PaCo: Bootstrapping for CKKS via Partial CoeffToSlot

Jean-Sébastien Coron(✉) and Tim Seuré

University of Luxembourg, Esch-sur-Alzette, Luxembourg
{jean-sebastien.coron, tim.seure}@uni.lu

Abstract. We introduce PaCo, a novel and efficient bootstrapping procedure for the CKKS homomorphic encryption scheme, where PaCo stands for *(Bootstrapping via) Partial CoeffToSlot*. At a high level, PaCo reformulates the CKKS decryption equation in terms of blind rotations and modular additions. This reformulated decryption circuit is then evaluated homomorphically within the CKKS framework. Our approach makes use of the circle group in the complex plane to simulate modular additions via complex multiplication, and utilizes alternative polynomial ring structures to support blind rotations. These ring structures are enabled by a variant of the CoeffToSlot operation, which we call a *partial CoeffToSlot*. This yields a new bootstrapping approach within CKKS, achieving a computational complexity which is logarithmic in the number of complex slots. We further introduce a parallelized variant that enables bootstrapping over all CKKS slots with enhanced throughput, highlighting PaCo's suitability for practical and large-scale homomorphic applications. In addition to the bootstrapping technique itself, we develop several supporting tools — particularly in the context of bit-reversing and alternative ring structures for CKKS — which can be of independent interest to the community. Finally, a proof-of-concept implementation confirms that PaCo achieves performance competitive with state-of-the-art methods for CKKS bootstrapping.

Keywords: Homomorphic encryption · Bootstrapping · RLWE · Lattices.

1 Introduction

1.1 General Overview

Homomorphic Encryption. Homomorphic encryption (HE) allows computations to be performed directly on encrypted data without the need for decryption, thereby preserving privacy in scenarios such as outsourced data processing. Fully homomorphic encryption (FHE) extends this capability by allowing the evaluation of arbitrary circuits on ciphertexts, a so-called *homomorphic* evaluation. Since Gentry's foundational work in 2009 [20], numerous FHE schemes have emerged, including BGV [8], BFV [7,19], and GSW [21], all making use of lattice-based assumptions such as the Ring Learning With Errors (RLWE) hardness assumption [28,32]. These schemes differ in their native data types: BGV/BFV operate on vectors over finite fields, FHEW/TFHE [15,18] target binary data, and CKKS [14] supports approximate arithmetic over complex numbers.

The CKKS Scheme. Among modern HE schemes, CKKS stands out for enabling approximate componentwise arithmetic on complex vectors, supporting addition, multiplication, conjugation, and slot rotations — ideal for real-world privacy-preserving applications.

A key limitation in CKKS is that each rescaling step, necessary after every homomorphic multiplication of ciphertexts, consumes part of the available modulus. After a bounded number of rescaling operations, the ciphertext runs out of modulus, and further computation becomes impossible. *Bootstrapping* addresses this limitation [9,12,25] by refreshing a ciphertext to regain modulus, effectively enabling indefinite homomorphic computations.

Although CKKS is designed to support approximate arithmetic on complex vectors $\boldsymbol{z} \in \mathbb{C}^{N/2}$ for some power of two N, the scheme is based on the RLWE problem [28], and the complex vectors are therefore encoded as integer polynomials $m \in \mathbb{Z}[X]/(X^N + 1)$ via a variant of the inverse discrete Fourier transform multiplied by a large integer Δ, the *scaling factor*. Going from a complex vector $\boldsymbol{z}$ to a polynomial m is called *encoding*, while the reverse process is called *decoding*. A *CKKS encryption* of a plaintext polynomial $m \in \mathbb{Z}[X]/(X^N + 1)$ (and hence of the complex vector that it encodes) is a tuple $\mathsf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1)$ with $\mathsf{ct}_i \in \mathbb{Z}[X]/(X^N+1)$ satisfying the decryption equation $\mathsf{ct}_0 + s \cdot \mathsf{ct}_1 = m + e \bmod q$ for a small error $e \in \mathbb{Z}[X]/(X^N + 1)$, *i.e.*, with small coefficients, where the polynomial $s \in \mathbb{Z}[X]/(X^N + 1)$ is the secret key and q is a large modulus. This means that m cannot be recovered precisely from the ciphertext ct, hence the approximate nature of the scheme. The CKKS scheme supports homomorphic operations, including componentwise addition and multiplication of encrypted vectors, as well as conjugation and matrix multiplication.

