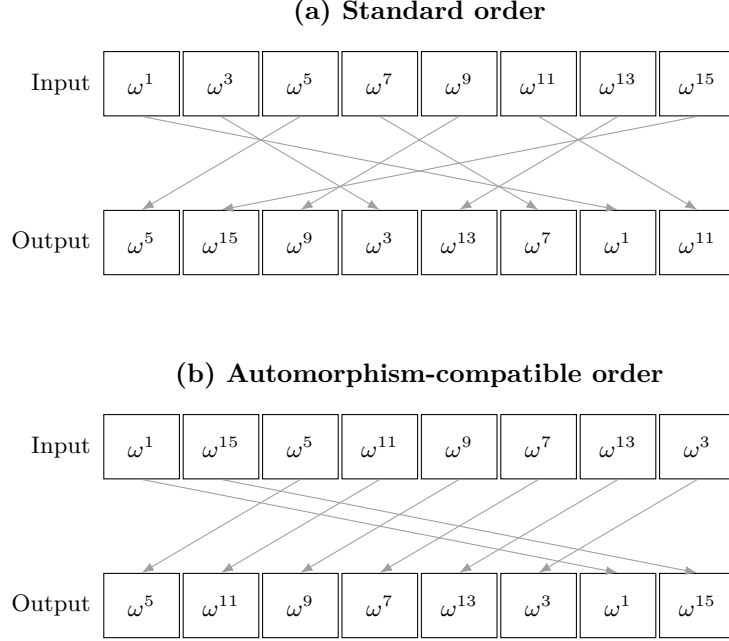


Figure 1: Effect of the Galois automorphism $\omega \mapsto \omega^5$ (a rotation of the plaintext slots by one) on a ciphertext stored in double-CRT format, for $N = 8$. The input row lists the powers of ω (a primitive 16th root of unity) at which the polynomial is evaluated, in the stored order. The automorphism sends the value evaluated at ω^j to the value evaluated at $\omega^{5j \bmod 16}$, and the output row shows the resulting stored order of evaluation points (the vertical arrows denote the automorphism acting on each stored value). In the standard order, the output is an arbitrary-looking reordering of the input whereas in our automorphism-compatible order, the output is a simple left rotate of the data.

thus be viewed as a butterfly algorithm for (evaluation at) *half* of the primitive roots of unity for polynomials of degree $N/2$ (either over $\mathbb{C}$ or a finite field), recalling that the other half of the set of primitive roots are $\zeta^{-1}, \zeta^{-5}, \dots, \zeta^{-5^{N/2-1}}$. It is worth noting that $\zeta^{5^{N/2}} = \zeta$, when ζ is a $2N$ th primitive root of unity. One can use this algorithm to evaluate at all primitive roots, but that would require running the algorithm twice on polynomials of degree N , once with ζ and again with ζ^{-1} .

The present work resolves this with a significant innovation: we integrate the evaluations at *all* the primitive $2N$ th roots of unity (both the positive and negative powers of 5) into a single in-place butterfly algorithm (Section 3) that has the identical structure to the standard NTT and differs only in its precomputed twiddles. This makes the automorphism-compatible order usable not only for the homomorphic linear transforms of bootstrapping but as the NTT used throughout an FHE implementation.

Hardware acceleration of FHE. The high cost of FHE has motivated a large body of work on dedicated FHE accelerators, spanning FPGAs, GPUs, and custom ASICs. On FPGAs, HEAX [RLPD20] accelerates the key-switching linear algebra for CKKS, while FAB [AdCY⁺23], Poseidon [YZF⁺23], and Medha [MAK⁺23] target full bootstrappable pipelines. On GPUs, the memory-centric optimizations of Jung et al. [JKA⁺21] achieved more than a $100\times$ speedup for CKKS bootstrapping, and the design has been carried forward by subsequent GPU libraries. The most aggressive designs are custom ASICs: F1 [SFK⁺21], the first programmable FHE accelerator, followed by Crater-

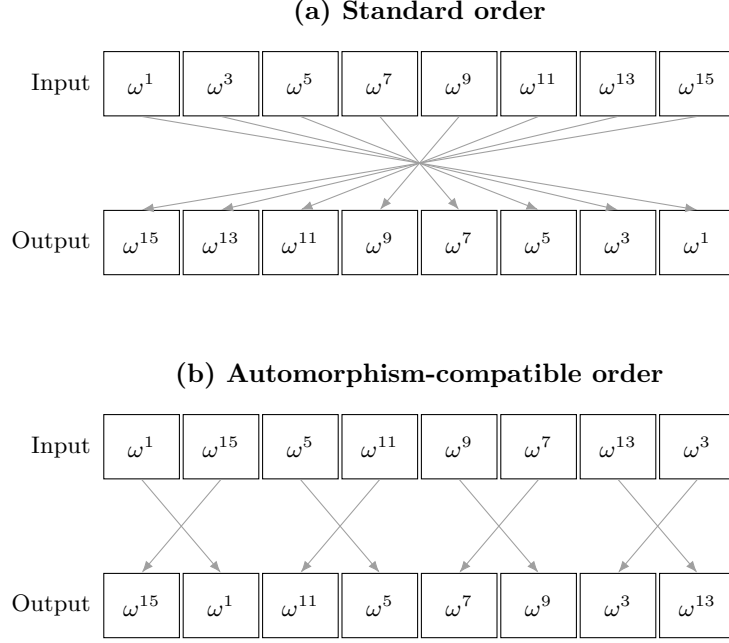


Figure 2: Effect of the conjugation automorphism $\omega \mapsto \omega^{-1}$ on a ciphertext stored in double-CRT format, for $N = 8$. As in Figure 1, the input row lists the evaluation points in the stored order, and the output row shows the resulting points after the automorphism, which sends the value evaluated at ω^j to the value evaluated at $\omega^{-j \bmod 16}$. In the standard order, conjugation reverses the entire array whereas in our automorphism-compatible order, conjugation merely swaps each adjacent pair of values.

Lake [SFK⁺22], BTS [KKK⁺22], ARK [KLK⁺22], SHARP [KKC⁺23], CiFHER [KKCA24], and BASALISC [GBP⁺23]. Across all of these platforms, the NTT and the Galois automorphisms are the only operations that induce data dependencies among the elements of a single residue polynomial (as we discuss in Section 4), and they are consequently the two operations that dominate the on-chip data-movement budget.

2 Preliminaries

Let $\mathbb{F}$ denote an arbitrary field. Let $\mathbb{C}$ denote the field of complex numbers and $\mathbb{F}_p$ denote the finite field of p elements, for a prime p . Let $[N]$ denote the set $\{0, 1, \dots, N-1\}$.

2.1 Negacyclic NTT and DFT

Fix a field $\mathbb{F}$ such as a finite field or the field of complex numbers. Let N be a power of two, say 2^n . Let $\{f_i\}_{i \in [N]}$ be a vector of elements of $\mathbb{F}$. The negacyclic number-theoretic transform (NTT) of $\{f_i\}_i$ is defined as the evaluation of the degree $N-1$ polynomial $f(X) = f_0 + f_1X + \dots + f_{N-1}X^{N-1}$ at the $2N$ th primitive roots of unity, i.e. the roots of $X^N + 1$. Let ω be a $2N$ th primitive root of unity of $\mathbb{F}$. Since N is a power of two, the roots of $X^N + 1$ are given by the odd powers of ω , i.e. $\{\omega^{2j+1}\}_{j \in [N]}$. The output of the negacyclic NTT is the vector $\{\hat{f}_j\}_{j \in [N]}$, where

$$\hat{f}_j = \sum_{i=0}^{N-1} \omega^{(2j+1)*i} * f_i.$$

We use the notation $\text{ngNTT}(j \in [N], \{f_i\}_i, N, 2N)$ to denote this transformation. More generally, we use the notation $\text{ngNTT}(j \in J, \{f_i\}_i, m, n)$ to denote the set of transforms, one for each j in J , given by

$$\hat{f}_j = \sum_{i=0}^{m-1} \omega^{(2j+1)*i} * f_i,$$

where ω is a primitive n th root of unity in $\mathbb{F}$. When the last argument is clear from context, we will drop it. When $\mathbb{F} = \mathbb{C}$, the above transform is referred to as the discrete Fourier transform (DFT), whereas when $\mathbb{F}$ is a finite field, it is referred to as the number-theoretic transform (NTT). As the algorithms for computing the above transform and its inverse are analogous when $\mathbb{F}$ is a finite field or the complex numbers, we will generally use the NTT terminology throughout this work.