Original CKKS Bootstrapping. The conventional CKKS bootstrapping procedure [12] is based on approximating the reduction-modulo-q function using a polynomial approximation of a scaled sine function. The process begins with a ciphertext $\mathsf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1)$ such that $\mathsf{ct}_0 + s \cdot \mathsf{ct}_1 = m \bmod q$. For simplicity, we treat the inherent error as part of the message $m = \sum_{i=0}^{N-1} m_i X^i$. The bootstrapping process proceeds through the following steps:

- *ModRaise*: The ciphertext ct is reinterpreted under a significantly larger modulus $Q \gg q$, resulting in the relation $\mathsf{ct}_0 + s \cdot \mathsf{ct}_1 = m' \bmod Q$, where $m' = m + qJ$ for some integer polynomial $J = \sum_{i=0}^{N-1} J_i X^i$.
- *CoeffToSlot*: A homomorphic version of the encoding process is applied to move the N coefficients of the polynomial m' into the slots of two encrypted vectors of length $N/2$ (up to a scaling factor). Consequently, the slots of these vectors contain the integers $m_i + qJ_i$ for $i \in [0, N)$.
- *EvalMod*: The terms qJ_i in the encrypted slots are removed by approximating the symmetric reduction-modulo-q function $x \mapsto [x]_q$ with a scaled sine

function, namely $[x]_q \approx q/(2\pi) \cdot \sin(2\pi x/q)$ for $|[x]_q| \ll q$. To compute $\sin y$ for $y = 2\pi x/q$, one first evaluates the degree-d Taylor approximation of $\exp(y\mathrm{I}/2^r)$, where d and r are positive integers and $\mathrm{I} = \sqrt{-1} \in \mathbb{C}$ is the imaginary unit. This approximation is then raised to the 2^r-th power via repeated squaring to approximate $\exp(y\mathrm{I})$, and the sine value is extracted as the imaginary part. When applied homomorphically, this process produces two encrypted vectors whose slots approximate the coefficients m_i.

- *SlotToCoeff*: Lastly, by applying a homomorphic version of the decoding process, one moves back the (approximated) coefficients m_i from the slots of the two encrypted vectors to the coefficients of the plaintext polynomial. One obtains a ciphertext $\mathbf{ct}' = (\mathsf{ct}'_0, \mathsf{ct}'_1)$ satisfying $\mathsf{ct}'_0 + s \cdot \mathsf{ct}'_1 = m + e \bmod q'$ for some small error e and a modulus $q' \in (q, Q)$. The ciphertext $\mathbf{ct}'$ is thus a refreshed version of $\mathbf{ct}$.

The SlotToCoeff and CoeffToSlot operations account for a significant portion of the bootstrapping runtime, as they involve homomorphic multiplication by a Vandermonde matrix or its conjugate transpose. In the method described in [12], this requires $\mathcal{O}(N)$ homomorphic operations. More generally, to bootstrap a message m lying in a subring of the form $\mathbb{Z}[X^{N/(2n)}]/(X^N + 1)$ and therefore encoding a complex vector $\boldsymbol{z} \in \mathbb{C}^n$, the method requires $\mathcal{O}(n)$ homomorphic operations. Cheon *et al.* [11] proposed a technique to reduce the complexity of the SlotToCoeff and CoeffToSlot operations from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$. This is accomplished by decomposing the Vandermonde matrix into a product of sparse matrices with a special structure, so-called *tridiagonal matrices.*

Later Work on CKKS Bootstrapping. Additional improvements have since been proposed. Han and Ki [23] introduced optimized parameter choices and evaluation strategies for sine-based approximations. Bossuat *et al.* [5] reduced evaluation complexity by employing non-sparse bootstrapping keys. Jutla and Manohar [24] improved the accuracy of EvalMod using a sine series approximation. Lee *et al.* [27] minimized error variance to enable high-precision bootstrapping, while Bae *et al.* [2] proposed the META-BTS framework, which extends bootstrapping precision beyond prior limits. Bae *et al.* [3] introduced several CKKS bootstrapping algorithms designed specifically for ciphertexts encoding binary data. In a related line of work, Bae *et al.* introduced the SI-BTS framework [4], a CKKS bootstrapping method specialized for small-integer plaintexts, particularly supporting functional bootstrapping.

Our Contribution. We introduce PaCo, a novel bootstrapping procedure for the CKKS scheme optimized for parallel efficiency. Given a small power-of-two parameter $C \in [2, N/(4h)]$, where h is the Hamming weight of the secret key, PaCo bootstraps plaintexts in $\mathbb{Z}[X^{N/C}]/(X^N + 1)$. Specifically, it transforms an encryption of $m = \sum_{i=0}^{N-1} m_i X^i$ into a bootstrapped encryption of the sparser $\tilde{m} = \sum_{i=0}^{C-1} m_{i \cdot N/C} X^{i \cdot N/C}$ using only $\mathcal{O}(\log C)$ homomorphic operations. To handle more than C coefficients, PaCo is applied up to N/C times in parallel, enabling convenient and efficient scalability with negligible overhead.

PaCo stands for *(Bootstrapping via) Partial CoeffToSlot*, as it employs a subset of the matrices used in the matrix decomposition for CoeffToSlot [11]. The method reformulates the decryption equation in terms of blind rotations and modular additions, then evaluates the resulting circuit homomorphically in the CKKS framework using polynomial ring structures and the circle group in the complex plane. For this, we develop auxiliary tools — especially for bit-reversing and alternative CKKS ring structures — that may be of independent interest to the HE community. A proof-of-concept implementation confirms that PaCo offers a modest improvement over state-of-the-art CKKS bootstrapping methods, even in the absence of parallelization. When parallelized, PaCo achieves further performance gains with negligible overhead, demonstrating excellent scalability. Unlike conventional CKKS bootstrapping methods, PaCo does not rely on polynomial approximations of periodic functions, and therefore achieves a failure probability of zero in that regard.

Related Work. Over the course of writing this paper, Cheon *et al.* [13] introduced SHIP, a CKKS bootstrapping method closely related to ours. Although our work was conducted independently, it shares several conceptual similarities with SHIP, including the use of the circle group, blind rotations, and a high degree of parallelizability. The main distinction lies in the execution of blind rotations: PaCo performs these via multiplications by secret monomials, enabled through a partial CoeffToSlot which embeds polynomial ring structures into the CKKS framework. In contrast, SHIP applies blind rotations directly to vectors in $\mathbb{C}^{N/2}$, thereby sidestepping the partial CoeffToSlot step, but at the expense of potentially poorer slot-scaling and larger bootstrapping keys. Further, SHIP implements a SlotToCoeff operation with linear complexity in the number of slots, as opposed to the logarithmic-cost decomposition used in PaCo; this appears to stem from challenges related to coefficient ordering. Whether our bit-reversing techniques could help overcome this limitation in SHIP remains an open question.

1.2 Overview of New Approach

We now present a streamlined overview of our simplified bootstrapping approach.

Alternative CKKS Ring Structures. We first explain how we use alternative ring structures for CKKS. Although the scheme is fundamentally designed to support arithmetic on vectors of complex numbers — *i.e.*, elements of the ring $\mathbb{C}^n = \mathbb{C} \times \cdots \times \mathbb{C}$ — we observe that $\mathbb{C}^n$ is just one of many isomorphic representations that we can use to structure our computations more flexibly.

Let $\mathcal{R}$ be a ring which is also a complex vector space, and let $\sigma : \mathcal{R} \to \mathbb{C}^n$ be a ring isomorphism which is also a $\mathbb{C}$-linear map. Fix a basis for the vector space $\mathcal{R}$, and denote by $[r] \in \mathbb{C}^n$ the coordinate vector of $r \in \mathcal{R}$ with respect to this basis. We represent σ via an invertible matrix $\boldsymbol{U} \in \mathbb{C}^{n\times n}$, so that $\boldsymbol{U} \cdot [r] = \sigma(r)$ for all r.

This relationship can be visualized via the following commutative diagrams, where $\boldsymbol{U}$ and $\boldsymbol{U}^{-1}$ denote the linear maps $\boldsymbol{z} \mapsto \boldsymbol{U} \cdot \boldsymbol{z}$ and $\boldsymbol{z} \mapsto \boldsymbol{U}^{-1} \cdot \boldsymbol{z}$, respectively:

$$\begin{tikzcd}
\mathcal{R} \arrow[d, "{[\cdot]}"'] \arrow[dr, "\sigma"] & \\
\mathbb{C}^n \arrow[r, "\boldsymbol{U}"'] & \mathbb{C}^n
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
& \mathcal{R} \arrow[d, "{[\cdot]}"] \\
\mathbb{C}^n \arrow[ur, "\sigma^{-1}"] \arrow[r, "\boldsymbol{U}^{-1}"'] & \mathbb{C}^n
\end{tikzcd}$$

To encrypt (or encode) elements $r \in \mathcal{R}$, we simply encrypt (or encode) the corresponding vector $\sigma(r) \in \mathbb{C}^n$. Since σ is a ring morphism, this method of encoding and encrypting is fully compatible with the homomorphic addition and multiplication of CKKS. For example, if $r_0, r_1 \in \mathcal{R}$, then homomorphically multiplying encryptions of the vectors $\sigma(r_0)$ and $\sigma(r_1)$ results in an encryption of the vector $\sigma(r_0 \cdot r_1)$. Given $r \in \mathcal{R}$, to recover an encryption of $[r]$ from an encryption of $\sigma(r)$, for example to compute products involving the coordinates of r, it suffices to homomorphically left-multiply by $\boldsymbol{U}^{-1}$. In our setting, $\boldsymbol{U}^{-1}$ can always be decomposed into tridiagonal matrices (and a permutation matrix), allowing for an efficient matrix-ciphertext multiplication.