2.2 In-Place Butterfly Algorithms for NTT

A butterfly algorithm computes the NTT in $O(N \log N)$ time by recursively splitting the transform into two half-size transforms and combining their outputs using *twiddle factors* (fixed powers of ω) via 2×2 butterfly operations. The Cooley-Tukey algorithm [CT65] uses a butterfly operation of the form $(a, b) \mapsto (a + \omega^j b, a - \omega^j b)$. Such algorithms are in-place when each butterfly overwrites its two inputs with its two outputs, so that the entire transform is computed using only $O(1)$ auxiliary storage.

3 Automorphism-Compatible NTT

In this section, we compute an automorphism-compatible NTT. Let ω be a $2N$ th primitive root of unity in $\mathbb{F}$. When using the automorphism-compatible NTT to convert between the coefficient and evaluation representations of ciphertext polynomials, we will have $\mathbb{F}$ be $\mathbb{F}_q$ for some prime $q \equiv 1 \pmod{2N}$, where q is the prime associated with the a particular limb. Alternatively, we can have $\mathbb{F} = \mathbb{C}$ when the automorphism-compatible NTT is used for encoding/decoding of the plaintext in CKKS.

Recall that in the definition of ngNTT on input $f(X) = f_0 + f_1 X + \dots + f_{N-1} X^{N-1}$, the j th output was defined as $\hat{f}_j = f(\omega^{2j+1})$. We will refer to our automorphism-compatible NTT by ngNTT^* , which is the same as ngNTT except that the output is in a different order. In ngNTT^* , the output is ordered as evaluations on the primitive roots $\omega, \omega^{-1}, \omega^5, \omega^{-5}, \dots, \omega^{5^{N/2-1}}, \omega^{-5^{N/2-1}}$. Recall that since $\mathbb{Z}_{2N}^* \cong \mathbb{Z}_{N/2} \times \mathbb{Z}_2$ and generated by 5 and -1 which have orders $N/2$ and 2, respectively, this is simply a reordering of the odd powers of ω .

3.1 In-Place Butterfly Algorithm: Decimation in Time

We derive in-place butterfly algorithms for computing ngNTT^* . We begin with a decimation in time algorithm, which can be viewed as an analogue of the Cooley-Tukey algorithm [CT65], but with the automorphism-compatible output order. We begin by splitting ngNTT^* into two transforms, ngNTT^+ and ngNTT^- , that represent the evaluation of $f(X)$ at the primitive roots of unity that are the positive and negative powers of 5, respectively. Let $\text{ngNTT}^+(J, \{f_i\}_i, m, 2N)$ be the set of transforms of $\{f_i\}_i$ (a sequence of length m indexed by i), one for each j in J , given by

$$\sum_{i=0}^{m-1} \omega^{5^j * i} * f_i,$$

where ω is a primitive $2N$ th root of unity in $\mathbb{F}$. Recall that $j \in [N/2]$ since $5^{N/2} \equiv 1 \pmod{2N}$. It follows that

$$\begin{aligned}\hat{f}_j &= \sum_{i=0}^{N-1} \omega^{5^j * i} * f_i = \sum_{i=0}^{N/2-1} \omega^{5^j * (2i)} * f_{2i} + \sum_{i=0}^{N/2-1} \omega^{5^j * (2i+1)} * f_{2i+1} \\ &= \sum_{i=0}^{N/2-1} \omega^{5^j * (2i)} * f_{2i} + \omega^{5^j} * \sum_{i=0}^{N/2-1} \omega^{5^j * (2i)} * f_{2i+1} \\ &= \sum_{i=0}^{N/2-1} (\omega^2)^{5^j * i} * f_{2i} + \omega^{5^j} * \sum_{i=0}^{N/2-1} (\omega^2)^{5^j * i} * f_{2i+1}\end{aligned}$$

Moreover, $\omega^{5^{j+N/4} \pmod{2N}} = \omega^{5^j \cdot 5^{N/4}} = \omega^{5^j \cdot (N+1)} = \omega^{5^j} \cdot (\omega^N)^{5^j} = -\omega^{5^j}$, since $\omega^N = -1$ and $5^{N/4} \equiv N+1 \pmod{2N}$ for N a power of two. Thus,

$$\begin{aligned}&\text{ngNTT}^+([N/2], \{f_i\}_i, N, 2N) \\ &= \begin{bmatrix} \text{ngNTT}^+([N/4], \{f_{2i}\}_i, N/2, N) & + & \omega^{5^j} * \text{ngNTT}^+([N/4], \{f_{2i+1}\}_i, N/2, N) \\ \text{ngNTT}^+([N/4], \{f_{2i}\}_i, N/2, N) & - & \omega^{5^j} * \text{ngNTT}^+([N/4], \{f_{2i+1}\}_i, N/2, N) \end{bmatrix}\end{aligned}$$

These will be the even-indexed outputs in ngNTT^* . Converting indices, we have

$$\begin{aligned}&\text{ngNTT}^*(2 * [N/2], \{f_i\}_i, N, 2N) \\ &= \begin{bmatrix} \text{ngNTT}^*(2 * [N/4], \{f_{2i}\}_i, N/2, N) & + & \omega^{5^{j/2}} * \text{ngNTT}^*(2 * [N/4], \{f_{2i+1}\}_i, N/2, N) \\ \text{ngNTT}^*(2 * [N/4], \{f_{2i}\}_i, N/2, N) & - & \omega^{5^{j/2}} * \text{ngNTT}^*(2 * [N/4], \{f_{2i+1}\}_i, N/2, N) \end{bmatrix}\end{aligned}$$

The notation $2 * [N/2]$ means that $\{0, 2, 4, \dots, N\}$. This was half of the transform ngNTT^* . The other half is given by

$$\begin{aligned}\hat{f}_j &= \sum_{i=0}^{N-1} \omega^{-5^j * i} * f_i = \sum_{i=0}^{N/2-1} \omega^{-5^j * (2i)} * f_{2i} + \sum_{i=0}^{N/2-1} \omega^{-5^j * (2i+1)} * f_{2i+1} \\ &= \sum_{i=0}^{N/2-1} \omega^{-5^j * (2i)} * f_{2i} + \omega^{-5^j} * \sum_{i=0}^{N/2-1} \omega^{-5^j * (2i)} * f_{2i+1} \\ &= \sum_{i=0}^{N/2-1} (\omega^2)^{-5^j * i} * f_{2i} + \omega^{-5^j} * \sum_{i=0}^{N/2-1} (\omega^2)^{-5^j * i} * f_{2i+1}\end{aligned}$$

Refer to this transform by ngNTT^- . As before, we also have $\omega^{-5^{j+N/4}} = \omega^{-5^j \cdot 5^{N/4}} = -\omega^{-5^j}$. Thus,

$$\begin{aligned}&\text{ngNTT}^-([N/2], \{f_i\}_i, N, 2N) \\ &= \begin{bmatrix} \text{ngNTT}^-([N/4], \{f_{2i}\}_i, N/2, N) & + & \omega^{-5^j} * \text{ngNTT}^-([N/4], \{f_{2i+1}\}_i, N/2, N) \\ \text{ngNTT}^-([N/4], \{f_{2i}\}_i, N/2, N) & - & \omega^{-5^j} * \text{ngNTT}^-([N/4], \{f_{2i+1}\}_i, N/2, N) \end{bmatrix}\end{aligned}$$

These will be the odd-indexed outputs in ngNTT^* . Converting indices, we have

$$\begin{aligned}&\text{ngNTT}^*(2 * [N/2] + 1, \{f_i\}_i, N, 2N) \\ &= \begin{bmatrix} \text{ngNTT}^*(2 * [N/4] + 1, \{f_{2i}\}_i, N/2, N) + \omega^{-5^{\lfloor j/2 \rfloor}} * \text{ngNTT}^*(2 * [N/4] + 1, \{f_{2i+1}\}_i, N/2, N) \\ \text{ngNTT}^*(2 * [N/4] + 1, \{f_{2i}\}_i, N/2, N) - \omega^{-5^{\lfloor j/2 \rfloor}} * \text{ngNTT}^*(2 * [N/4] + 1, \{f_{2i+1}\}_i, N/2, N) \end{bmatrix}\end{aligned}$$

We can combine these both into one recursion. Note that when j is even, we use $\pm\omega^{5^{\lfloor j/2 \rfloor}}$ as the twiddle factors and when j is odd, we use $\pm\omega^{-5^{\lfloor j/2 \rfloor}}$. So, we can use $\pm\omega^{(-1)^j 5^{\lfloor j/2 \rfloor}}$ as twiddles for the combined recursion. Thus,

$$\begin{aligned} & \text{ngNTT}^*([N], \{f_i\}_i, N, 2N) \\ &= \begin{bmatrix} \text{ngNTT}^*([N/2], \{f_{2i}\}_i, N/2, N) & + & \omega^{(-1)^j 5^{\lfloor j/2 \rfloor}} * \text{ngNTT}^*([N/2], \{f_{2i+1}\}_i, N/2, N) \\ \text{ngNTT}^*([N/2], \{f_{2i}\}_i, N/2, N) & - & \omega^{(-1)^j 5^{\lfloor j/2 \rfloor}} * \text{ngNTT}^*([N/2], \{f_{2i+1}\}_i, N/2, N) \end{bmatrix} \end{aligned}$$

Generalizing this leads to an in-place butterfly algorithm with the output in the correct order, but the input needs to be bit-reversed. We have the following pseudocode for an in-place butterfly algorithm for ngNTT^* .