We now detail the isomorphism σ that we will use. This isomorphism enables homomorphic evaluation of multiple polynomial multiplications in parallel, following a Single Instruction, Multiple Data (SIMD) paradigm (see, *e.g.*, [31] for a comprehensive overview of batching in homomorphic encryption).

Take any non-zero complex number $\alpha \neq 0$ and any integer $B \geqslant 1$. Evaluating at the B-th roots of α yields a ring isomorphism $\mathbb{C}[Z]/(Z^B - \alpha) \cong \mathbb{C}^B$. Similarly, we have an isomorphism $\sigma : \prod_{v=0}^{h-1} \mathbb{C}[Z]/(Z^B - \alpha_v) \to (\mathbb{C}^B)^h = \mathbb{C}^n$, where $hB = n$ and each $\alpha_v \neq 0$ is a complex number. This setup lets us perform h independent homomorphic multiplications of encrypted (or encoded) complex polynomials whose products have degree less than B, and pack the resulting coefficients into the slots of a single encrypted (or encoded) complex vector using the method described above, by homomorphically multiplying by $\boldsymbol{U}^{-1}$. Note that the latter can also be seen as a form of the CoeffToSlot operation. In the particular case we are interested in, the decomposition of $\boldsymbol{U}^{-1}$ involves tridiagonal matrices that are those appearing in the decomposition of the usual CoeffToSlot operation. However, only a subset of the full CoeffToSlot decomposition will be used. Thus, we refer to this step as a *partial CoeffToSlot operation.*

Decryption Equations. We begin with a binary secret key $s \in \mathbb{Z}[X]/(X^N + 1)$ of Hamming weight h, assumed to be of the form $s = \sum_{v=0}^{h-1} X^{u_v \cdot h + v}$, where the integers u_v, referred to as *shift indices*, are chosen uniformly at random from $[0, B)$ for $B = N/h$, with $u_0 = 0$. By writing $s = \sum_{i=0}^{N-1} s_i X^i$, when arranging the coefficients s_i into h blocks according to the reduction of i modulo h as illustrated in Figure 1, each block contains exactly one non-zero coefficient. In Section 8.1, we argue that this structure should not compromise the security of the CKKS scheme.

Suppose we have a ciphertext **ct** encrypting $m = \sum_{i=0}^{N-1} m_i X^i$. For simplicity, we focus on the procedure for obtaining a bootstrapped encryption of the sparser

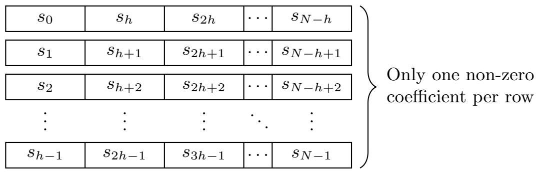
s_0 s_h s_{2h} $\cdots$ s_{N-h}
s_1 s_{h+1} s_{2h+1} $\cdots$ s_{N-h+1}
s_2 s_{h+2} s_{2h+2} $\cdots$ s_{N-h+2}
s_{h-1} s_{2h-1} s_{3h-1} $\cdots$ s_{N-1}
Only one non-zero coefficient per row

Fig. 1. Regular structure of our secret key.

polynomial $\tilde{m} = \sum_{i=0}^{B-1} m_{i \cdot N/B} Y^i$, where $Y = X^{N/B} = X^h$. The regularity of the secret key s, together with the assumption that $u_0 = 0$, allows us to transform the decryption equation $m = \mathsf{ct}_0 + s \cdot \mathsf{ct}_1 \bmod q$ into $m = \sum_{v=0}^{h-1} X^{u_v \cdot h} \cdot a_v \bmod q$, where $a_0 = \mathsf{ct}_0 + \mathsf{ct}_1$ and $a_v = X^v \cdot \mathsf{ct}_1$ for $v \in [1, h)$. Since $\tilde{m}$ is a polynomial in the variable $Y = X^h$, we can replace the $a_v = \sum_{i=0}^{N-1} a_{v,i} X^i$ by $\tilde{a}_v = \sum_{i=0}^{B-1} a_{v,ih} Y^i$, yielding a decryption equation for $\tilde{m}$:

$$\tilde{m} = \sum_{v=0}^{h-1} Y^{u_v} \cdot \tilde{a}_v = \sum_{v=0}^{h-1} a'_v \mod q, \tag{1}$$

where $a'_v = Y^{u_v} \cdot \tilde{a}_v$. (We use the tilde symbol, and later also the hat symbol, to indicate that a polynomial includes only a subset of its original coefficients, and the prime symbol to denote that a polynomial has undergone a blind rotation.)