Algorithm 1 In-place butterfly algorithm for ngNTT^*

Require: Input vector X of length N , a power of 2. ω is a primitive $2N$ th root of unity.

Ensure: Output X contains ngNTT^* of input.

```

1:  $X \leftarrow \text{bitrev}(X)$  ▷ input needs to be index bit-reversed
2: for  $m = 1$  to  $\log N$  do
3:    $M \leftarrow 2^m$ 
4:    $\omega_+ \leftarrow \omega^{N/M}$ 
5:    $\omega_- \leftarrow \omega_+^{-1}$ 
6:   for  $k = 0$  to  $N - 1$  increment by  $M$  do
7:      $\theta_+ \leftarrow \omega_+$ 
8:      $\theta_- \leftarrow \omega_-$ 
9:      $\theta \leftarrow \theta_+$ 
10:    for  $j = 0$  to  $M/2 - 1$  do
11:       $t \leftarrow \theta * X[k + j + M/2]$ 
12:       $u \leftarrow X[k + j]$ 
13:       $X[k + j] \leftarrow u + t$ 
14:       $X[k + j + M/2] \leftarrow u - t$ 
15:      if  $j \bmod 2 == 0$  then ▷ going from plus to minus
16:         $\theta_+ \leftarrow (\theta_+)^5$ 
17:         $\theta \leftarrow \theta_-$ 
18:      else if  $j \bmod 2 == 1$  then ▷ going from minus to plus
19:         $\theta_- \leftarrow (\theta_-)^5$ 
20:         $\theta \leftarrow \theta_+$ 
21:      end if
22:    end for
23:  end for
24: end for
25: return  $X$ 

```

Algorithm 1 has the same structure as the Cooley-Tukey algorithm, but uses different twiddle factors to get the output to occur in an automorphism-compatible order. These twiddle factors are commonly precomputed, so any specific NTT implementations/hardware can be easily adapted to use our automorphism-compatible algorithms by simply updating the stored twiddle factors. Letting $M = 2^m$, at the m th iteration of the recursion, the twiddle factors are $\pm(\omega^{N/M})^{(-1)^j 5^{\lfloor j/2 \rfloor}}$, where j goes from 0 to $M/2 - 1$.

We also immediately obtain an in-place butterfly algorithm for the inverse of ngNTT^* . The pseudocode is given in Algorithm 2.

Algorithm 2 In place butterfly algorithm for inverse ngNTT*

Require: Input vector X of length N , a power of 2. ω is a primitive $2N$ th root of unity.

Ensure: Output X contains the inverse ngNTT* of input.

```

1: for  $m = \log N$  to 1 do
2:    $M \leftarrow 2^m$ 
3:    $\omega_+ \leftarrow (\omega^{-1})^{N/M}$ ;  $\omega_- \leftarrow \omega_+^{-1}$ 
4:   for  $k = 0$  to  $N - 1$  increment by  $M$  do
5:      $\theta_+ \leftarrow \omega_+$ ;  $\theta_- \leftarrow \omega_-$ ;  $\theta \leftarrow \theta_+$ 
6:     for  $j = 0$  to  $M/2 - 1$  do
7:        $t \leftarrow X[k + j + M/2]$ ;  $u \leftarrow X[k + j]$ 
8:        $X[k + j] \leftarrow u + t$ ;  $X[k + j + M/2] \leftarrow (u - t) * \theta$ 
9:       if  $j \bmod 2 == 0$  then ▷ going from plus to minus
10:         $\theta_+ \leftarrow (\theta_+)^5$ ;  $\theta \leftarrow \theta_-$ 
11:       else if  $j \bmod 2 == 1$  then ▷ going from minus to plus
12:         $\theta_- \leftarrow (\theta_-)^5$ ;  $\theta \leftarrow \theta_+$ 
13:       end if
14:     end for
15:   end for
16: end for
17: for  $j = 0$  to  $N - 1$  do  $X[j] \leftarrow N^{-1} * X[j]$ 
18: end for
19:  $X \leftarrow \text{bitrev}(X)$  ▷ output needs to be index bit-reversed
20: return  $X$ 

```

3.2 In-Place Butterfly Algorithm: Decimation in Frequency

We now derive an in-place butterfly algorithm for ngNTT* that recurses via decimation in frequency. This algorithm can be viewed as an analogue of the Gentleman-Sande algorithm [GS66], but with the automorphism-compatible output order (bit-reversed). As before, let ω be a $2N$ th primitive root of unity in $\mathbb{F}$ and let ngNTT^+ and ngNTT^- denote the evaluations of $f(X)$ on primitive roots of unity that are the positive and negative powers of 5, respectively. We begin by analyzing ngNTT^+ .

$$\begin{aligned}
 \hat{f}_j &= \sum_{i=0}^{N-1} \omega^{5^j * i} * f_i = \sum_{i=0}^{N/2-1} \omega^{5^j * i} * f_i + \sum_{i=0}^{N/2-1} \omega^{5^j * (i+N/2)} * f_{i+N/2} \\
 &= \sum_{i=0}^{N/2-1} \omega^{5^j * i} * (f_i + \mathbf{i} * f_{i+N/2})
 \end{aligned}$$

since $\omega^{5^j * N/2} = (\omega^{N/2})^{5^j} = (\mathbf{i})^{5^j} = \mathbf{i}$ in $\mathbb{F}$ since $5^j \equiv 1 \pmod{4}$. It follows that $\text{ngNTT}^+([N/2], \{f_i\}_i, N, 2N) = \text{ngNTT}^+([N/2], \{f_i + \mathbf{i} * f_{i+N/2}\}_i, N/2, 2N)$.

Generalizing, we want to compute $\text{ngNTT}^+(\{j \in [N/2]\} \mid j|_k = j', \{f_i\}_i, N/2^{k+1}, 2N)$, where $j|_k = j'$ denotes the k least significant bits of j are j' . Let $j = j' + 2^k b_j + 2^{k+1} j''$, where b_j is the $k+1$ th least significant bit of j . It follows that

$$\begin{aligned}
 \hat{f}_j &= \sum_{i=0}^{N/2^{k+1}-1} \omega^{5^j * i} * f_i = \sum_{i=0}^{N/2^{k+2}-1} \omega^{5^j * i} * f_i + \sum_{i=0}^{N/2^{k+2}-1} \omega^{5^j * (i+N/2^{k+2})} * f_{i+N/2^{k+2}} \\
 &= \sum_{i=0}^{N/2^{k+2}-1} \omega^{5^j * i} * \left(f_i + (-1)^{b_j} (\omega^{N/2^{k+2}})^{5^{j'}} * f_{i+N/2^{k+2}} \right)
 \end{aligned}$$

since $\omega^{5^j * N/2^{k+2}} = (\omega^{N/2^{k+2}})^{5^j}$ and $5^j = 5^{j'+2^k b_j + 2^{k+1} j''} \equiv 5^{j'} * 5^{2^k b_j} \equiv 5^{j'} * (2^{k+2} + 1)^{b_j} \pmod{2^{k+3}}$ because $5^{2^k} \equiv 2^{k+2} + 1 \pmod{2^{k+3}}$. Thus, when $b_j = 0$, $(\omega^{N/2^{k+2}})^{5^j} = (\omega^{N/2^{k+2}})^{5^{j'}}$ and when $b_j = 1$, $(\omega^{N/2^{k+2}})^{5^j} = (\omega^{N/2^{k+2}})^{5^{j'} * (2^{k+2} + 1)} = -(\omega^{N/2^{k+2}})^{5^{j'}}$. So, it follows that

$$\begin{aligned} & \text{ngNTT}^+(\{j \in [N/2]\} \mid j|_k = j', \{f_i\}_i, N/2^{k+1}, 2N) \\ &= \begin{cases} \text{ngNTT}^+(\{j \in [N/2]\} \mid j|_{k+1} = j', \{f_i + (\omega^{N/2^{k+2}})^{5^{j'}} * f_{i+N/2^{k+2}}\}_i, N/2^{k+2}, 2N) \\ \text{ngNTT}^+(\{j \in [N/2]\} \mid j|_{k+1} = 2^k + j', \{f_i - (\omega^{N/2^{k+2}})^{5^{j'}} * f_{i+N/2^{k+2}}\}_i, N/2^{k+2}, 2N) \end{cases} \end{aligned}$$

This gives a butterfly algorithm for half the transformation (the roots ω^{5^j}) where the output is bit reversed, and the input is in the original order. An analogous butterfly algorithm holds for the second half of the transformation (the roots ω^{-5^j}). Skipping detailed calculation, we note that $\text{ngNTT}^-([N/2], \{f_i\}_i, N, 2N) = \text{ngNTT}^-([N/2], \{f_i - \mathbf{i} * f_{i+N/2}\}_i, N/2, 2N)$. And generalizing, we have

$$\begin{aligned} & \text{ngNTT}^-(\{j \in [N/2]\} \mid j|_k = j', \{f_i\}_i, N/2^{k+1}, 2N) \\ &= \begin{cases} \text{ngNTT}^-(\{j \in [N/2]\} \mid j|_{k+1} = j', \{f_i + (\omega^{N/2^{k+2}})^{-5^{j'}} * f_{i+N/2^{k+2}}\}_i, N/2^{k+2}, 2N) \\ \text{ngNTT}^-(\{j \in [N/2]\} \mid j|_{k+1} = 2^k + j', \{f_i - (\omega^{N/2^{k+2}})^{-5^{j'}} * f_{i+N/2^{k+2}}\}_i, N/2^{k+2}, 2N) \end{cases} \end{aligned}$$

Observe that these butterfly algorithms for ngNTT^+ and ngNTT^- can easily be combined into a single butterfly algorithm for ngNTT^* . Since the above algorithms were decimation in frequency, we immediately divide the output indices into even and odd. But the even output indices of ngNTT^* give ngNTT^+ and the odd output indices give ngNTT^- . In particular, we have

$$\text{ngNTT}^*([N], \{f_i\}_i, N, 2N) = \begin{cases} \text{ngNTT}^+([N/2], \{f_i + \mathbf{i} * f_{i+N/2}\}_i, N/2, 2N) \\ \text{ngNTT}^-([N/2], \{f_i - \mathbf{i} * f_{i+N/2}\}_i, N/2, 2N) \end{cases}$$

3.3 Order of Roots Requirements

Let $\{\omega_j\}_{j \in [N]}$ be the set of points that we wish to perform the negacyclic number-theoretic transform on (these are the primitive $2N$ -th roots of unity). We study the orderings of these roots that admit in-place butterfly algorithms. Let ω denote a base primitive $2N$ -th root of unity. We can write $\omega_j = \omega^{f(j)}$, where $f : [N] \rightarrow 2 * [N] + 1$ is a bijection. That is, f maps the indices to odd integers modulo $2N$. We show the following theorem.

Theorem 1. *Let $f : [N] \rightarrow 2 * [N] + 1$ be a bijection. The number of bijections that support the Cooley-Tukey butterfly algorithm are 2^{N-1} and are those that satisfy for all $k \in [\log N]$*

$$\omega^{2^k f(j)} = -\omega^{2^k f(j+N/2^{k+1})}$$

for $j \in [N/2^{k+1}]$. In particular, the standard order of roots and the automorphism-compatible order of roots both support the Cooley-Tukey butterfly algorithm. The bijections that support the Gentleman-Sande butterfly algorithm are the same as those that support the Cooley-Tukey algorithm. Moreover, the bit-reversed order is not compatible with either butterfly algorithm.

Proof. Examining the derivation for the Cooley-Tukey butterfly algorithm (decimation in time), we note the following requirement for the first iteration.

- $\omega_j = -\omega_{j+N/2}$ for $j \in [N/2]$

Note that this implies that $\omega_j^2 = \omega_{j+N/2}^2$ and that the induced function $g : [N/2] \rightarrow 2 * [N/2] + 1$ on the N -th roots of unity (the odd powers of ω^2 in $[N]$) given by $g(j) \equiv f(j) \bmod N$ is a permutation. For the next level of recursion, we require

- $\omega_j^2 = -\omega_{j+N/4}^2$ for $j \in [N/4]$.

Generally, we require that for all $k \in [\log N]$,

- $\omega_j^{2^k} = -\omega_{j+N/2^{k+1}}^{2^k}$ for $j \in [N/2^{k+1}]$.

So admissible bijections $f : [N] \rightarrow 2 * [N] + 1$ are those that satisfy for all $k \in [\log N]$

$$\omega^{2^k f(j)} = -\omega^{2^k f(j+N/2^{k+1})}$$

for $j \in [N/2^{k+1}]$.

Since $\omega^N = -1$, this implies that $2^k f(j) \equiv N + 2^k f(j + N/2^{k+1}) \bmod 2N$, for $j \in [N/2^{k+1}]$. To count the number of admissible bijections, note that the above implies $f(j) \equiv N/2^k + f(j + N/2^{k+1}) \bmod 2N/2^k$ for $j \in [N/2^{k+1}]$. Once $f(j)$ is determined mod $2N/2^{k+1}$, there are two possible options for $f(j) \bmod 2N/2^k$ and then $f(j + N/2^{k+1}) \bmod 2N/2^k$ is determined by the previous equation.

In general, let $N = 2^n$. Then, $2^n/2^k = 2$ when $k = n - 1$. At recursive level $N/2^k$, there are $N/2^k$ binary choices (symmetry around $N/2^k \bmod 2N/2^{k-1}$). So, the number of binary choices are

$$\sum_{k=0}^{n-1} N/2^{k+1} = \sum_{i=0}^{n-1} 2^i = 2^n - 1 = N - 1.$$

So, the number of admissible bijections f is 2^{N-1} compared to $N!$ total bijections.

In the standard order, $\omega_j = \omega^{2j+1}$ so $f(j) = 2j + 1$. Note that for all $k \in [\log N]$, $j \in [N/2^{k+1}]$,

$$-\omega^{2^k f(j+N/2^{k+1})} = -\omega^{2^k (2(j+N/2^{k+1})+1)} = -\omega^N * \omega^{2^k (2j+1)} = \omega^{2^k f(j)}$$

and so it is an admissible bijection.

In the automorphism-compatible order, $f(j) = (-1)^j 5^{\lfloor j/2 \rfloor} \bmod 2N$. It follows that

$$\begin{aligned} -\omega^{2^k f(j+N/2^{k+1})} &= -\omega^{2^k (-1)^{j+N/2^{k+1}} 5^{\lfloor j/2 + N/2^{k+2} \rfloor}} \\ &= -\omega^{2^k (-1)^j 5^{\lfloor j/2 \rfloor} 5^{N/2^{k+2}}} \\ &= -\omega^{2^k f(j) 5^{N/2^{k+2}}} \\ &= -\omega^{2^k f(j) (N/2^k + 1)} \\ &= -\omega^{2^k f(j)} * (-1)^{f(j)} \\ &= \omega^{2^k f(j)} \end{aligned}$$

for $k \in [\log N - 1]$, using the fact that $5^{N/4} \equiv N + 1 \bmod 2N$. For $k = \log N - 1$ and $j = 0$,

$$\begin{aligned} -\omega^{2^k f(j+N/2^{k+1})} &= -\omega^{N/2 * (-1) * 5^{\lfloor 1/2 \rfloor}} \\ &= \omega^{N/2} = \omega^{2^k f(j)} \end{aligned}$$

since $(\omega^{N/2})^{-1} = -\omega^{N/2}$.