We emphasize that in the decryption equation for $\tilde{m}$, the shift indices $u_v \in [0, B)$ are part of the secret key s and hence not openly known, while the polynomials $\tilde{a}_v$ can be extracted from the ciphertext $\mathbf{ct}$, importantly without secret knowledge. Therefore, the polynomials $\tilde{a}_v$ first undergo a *blind rotation*, that is, a multiplication (modulo $Y^B + 1$) by the secret monomials Y^{u_v}, before being summed up (modulo q). The objective now is to homomorphically reproduce the decryption circuit for $\tilde{m}$ using the tools offered by the CKKS scheme, thereby establishing a new bootstrapping equation.

Embedding in the Circle Group. To replicate the decryption equation for $\tilde{m}$ homomorphically within CKKS, we embed the additive group $\mathbb{Z}_q$ into the circle group of complex numbers. Specifically, we use the group morphism:

$$\psi : \mathbb{Z}_q \ni a \mapsto \exp\left(\frac{2\pi \mathrm{I} \cdot a}{q}\right) \in \mathbb{C}^\times.$$

Writing $\tilde{m} = \sum_{i=0}^{B-1} \tilde{m}_i Y^i$ and $a'_v = \sum_{i=0}^{B-1} a'_{v,i} Y^i$ for every $v \in [0, h)$, we observe from (1) that $\tilde{m}_i = \sum_{v=0}^{h-1} a'_{v,i} \bmod q$ for each $i \in [0, B)$. Applying the embedding to the latter equation yields $\psi(\tilde{m}_i) = \prod_{v=0}^{h-1} \psi(a'_{v,i})$. Assuming that $|\tilde{m}_i| \ll q$, the small-angle approximation gives $\tilde{m}_i \approx q/(2\pi) \cdot \mathrm{Im}(\psi(\tilde{m}_i))$, as illustrated in Figure 2, where $\mathrm{Im}(\cdot)$ denotes the imaginary part. Combining this with the above, we obtain:

$$\tilde{m}_i \approx \frac{q}{2\pi} \cdot \mathrm{Im}\left(\prod_{v=0}^{h-1} \psi(a'_{v,i})\right). \tag{2}$$

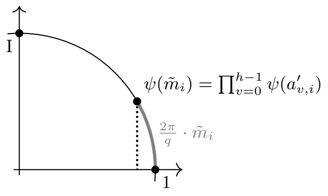
I
$\psi(\tilde{m}_i) = \prod_{v=0}^{h-1} \psi(a'_{v,i})$
$\frac{2\pi}{q} \cdot \tilde{m}_i$
1

Fig. 2. The length of the dotted line segment approximates the length of the gray arc in case $|\tilde{m}_i| \ll q$. Hence, $2\pi/q \cdot \tilde{m}_i \approx \mathrm{Im}(\psi(\tilde{m}_i))$.

Thus, our goal is to produce encrypted vectors whose slots hold the values $\psi(a'_{v,i})$. To do this, we map the coefficients of the polynomials $\tilde{a}_v = \sum_{i=0}^{B-1} \tilde{a}_{v,i} Y^i$ into the circle group using the isomorphism ψ. Specifically, for each $v \in [0, h)$, we define the complex polynomial $\tilde{b}_v = \sum_{i=0}^{B-1} \psi(\tilde{a}_{v,i}) Z^i$, which can be computed without requiring any secret information. Encrypted vectors containing the desired values $\psi(a'_{v,i})$ in their slots are then obtained by performing blind rotations followed by a partial CoeffToSlot operation, as we will detail now.