Next, we examine the requirements for the Gentleman-Sande butterfly algorithm (decimation in frequency) and show that the admissible bijections f are the same as those for the Cooley-Tukey algorithm. For the Gentleman-Sande butterfly algorithm, we have the following requirement (for k from 1 to $\log N$)

- $(\omega^{f(j)})^{N/2^k} = (-1)^{j_k} (\omega^{N/2^k})^{g(j_{|k-1})}$, where j_k denotes the k th least significant bit of j , $j_{|k-1}$ denotes the 0th to $k-1$ th least significant bits of j and g is some function. Note that the most significant bits are irrelevant (the same relationship must hold for all j 's with the same most significant bits beyond the k th bit).

By the same argument as for the Cooley-Tukey algorithm, this implies that (for k from 1 to $\log N$)

- $(\omega^{f(j)})^{N/2^k} = -(\omega^{f(j+2^{k-1})})^{N/2^k}$

for $j \in [2^{k-1}]$. Observe that this is the exact same condition on f that was required for the Cooley-Tukey algorithm after a change of variables $k' = \log N - k$.

Finally, we show that the bit-reversed order is not supported by these algorithms. In the bit-reversed order, $f(j) = 2 * \text{bitrev}(j) + 1$. To see that this is not an admissible bijection, consider the case $N = 4$. Then, the order of the 4 primitive 8th roots of unity is $\omega, \omega^5, \omega^3, \omega^7$. But, $\omega_0 = \omega \neq -\omega^3 = -\omega_2$ so the required condition is not satisfied. This is consistent with the fact that these butterfly algorithms must have either the input or output in bit-reversed order [Loa92], and we cannot use a bit-reversed order of the roots to cancel out the inherent bit-reversal of the output of the Gentleman-Sande algorithm and obtain an in-place butterfly algorithm with both the input and output in the standard order. $\square$

4 FHE using Automorphism-Compatible NTT

In this section, we review the homomorphic operations supported by the packed FHE schemes CKKS and BGV/BFV and show that, except for automorphisms, all homomorphic operations are oblivious to the order in which the evaluations of a ciphertext polynomial are stored in double-CRT format. Thus, the automorphism-compatible NTT of Section 3 can serve as a drop-in replacement for NTT in FHE: reordering the evaluation points changes only the automorphism step, which our ordering in fact simplifies, while leaving every other operation unaffected.

4.1 Double-CRT Representation

Recall that in both CKKS and BGV/BFV, a ciphertext is a pair $\text{ct} = (c_0, c_1) \in \mathcal{R}_Q^2$, where $\mathcal{R} := \mathbb{Z}[X]/(X^N + 1)$ with $N = 2^n$ a power of two, $Q = \prod_{i=0}^{\ell-1} q_i$ is a product of distinct primes with $q_i \equiv 1 \pmod{2N}$, and $\mathcal{R}_Q := \mathcal{R}/Q\mathcal{R}$. The Chinese remainder theorem gives the ring isomorphism

$$\mathcal{R}_Q \cong \mathcal{R}_{q_0} \times \mathcal{R}_{q_1} \times \cdots \times \mathcal{R}_{q_{\ell-1}},$$

so a polynomial $a(X) \in \mathcal{R}_Q$ is stored as its ℓ limbs $a^{(i)}(X) := a(X) \pmod{q_i} \in \mathcal{R}_{q_i}$. Within each limb, because $q_i \equiv 1 \pmod{2N}$, all $2N$ th primitive roots of unity lie in $\mathbb{Z}_{q_i}$, and the negacyclic NTT maps the coefficient vector $(a_0^{(i)}, a_1^{(i)}, \dots, a_{N-1}^{(i)})$ of $a^{(i)}(X)$ (called the coefficient representation) to the vector of evaluations $(a^{(i)}(\omega_0), a^{(i)}(\omega_1), \dots, a^{(i)}(\omega_{N-1}))$ at the N primitive $2N$ th roots of unity $\{\omega_j\}_{j \in [N]}$. Storing each limb by its NTT evaluations is the *double-CRT* (or evaluation) representation. Prior work stored the ciphertext data as a vector of evaluations at evaluation points in the standard order $\omega_j = \omega^{2j+1}$ since this is the order output by in-place butterfly algorithms for negacyclic NTT. However, as we showed in Section 3, by using different twiddle factors, the output of the negacyclic NTT can be in our automorphism-compatible order $\omega_j = \omega^{(-1)^j 5^{\lfloor j/2 \rfloor}}$. We now show that (other than the automorphism operation), the other homomorphic operations do not depend on the order of the evaluation points.

4.2 Coordinate-Wise Operations: Addition and Multiplication

The most basic homomorphic operations are addition and multiplication of ciphertexts, along with addition and multiplication by scalars.

Homomorphic addition. Given $\text{ct} = (c_0, c_1)$ and $\text{ct}' = (c'_0, c'_1)$, homomorphic addition outputs $(c_0 + c'_0, c_1 + c'_1) \in \mathcal{R}_Q^2$. In double-CRT form, each component is the sum of the corresponding limbs, and since the NTT is a linear map on each limb, addition is performed coordinate-wise:

$$(c_b + c'_b)(\omega_j) = c_b(\omega_j) + c'_b(\omega_j) \quad \text{for every } j \in [N].$$

The value written to coordinate j of the result depends only on the two inputs at coordinate j ; it makes no reference to which root ω_j that slot represents. Hence addition is order-oblivious.

Homomorphic multiplication. Multiplication of two ciphertexts produces a degree-2 ciphertext whose components are the ring products $c_0 c'_0$, $c_0 c'_1 + c_1 c'_0$, and $c_1 c'_1$ in $\mathcal{R}_Q$. The point of evaluating on the $2N$ th roots of unity is that negacyclic polynomial multiplication in $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ becomes pointwise multiplication of evaluations: for any $a, b \in \mathcal{R}_{q_i}$,

$$(a \cdot b)(\omega_j) = a(\omega_j) b(\omega_j) \quad \text{for every } j \in [N].$$

As with addition, the value in coordinate j of the product is a function of the two inputs in coordinate j alone, independent of the identity of ω_j . Multiplication is therefore order-oblivious. The same argument covers plaintext–ciphertext addition and multiplication and scalar operations, all of which are coordinate-wise.

The reason these operations are order-oblivious is that these operations are ring homomorphisms evaluated under the CRT isomorphism $\mathcal{R}_{q_i} \cong \prod_{j \in [N]} \mathbb{F}_{q_i}[X]/(X - \omega_j)$. Each factor $\mathbb{F}_{q_i}[X]/(X - \omega_j) \cong \mathbb{F}_{q_i}$ is an evaluation at a single point, and a homomorphism into a product of rings acts independently in each coordinate.

4.3 Operations on RNS Limbs: Rescaling and Modulus Switching

The next family of operations manipulates the RNS/CRT structure across limbs: CKKS rescaling, BGV/BFV modulus switching, and the modulus-raising and modulus-reduction steps (ModUp and ModDown) used inside key-switching.

Rescaling and modulus switching. CKKS rescaling divides a ciphertext by a prime $q_{\ell-1}$ and drops that limb, mapping $\mathcal{R}_Q^2 \rightarrow \mathcal{R}_{Q/q_{\ell-1}}^2$; BGV modulus switching is the analogous scale-and-round from modulus Q to $Q' \mid Q$. In double-CRT form these are implemented, for each coordinate j , as the same fixed affine map applied to the tuple of limb values $(a^{(0)}(\omega_j), \dots, a^{(\ell-1)}(\omega_j))$ at that coordinate, followed by dropping one (or more) elements of the tuple. The map applied is identical for every coordinate j and does not depend on j ; it only depends on the limb index. Therefore, rescaling and modulus switching are order-oblivious.

ModUp and ModDown. The RNS basis-extension routines used in key-switching, ModUp (extend the RNS basis by additional primes) and ModDown (reduce the basis), are the one place where one might expect the ordering to matter, because they require computation on the polynomial *coefficients*. However, the coefficient-domain computation is performed coefficient by coefficient in a SIMD manner. Thus, while ModUp/ModDown must temporarily leave the evaluation domain, the coefficient-domain step treats all N

coefficients identically, so the order in which the coefficients are stored is irrelevant. In particular, when we use one of our in-place butterfly algorithms, the inverse NTT leaves the coefficients in bit-reversed order; because the subsequent per-coefficient computation is order-oblivious in exactly the same sense as the coordinate-wise operations above, this bit-reversal is immaterial, and the forward NTT returns the data to the evaluation ordering with which we started. No explicit reordering is needed at any point.

4.4 Key-Switching and Relinearization

Key-switching transforms a ciphertext decryptable under one secret key into one decryptable under another; relinearization (reducing a degree-2 product back to degree 1) and the post-processing of automorphisms are its two principal uses. A key-switching key is a set of encryptions swk of the source key under the target key, and key-switching computes an inner product of a decomposition of the input ciphertext component against swk , followed by a ModDown . Every step of this computation (the digit/RNS decomposition of the input limb, the pointwise products against the key-switching key, the accumulation, and the concluding ModDown) is built from the coordinate-wise arithmetic of Section 4.2 and the limb operations of Section 4.3. Each of those constituents is order-oblivious and, therefore, key-switching and relinearization are order-oblivious.

4.5 Automorphisms

The one homomorphic operation whose behavior on double-CRT data depends on the ordering of the evaluation points is the application of a Galois automorphism, which underlies slot rotation/conjugation in CKKS and BGV/BFV. The Galois group $\text{Gal}(\mathbb{Q}(\zeta)/\mathbb{Q}) \cong \mathbb{Z}_{2N}^*$ acts on $\mathcal{R}$ by $\sigma_t : X \mapsto X^t$ for $t \in \mathbb{Z}_{2N}^*$; applied to a plaintext it permutes the slots (a power of the rotation-by-one map $\sigma_5 : X \mapsto X^5$, together with the conjugation $\sigma_{-1} : X \mapsto X^{-1}$). The same automorphism can be applied homomorphically to the ciphertext polynomials. Consider its effect on a single limb $a(X) \in \mathcal{R}_{q_i}$ in the evaluation domain. Applying σ_t sends $a(X)$ to $a(X^t)$, whose evaluation at $\omega_j = \omega^{f(j)}$ for a bijection $f : [N] \rightarrow 2 * [N] + 1$ is

$$a(X^t)|_{X=\omega^{f(j)}} = a(\omega^{t \cdot f(j)}) = a(\omega_{f^{-1}(t \cdot f(j))}).$$

That is, in the evaluation domain, the automorphism σ_t acts as a permutation of the data

$$\pi_t : j \mapsto f^{-1}(t \cdot f(j) \bmod 2N).$$

Note that applying the automorphism σ_t is not order-oblivious and, moreover, the permutation π_t depends on the ordering f . Under the standard order $f(j) = 2j + 1$, the map π_t is an “arbitrary”-looking permutation of $[N]$ with no simple structure. However, under our automorphism-compatible order $f(j) = (-1)^{j5^{\lfloor j/2 \rfloor}}$, the resulting permutations π_t of the data are simple, regular data movements: the rotation automorphism σ_{5^r} can be applied via a left rotate of the data by $2r$ positions, and the conjugation automorphism σ_{-1} can be applied via pairwise swaps of adjacent data entries.

4.6 Bootstrapping

Bootstrapping is the most expensive procedure in CKKS and BGV/BFV and necessary to “refresh” a ciphertext and enable further homomorphic computation. One might worry that it introduces an order dependence beyond that of the automorphism. However, it does not since bootstrapping is a composition of the operations already considered, and it inherits order-obliviousness from them just as key-switching does in Section 4.4. A bootstrapping procedure consists of four stages [CHK⁺18a, HS15]: (i) a modulus raising (ModRaise) that reinterprets the ciphertext at a larger modulus; (ii) *CoeffsToSlots*, a

homomorphic linear transform that moves the coefficients of the underlying plaintext into the slots; (iii) a homomorphic evaluation of the modular-reduction function (in CKKS a scaled sine approximation (*EvalMod*), and in BGV/BFV a digit-extraction polynomial) applied to remove the noise introduced by ModRaise; and (iv) *SlotsToCoeffs*, the inverse homomorphic linear transform.

We examine each stage in turn. ModRaise reinterprets each limb at a larger modulus and is applied per-coefficient across limbs, exactly like the rescaling and modulus-switching operations of Section 4.3, and is thus order-oblivious. The two homomorphic linear transforms, *CoeffsToSlots* and *SlotsToCoeffs*, are implemented entirely from ciphertext rotations, multiplications by plaintext constants, and additions, arranged in a baby-step/giant-step schedule or an FFT-like factorization into stages [CCS19, HK20, HS18]. The plaintext constants they multiply against (the “diagonals” of the transform) are applied coordinate-wise, so the only order-sensitive component in these transforms is the rotation itself. In BGV/BFV, the linear transforms may additionally use the Frobenius automorphism $\sigma : X \mapsto X^p$; this is itself a Galois automorphism and so is subsumed by the same discussion. Finally, *EvalMod* and digit extraction are polynomials evaluated on the ciphertext coordinate-wise, using only ciphertext addition and multiplication and addition and multiplication by plaintext constants [CHK⁺18a, GV23]; they are therefore coordinate-wise operations of the kind in Section 4.2 and are order-oblivious. Since every stage is either order-oblivious outright or order-oblivious apart from its rotations, bootstrapping depends on the ordering of the double-CRT data only through its rotations, of which it uses a great many. One additional point is that for sparsely-packed ciphertexts, bootstrapping performs a subsum, or trace, step to remove the empty slots; this step is itself implemented via automorphisms [CCS19] and so is likewise order-sensitive only through them.

4.7 CKKS Plaintext Encoding

To illustrate the difference between our approach and prior work, we provide, for the CKKS scheme, commutative diagrams showing how a vector of complex numbers is encoded as a ring polynomial and stored in double-CRT format and the effects of automorphisms. Figure 3 shows our encoding while Figure 4 shows how the RNS-CKKS scheme [CHK⁺18b] does the encoding. Observe that while both approaches use the automorphism-compatible order when mapping $\mathbb{C}^{N/2}$ to $\mathbb{R}[X]/(X^N + 1)$ (via ngIFFT^*), our approach uses the automorphism-compatible ngNTT^* to map into $\mathbb{Z}_Q^N$, whereas [CHK⁺18b] uses ngNTT which outputs in the standard order. Note that $\beta_n \circ \text{ngIFFT}^*$ and $\text{ngNTT}^* \circ \beta_n$ can be computed directly by in-place butterfly algorithms as we have made the bit-reversal β_n on the output/input, respectively, explicit. Furthermore, note that our approach enables the automorphism to be performed on double-CRT data by the simple left rotate of the data ($i \rightarrow i - 2 \bmod N$), whereas [CHK⁺18b] require a much more complicated permutation. One additional point is that [CHK⁺18b] actually stores the double-CRT data in bit-reversed order, so they have to perform bit-reversals when applying the automorphism σ .

5 Hardware Implications

By replacing the standard negacyclic NTT and inverse negacyclic NTT algorithms with our automorphism-compatible negacyclic NTT and inverse negacyclic NTT algorithms, one can perform ciphertext rotations by a simple cyclic shift of the data in double-CRT form and conjugation by pairwise swaps of adjacent data, as demonstrated in Section 4.5 (see also Figure 3). This eliminates the need to permute the data according to an “arbitrary”-looking permutation to perform ciphertext rotations, which is required when using the standard NTT. The overall runtime performance gain in software is negligible, since the runtime is dominated by other factors such as the time to compute the NTT itself;

$$\begin{array}{ccccccc}
\mathbb{C}^{N/2} & \xrightarrow{\beta_n \circ \text{ngIFFT}^*} & \mathbb{R}[X]/(X^N + 1) & \xrightarrow{\pi} & \mathbb{Z}_Q[X]/(X^N + 1) & \xrightarrow{\text{ngNTT}^* \circ \beta_n} & \mathbb{Z}_Q^N \\
\downarrow \text{rot}^* & & \downarrow \sigma & & \downarrow \sigma & & \downarrow i \rightarrow i-2 \bmod N \\
\mathbb{C}^{N/2} & \xrightarrow{\beta_n \circ \text{ngIFFT}^*} & \mathbb{R}[X]/(X^N + 1) & \xrightarrow{\pi} & \mathbb{Z}_Q[X]/(X^N + 1) & \xrightarrow{\text{ngNTT}^* \circ \beta_n} & \mathbb{Z}_Q^N
\end{array}$$