Blind Rotations. Our goal now is to homomorphically compute the shifted polynomials $b'_v = Z^{u_v} \cdot \tilde{b}_v$. (Note that the shift from $\tilde{b}_v$ to b'_v is the same as the shift from $\tilde{a}_v$ to a'_v.) Since each b'_v has degree strictly bounded by $2B$, we can do the computations modulo $Z^{2B} - \alpha_v$ for some complex $\alpha_v \neq 0$. For this, we use the isomorphism $\sigma : \prod_{v=0}^{h/4-1} \mathbb{C}[Z]/(Z^{2B} - \alpha_v) \to \mathbb{C}^{N/2}$, which allows packing $h/4$ polynomials into a single complex vector of length $N/2$. For instance, using σ, we encode the polynomials $(\tilde{b}_v)_{v \in [0,h/4)}$ into a vector $\sigma(\tilde{b}_v)_{v \in [0,h/4)}$ which we simply refer to as a *coefficient encoding* (it only contains information about coefficients of the initial ciphertext). We similarly encode the rotation monomials $(Z^{u_v})_{v \in [0,h/4)}$. Since these monomials depend on the secret key, these encodings must also be encrypted; such a ciphertext is called a *bootstrapping key*. A homomorphic multiplication of the encoded vector $\sigma(\tilde{b}_v)_{v \in [0,h/4)}$ with the encrypted vector $\sigma(Z^{u_v})_{v \in [0,h/4)}$ then yields an encryption of the vector $\sigma(Z^{u_v} \cdot \tilde{b}_v)_{v \in [0,h/4)} = \sigma(b'_v)_{v \in [0,h/4)}$. The same technique applies to the other three blocks of polynomials. We refer to this operation as a *blind rotation*: the coefficients of each polynomial $\tilde{b}_v$ are rotated by a secret shift u_v through multiplication by the monomial Z^{u_v}. This differs from the blind rotation method of [13], where secret rotations are applied directly to vectors in $\mathbb{C}^{N/2}$, requiring many associated bootstrapping keys. In contrast, our approach needs only four bootstrapping keys, one for each of the four blocks of polynomials.

Partial CoeffToSlot Operation. For every $v \in [0, h)$, write $b'_v = \sum_{i=0}^{2B-1} b'_{v,i} Z^i$. At this stage, we have encryptions of the vectors $\sigma(b'_v)_{v \in [0,h/4)}$. We apply a partial CoeffToSlot operation to these encryptions, so that the slots of the resulting encrypted vectors contain the coefficients $b'_{v,i}$ for all $v \in [0, h)$ and $i \in [0, 2B)$.

Obtaining a Bootstrapped Encryption of $\tilde{m}$. As illustrated in Figure 3, it holds that $\psi(a'_{v,i}) = b'_{v,i} + \overline{b'_{v,i+B}}$. Combining this equation with (2) yields an approximation of the coefficients of $\tilde{m}$, only involving the $b'_{v,i}$. With this, we can homomorphically compute an encrypted vector whose slots approximate the coefficients of $\tilde{m}$. To complete the bootstrapping, it suffices to apply a conventional (full) SlotToCoeff operation.

	u_v coeff.			$B - u_v$ coeff.		
All B coeff. of a'_v:	$-\tilde{a}_{v,B-u_v}$	$\cdots$	$-\tilde{a}_{v,B-1}$	$\tilde{a}_{v,0}$	$\cdots$	$\tilde{a}_{v,B-u_v-1}$
First B coeff. of b'_v:	0	$\cdots$	0	$\psi(\tilde{a}_{v,0})$	$\cdots$	$\psi(\tilde{a}_{v,B-u_v-1})$
Last B coeff. of b'_v:	$\psi(\tilde{a}_{v,B-u_v})$	$\cdots$	$\psi(\tilde{a}_{v,B-1})$	0	$\cdots$	0

Fig. 3. This figure illustrates that $\psi(a'_{v,i}) = b'_{v,i} + \overline{b'_{v,i+B}}$. If $i \in [0, u_v)$, then $b'_{v,i} = 0$ and $b'_{v,i+B} = \psi(-a'_{v,i}) = \overline{\psi(a'_{v,i})}$; if $i \in [u_v, B)$, then $b_{v,i+B} = 0$ and $b'_{v,i} = \psi(a'_{v,i})$.

Summary. At a high level, PaCo consists of the following steps:

- Reformulate the decryption equation so that it can be expressed using only blind rotations of specific polynomials, followed by additions modulo q.
- Map the coefficients of these polynomials onto the complex unit circle, and construct corresponding complex-valued polynomials.
- Carry out blind rotations on these complex polynomials.
- Apply a partial CoeffToSlot operation, so that the coefficients of the rotated polynomials are embedded in the slots of the resulting encrypted vectors.
- Use standard homomorphic operations to compute an encrypted vector in which each slot encodes a coefficient of the original plaintext polynomial.
- Finish with a conventional SlotToCoeff operation to get the bootstrapped ciphertext.

Unlike traditional CKKS bootstrapping methods, our approach does not rely on homomorphically evaluating a polynomial approximation of a periodic function, resulting in zero failure probability.