Figure 3: Our approach: Commutative diagram for power-of-two cyclotomic field automorphism $\sigma : X \rightarrow X^5$, with Q odd. The map rot^* left rotates the data by 1, i.e. $i \rightarrow i - 1 \bmod N/2$. The map π rounds/scale the coefficients to integers and embeds into $\mathbb{Z}_Q$. The map β_n is the n -bit index-bit-reversal. All the maps are invertible except π , which is also (approximately) invertible if $\mathbb{C}$ elements are small (w.r.t. Q).

$$\begin{array}{ccccccc}
\mathbb{C}^{N/2} & \xrightarrow{\beta_n \circ \text{ngIFFT}^*} & \mathbb{R}[X]/(X^N + 1) & \xrightarrow{\pi \circ \beta_n} & \mathbb{Z}_Q[X]/(X^N + 1) & \xrightarrow{\beta_n \circ \text{ngNTT}} & \mathbb{Z}_Q^N \\
\downarrow \text{rot}^* & & \downarrow \sigma & & \downarrow \sigma & & \downarrow \text{brev}(i) \rightarrow \text{brev}(\sigma(\text{brev}(i))) \\
\mathbb{C}^{N/2} & \xrightarrow{\beta_n \circ \text{ngIFFT}^*} & \mathbb{R}[X]/(X^N + 1) & \xrightarrow{\pi \circ \beta_n} & \mathbb{Z}_Q[X]/(X^N + 1) & \xrightarrow{\beta_n \circ \text{ngNTT}} & \mathbb{Z}_Q^N
\end{array}$$

Figure 4: [CHK⁺18b]: Commutative diagram for power-of-two cyclotomic field automorphism $\sigma : X \rightarrow X^5$, with Q odd. The map rot^* left rotates the data by 1, i.e. $i \rightarrow i - 1 \bmod N/2$. The map π rounds/scale the coefficients to integers and embeds into $\mathbb{Z}_Q$. The map β_n is the n -bit index-bit-reversal. All the maps are invertible except π , which is also (approximately) invertible if $\mathbb{C}$ elements are small (w.r.t. Q).

indeed, existing software libraries such as FullRNS-HEAAN [CHK⁺18b], SEAL [Mic23], OpenFHE [BBB⁺22], and Lattigo [Lat24] implement the automorphism as a gather over the NTT-domain values using a precomputed index table, whose cost is small relative to the surrounding computation. However, our approach does eliminate the need to store precomputed index tables or compute the index mapping “on the fly.” The advantage for FHE hardware accelerators, however, is significant, as we discuss next.

Automorphisms are a substantial hardware cost. As discussed in Section 4, the NTT and the automorphisms are the only operations that induce dependencies among the elements of a single residue polynomial; every other homomorphic operation is coordinate-wise or limb-wise and hence oblivious to the ordering of the evaluation points. Consequently, applying an automorphism is one of the few operations that an accelerator cannot implement as simple lane-parallel arithmetic, and every accelerator that supports ciphertext rotation must pay for the resulting permutation of the double-CRT data in one of two ways: (i) by dedicating a hardware functional unit to it, or (ii) by moving the data over shared on-chip fabric. It is worth emphasizing that, because the NTT is linear, this permutation is applied directly in the NTT (evaluation) domain; no accelerator does the naive approach of an inverse NTT, a coefficient permutation, and a forward NTT. The cost paid is therefore precisely the cost of permuting the double-CRT data, which is exactly what our automorphism-compatible ordering significantly reduces.

Accelerators with a dedicated automorphism unit. The most direct evidence for this cost is that essentially all high-performance FHE accelerators dedicate a functional unit to the automorphism. F1 [SFK⁺21,FSK⁺21] introduced the first vectorized automorphism unit (see Figure 6 and Section 5.1 therein), which realizes the permutation by decomposing it into row and column permutations around a matrix transpose. This automorphism unit takes up 14.6% of the area (0.58 of 3.97 mm²) and 11.3% of the thermal design power (0.99 of 8.75 W) of an F1 compute cluster (Table 2 therein). In CraterLake [SFK⁺22], the automorphism functional unit consumes 9.0 mm², or 3.7% of the area of the functional units (Table 2 therein), and is supported by a dedicated transpose network and a fixed permutation network. CraterLake is designed to support large-depth circuits via bootstrapping, so much of the functional-unit area is instead consumed by the change-of-RNS-basis unit (158.8 mm², 66% of the functional-unit area), which is used extensively during bootstrapping. The automorphism unit is nonetheless a distinct, non-trivial block that our approach would eliminate the need for. ARK [KLK⁺22] and SHARP [KKC⁺23] similarly employ a dedicated multi-stage shuffle network as their automorphism unit. Notably, both store their limbs in bit-reversed order specifically to make the automorphism efficient, which is itself direct evidence that the ordering of the double-CRT data is the crux of the automorphism’s cost. On FPGAs, FAB [AdCY⁺23] and Poseidon [YZF⁺23] likewise include a dedicated automorphism unit (realized as an address-generated memory permutation and a multi-stage sub-vector-swap datapath, respectively) and BASALISC [GBP⁺23] provides functional units for the automorphism as well.

Accelerators that pay via data movement. The remaining accelerators do not name a separate automorphism unit but instead pay the cost as explicit data movement over shared fabric. BTS [KKK⁺22] routes the automorphism permutation over the same crossbar-based network-on-chip that it builds for its NTT, so that “only (i) NTT and the automorphism incur inter-PE data exchanges.” Medha [MAK⁺23] applies the permutation as an on-the-fly-addressed memory read/write in the NTT domain, and GPU accelerators, beginning with the memory-centric design of Jung et al. [JKA⁺21], apply the automorphism as an out-of-place gather kernel over the NTT-domain residues. In each case, the on-chip network bandwidth or memory traffic devoted to the automorphism is a resource that our

automorphism-compatible ordering frees, since a cyclic shift or adjacent-pair swap requires no data-dependent routing.

As a specific example, we consider BTS, which has no dedicated automorphism unit but instead performs the automorphism as a data-dependent, rotation-amount-dependent permutation routed over its shared inter-PE crossbar network, one of only two operations (with the NTT) that use that network at all. This makes the automorphism a significant cost: the network is only 0.82% of chip area but draws roughly 28% of chip power. Quantified against BTS’s own stated parameters ($N = 2^{17}$, 64-bit words), a general automorphism moves roughly half of each 1.05 MB residue polynomial (about 524 KB) across the chip bisection, an exposed latency (the NTT hides its exchange rounds under butterfly compute, but the automorphism has none), whereas our cyclic shift by $2r$ moves at most one 64-coefficient wrap block (about 0.5 KB) and nothing at all when the shift aligns to the per-PE block. This is at least a $1024\times$ reduction in interconnect traffic per residue polynomial, and the per-limb address arithmetic drops from roughly N modular multiplications to a single shared offset. The takeaway is that even accelerators that avoid a dedicated automorphism unit still pay for the automorphism, and our ordering substantially reduces that cost to a data-independent shift.

On-the-fly twiddle generation. Our automorphism-compatible ordering also supports efficient on-the-fly twiddle generation. At a stage of size $M = 2^m \geq 4$, one can partition the twiddles into $L = 2^{\lceil (m-1)/2 \rceil}$ interleaved streams; the identity $(5^{L/2} - 1)^2 \equiv 0 \pmod{2M}$ ensures that each stream is a geometric progression. Each stream therefore advances by multiplication by a precomputed constant, matching the standard-order generator’s one modular multiplication per update while preserving the butterfly access order. The tradeoff is a separate multiplier per stream, giving $2L$ stored field elements instead of $L + 1$ for a standard-order generator with the same number of streams. This additional storage concerns only the $O(\sqrt{N})$ twiddle-generator state: the N -element polynomial array and in-place butterfly computation are unchanged. For $N = 16384$ with 64-bit residues, the largest-stage generator requires 2 KB per RNS prime instead of approximately 1 KB; including the 128 KB polynomial array, this increases combined storage by less than 1% when only the active stage’s generator state is retained.

Summary. Whether an accelerator pays for the automorphism through dedicated permutation hardware (a transpose unit, permutation network, or shuffle network, as in F1, CraterLake, ARK, SHARP, FAB, Poseidon, and BASALISC) or through data movement over shared fabric (as in BTS, Medha, and GPU designs), that cost arises solely because the evaluation points are stored in an order under which the automorphism is an “arbitrary”-looking permutation of the double-CRT data. Because our automorphism-compatible NTT has the identical butterfly structure to the standard NTT and differs only in its precomputed twiddle factors (Section 3), it can be dropped into any of these accelerators in place of the standard NTT (reusing the same hardware and adding no cost to any other operation) while reducing every automorphism to a cyclic shift and/or an adjacent-pair swap. This removes the need for a dedicated automorphism functional unit and frees the on-chip data-movement budget it would otherwise consume.

6 Implementation

To demonstrate that our algorithm works in practice, we modified the public github of the RNS-CKKS scheme [CHK⁺18b] and replaced its NTT, rotation and conjugation algorithms with the ones proposed in the current work. More specifically, we replaced the NTT and inverse-NTT algorithms in the following methods `qiNTTAndEqual`, `qiINTTAndEqual`, `piNTTAndEqual`, `piINTTAndEqual`, and the rotation and conjugation algorithms in the

`leftRot` and `conjugate` methods. All changes were done in the `context.cpp` source file. We validated that all the unit tests pass successfully. Furthermore, we benchmarked the runtime of the rotation operation with our optimized automorphism by data rotation. We used a MacBook Pro, an Apple M4 Pro with 12 cores and 24 GB RAM. The RNS-CKKS scheme was configured with ring dimension $N = 16384$ and a ciphertext at level 10 i.e., with 10 NTT limbs. The runtime for our rotation procedure was 0.101 ms vs. 0.261 ms for the original rotation i.e., a speedup of $2.59\times$. The reported runtime is the average of 1,000 measurements.

7 Conclusion

We gave a complete characterization of the orderings of the primitive $2N$ th roots of unity that admit in-place butterfly algorithms (Cooley-Tukey or Gentleman-Sande style) for the negacyclic NTT. Exactly 2^{N-1} of the $N!$ orderings admit such an algorithm, and we gave a simple recursive condition describing them. Notably, the exact same orderings work for both Cooley-Tukey and Gentleman-Sande style butterfly algorithms. The automorphism-compatible order is one such ordering, so we obtained in-place butterfly algorithms for the negacyclic NTT and its inverse that evaluate at the roots in this order. Since these algorithms have the same structure as the Cooley-Tukey and Gentleman-Sande algorithms, differing only in their precomputed twiddle factors, they can therefore easily replace the standard NTT.

The automorphism-compatible order is well suited to the SIMD FHE schemes CKKS and BGV/BFV since storing ciphertext data in the automorphism-compatible order reduces applying automorphisms to a rotation or an adjacent-pair swap of the data, eliminating the arbitrary-looking permutation that the standard order requires. This removes the need, in FHE hardware accelerators, for the dedicated automorphism functional unit or the equivalent on-chip data movement that current designs devote to this operation, while leaving every other operation unchanged. Our implementation confirms the practicality of the approach, achieving a $2.59\times$ speedup for the ciphertext rotation operation.

AI Disclosure

The authors used Claude Opus 4.8 to perform a literature review, help prepare the references, help estimate the savings in accelerators without a dedicated automorphism unit, and assist with making LaTeX figures. Claude was not used for any technical innovation.

Acknowledgements

The authors thank Bob Philhower for discussing the hardware implications of our automorphism-compatible NTT.

References

- [AdCY⁺23] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Tugce Yazicigil, Anantha P. Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. FAB: an fpga-based accelerator for bootstrappable fully homomorphic encryption. In *HPCA*, pages 882–895. IEEE, 2023.
- [BBB⁺22] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim,

- Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. Openfhe: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915, 2022. <https://eprint.iacr.org/2022/915>.
- [BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *ITCS*, pages 309–325. ACM, 2012.
- [Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012.
- [CCS19] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. In *EUROCRYPT (2)*, volume 11477 of *Lecture Notes in Computer Science*, pages 34–54. Springer, 2019.
- [CHK⁺18a] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *EUROCRYPT (1)*, volume 10820 of *Lecture Notes in Computer Science*, pages 360–384. Springer, 2018.
- [CHK⁺18b] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full rns variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*, pages 347–368. Springer, 2018.
- [CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT (1)*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.
- [CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.
- [FSK⁺21] Axel Feldmann, Nikola Samardzic, Aleksandar Krastev, Srini Devadas, Ronald G. Dreslinski, Karim Eldefrawy, Nicholas Genise, Chris Peikert, and Daniel Sánchez. F1: A fast and programmable accelerator for fully homomorphic encryption (extended version). *CoRR*, abs/2109.05371, 2021.
- [FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012. <https://eprint.iacr.org/2012/144>.
- [Gar59] Harvey L. Garner. The residue number system. *IRE Transactions on Electronic Computers*, EC-8(2):140–147, 1959.
- [GBP⁺23] Robin Geelen, Michiel Van Beirendonck, Hilder V. L. Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios Dimou, Ingrid Verbauwhede, Frederik Vercauteren, and David W. Archer. BASALISC: programmable hardware accelerator for BGV fully homomorphic encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(4):32–57, 2023.
- [Gen09a] Craig Gentry. A fully homomorphic encryption scheme. *Diss. Stanford University*, 2009.

- [Gen09b] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178. ACM, 2009.
- [GS66] W. M. Gentleman and G. Sande. Fast Fourier transforms: For fun and profit. In *Proceedings of the November 7–10, 1966, Fall Joint Computer Conference (AFIPS '66)*, pages 563–578. ACM, 1966.
- [GV23] Robin Geelen and Frederik Vercauteren. Bootstrapping for BGV and BFV revisited. *Journal of Cryptology*, 36(2):12, 2023.
- [HK20] Kyoohyung Han and Dohyeong Ki. Better Bootstrapping for Approximate Homomorphic Encryption. In Stanislaw Jarecki, editor, *Topics in Cryptology – CT-RSA 2020*, pages 364–390, Cham, 2020. Springer International Publishing.
- [HS15] Shai Halevi and Victor Shoup. Bootstrapping for helib. In *EUROCRYPT (1)*, volume 9056 of *Lecture Notes in Computer Science*, pages 641–670. Springer, 2015.
- [HS18] Shai Halevi and Victor Shoup. Faster homomorphic linear transformations in helib. In *CRYPTO (1)*, volume 10991 of *Lecture Notes in Computer Science*, pages 93–120. Springer, 2018.
- [JKA⁺21] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):114–148, 2021.
- [KKC⁺23] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. SHARP: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption. In *ISCA*, pages 18:1–18:15. ACM, 2023.
- [KKCA24] Sangpyo Kim, Jongmin Kim, Jaeyoung Choi, and Jung Ho Ahn. CiFHER: A chiplet-based FHE accelerator with a resizable structure. In *IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 119–130. IEEE, 2024.
- [KKK⁺22] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. BTS: an accelerator for bootstrappable fully homomorphic encryption. In *ISCA*, pages 711–725. ACM, 2022.
- [KLK⁺22] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, John Kim, Minsoo Rhu, and Jung Ho Ahn. ARK: fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse. In *MICRO*, pages 1237–1254. IEEE, 2022.
- [Lat24] Lattigo v6. Online: <https://github.com/tuneinsight/lattigo>, August 2024. EPFL-LDS, Tune Insight SA.
- [Loa92] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*, volume 10 of *Frontiers in Applied Mathematics*. SIAM, Philadelphia, PA, 1992.
- [MAK⁺23] Ahmet Can Mert, Aikata, Sunmin Kwon, Youngsam Shin, Donghoon Yoo, Yongwoo Lee, and Sujoy Sinha Roy. Medha: Microcoded hardware accelerator for computing on encrypted data. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(1):463–500, 2023.

- [Mic23] Microsoft SEAL (release 4.1). <https://github.com/Microsoft/SEAL>, January 2023. Microsoft Research, Redmond, WA.
- [Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6):34:1–34:40, 2009.
- [RLPD20] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. HEAX: an architecture for computing on encrypted data. In *ASPLOS*, pages 1295–1309. ACM, 2020.
- [SFK⁺21] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald G. Dreslinski, Christopher Peikert, and Daniel Sánchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO*, pages 238–252. ACM, 2021.
- [SFK⁺22] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sánchez. Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In *ISCA*, pages 173–187. ACM, 2022.
- [SV14] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *Des. Codes Cryptogr.*, 71(1):57–81, 2014.
- [YZF⁺23] Yinghao Yang, Huaizhi Zhang, Shengyu Fan, Hang Lu, Mingzhe Zhang, and Xiaowei Li. Poseidon: Practical homomorphic encryption accelerator. In *HPCA*, pages 870–881. IEEE, 2023.