Experimental Results. Our experiments confirm that PaCo consistently outperforms the original CKKS bootstrapping procedure, even without parallelization. In fact, PaCo achieves speedups ranging from roughly 1.5× to 2.6×. The experiments also show that PaCo scales efficiently with near-ideal speedup when parallelized. For example, utilizing 32 processors in parallel yields up to an additional 2× improvement in performance. Our implementation is publicly available online via GitHub.[1]

[1] `https://github.com/se-tim/PaCo-Implementation.git`

2 Notation

We start with a collection of general notation. All intervals are assumed to contain integers only. For a real number x, we denote by $\lfloor x \rceil$ the nearest integer to x. This notation is extended coefficientwise to polynomials. All logarithms are understood to be base-2. For a mathematical statement P, we denote by $\mathbb{1}_P$ the indicator function of P, equalling 1 if P is true and 0 otherwise.

For an integer $q \geqslant 1$ and a real number x, we denote by $[x]_q$ the reduction of x modulo q. Although this notation was previously used in the introduction to refer to the symmetric reduction modulo q, we now adopt the convention that $[x]_q$ always satisfies $0 \leqslant [x]_q < q$.

We denote the imaginary unit by $\mathrm{I} = \sqrt{-1} \in \mathbb{C}$. Given a complex number $z = a + b\mathrm{I} \in \mathbb{C}$, we define its conjugate by $\overline{z} = a - b\mathrm{I}$ and its imaginary part by $\mathrm{Im}(z) = b$. This notation is extended entrywise to vectors and matrices.

For two complex vectors $\boldsymbol{z} = (z_i)_{i\in[0,n)}$ and $\boldsymbol{w} = (w_i)_{i\in[0,n)}$, their Hadamard (entrywise) product is denoted by $\boldsymbol{z} \odot \boldsymbol{w} = (z_i \cdot w_i)_{i\in[0,n)}$, and the rotation of the vector $\boldsymbol{z}$ by $j \in \mathbb{Z}$ slots is denoted by $\mathsf{Rot}_j(\boldsymbol{z}) = (z_{[i+j]_n})_{i\in[0,n)}$.

For a set S, we write $S^{m\times n}$ to denote the set of $(m \times n)$-matrices with entries in S. For a matrix $\boldsymbol{A} \in S^{m\times n}$, we denote by $\boldsymbol{A}^\mathsf{T} \in S^{n\times m}$ its transpose. Given elements $s_0, \ldots, s_{n-1} \in S$, we denote by $\mathsf{Diag}(s_0, \ldots, s_{n-1}) \in S^{n\times n}$ the diagonal matrix with the s_i on the diagonal. The identity matrix of size $n \times n$ is denoted by $\boldsymbol{I}_n$, and the zero matrix of size $n \times n$ is denoted by $\boldsymbol{0}_n$. When writing a product of matrices as $\prod_{\ell=0}^{k-1} \boldsymbol{A}_\ell$, we mean $\boldsymbol{A}_{k-1} \cdots \boldsymbol{A}_0$, following the convention that matrix multiplication, like function composition, proceeds from right to left.

Throughout, we fix an integer $N \geqslant 1$ that is a power of two, referred to as the *ring dimension* for the CKKS scheme. Additionally, $n \geqslant 1$ will denote a strict divisor of N unless otherwise specified.

3 Background on CKKS

We now recall the key components of the CKKS scheme.

3.1 Encoding and Decoding

CKKS supports the encryption of vectors $\boldsymbol{z} \in \mathbb{C}^n$ for strict divisors n of N. This is achieved by interpreting a vector $\boldsymbol{z} \in \mathbb{C}^n$ as a polynomial $m \in \mathbb{Z}[Y]/(Y^{2n} + 1)$. To accomplish this, we consider the ring isomorphism:

$$\tau_n : \frac{\mathbb{R}[Y]}{(Y^{2n} + 1)} \ni p \mapsto (p(\zeta_{n,i}))_{i\in[0,n)} \in \mathbb{C}^n,$$

where $\zeta_{n,i} = \exp(2\pi\mathrm{I} \cdot 5^i/(4n))$ are half of the primitive $(4n)$-th roots of unity.

For a fixed positive integer Δ, referred to as the *scaling factor* for the CKKS scheme, we *encode* a complex vector $\boldsymbol{z} \in \mathbb{C}^n$ via:

$$\mathsf{Ecd}(\boldsymbol{z}) = \lfloor \Delta \tau_n^{-1}(\boldsymbol{z}) \rceil \in \frac{\mathbb{Z}[Y]}{(Y^{2n} + 1)};$$

by setting $Y = X^{N/(2n)}$, we view $\mathsf{Ecd}(\boldsymbol{z})$ as an element of $\mathbb{Z}[X]/(X^N + 1)$. Consequently, for two strict divisors $n \geqslant B$ of N and two vectors $\boldsymbol{z}_1 \in \mathbb{C}^B$ and $\boldsymbol{z}_0 = (\boldsymbol{z}_1)_{i\in[0,n/B)} \in \mathbb{C}^n$, it holds that $\mathsf{Ecd}(\boldsymbol{z}_0) = \mathsf{Ecd}(\boldsymbol{z}_1)$. Therefore, it makes sense to regard $\mathbb{C}^B$ (equipped with the Hadamard product) as a subring of $\mathbb{C}^n$ via the natural embedding $\mathbb{C}^B \ni \boldsymbol{z}_1 \mapsto (\boldsymbol{z}_1)_{i\in[0,n/B)} \in \mathbb{C}^n$.

We *decode* a polynomial $m \in \mathbb{Z}[Y]/(Y^{2n} + 1)$ via:

$$\mathsf{Dcd}_n(m) = \frac{\tau_n(m)}{\Delta} \in \mathbb{C}^n.$$

For two complex vectors $\boldsymbol{z}_0, \boldsymbol{z}_1 \in \mathbb{C}^n$, it then holds that $\mathsf{Ecd}(\boldsymbol{z}_0) + \mathsf{Ecd}(\boldsymbol{z}_1) \approx \mathsf{Ecd}(\boldsymbol{z}_0 + \boldsymbol{z}_1)$. Defining the operation $m_0 \times m_1 = \lfloor m_0 m_1/\Delta \rceil \in \mathbb{Z}[X]/(X^N + 1)$ for $m_0, m_1 \in \mathbb{Z}[X]/(X^N + 1)$, we further have that $\mathsf{Ecd}(\boldsymbol{z}_0) \times \mathsf{Ecd}(\boldsymbol{z}_1) \approx \mathsf{Ecd}(\boldsymbol{z}_0 \odot \boldsymbol{z}_1)$.

3.2 Ciphertexts

A *CKKS ciphertext* encrypting a polynomial $m \in \mathbb{Z}[X]/(X^N + 1)$ under a modulus q is a tuple $\mathbf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1)$ where the $\mathsf{ct}_i \in \mathbb{Z}[X]/(X^N + 1)$ satisfy the decryption equation $\mathsf{ct}_0 + s \cdot \mathsf{ct}_1 = m + e \bmod q$ for a small error $e \in \mathbb{Z}[X]/(X^N + 1)$ and $s \in \mathbb{Z}[X]/(X^N + 1)$ the secret key. We denote this encryption by $\mathbf{ct} = \mathsf{Enc}(m)$, while keeping in mind that decryption does not recover m exactly. If $m \approx \mathsf{Ecd}(\boldsymbol{z})$ for a complex vector $\boldsymbol{z} \in \mathbb{C}^n$, we also write $\mathbf{ct} = \mathsf{Enc}^*(\boldsymbol{z})$; we say that $\mathbf{ct}$ is both an *encryption* of the polynomial m and of the vector $\boldsymbol{z}$.

Typically, the secret key $s = \sum_{i=0}^{N-1} s_i X^i$ has coefficients s_i chosen to be binary ($s_i \in [0, 2)$) or ternary ($s_i \in [-1, 1]$). Its Hamming weight $h = \#\{i \in [0, N) : s_i \neq 0\}$ can be chosen somewhat small, *e.g.*, $h = 64$.

Within the CKKS scheme, ciphertexts are usually considered under a modulus belonging to a fixed sequence of moduli $q_0, \ldots, q_L$. Each modulus in this sequence satisfies $q_\ell = q \cdot \Delta^\ell$, where q is referred to as the *base modulus*, and Δ is the scaling factor introduced earlier. A ciphertext under the modulus q_ℓ is said to be at *level* equal to ℓ. Homomorphic operations which involve multiplications typically reduce ciphertext levels.

3.3 Elementary Operations

We briefly review the homomorphic operations supported by CKKS that are relevant to our work. We do not recall how these operations are performed, but instead refer the reader to [12,14]. Consider ciphertexts $\mathbf{ct} = \mathsf{Enc}^*(\boldsymbol{z})$ and $\mathbf{ct}_i = \mathsf{Enc}^*(\boldsymbol{z}_i)$ for $i \in [0, 2)$, with $\boldsymbol{z}, \boldsymbol{z}_i \in \mathbb{C}^n$. Additionally, let $\boldsymbol{w} \in \mathbb{C}^n$ be a plaintext vector and $\lambda \in \mathbb{C}$ a scalar. Under the assumption that all necessary keys are available, the operations listed in Table 1 can be performed homomorphically.

We note that in Table 1, all homomorphic multiplications already include the rescaling step, ensuring that the resulting encrypted vectors do not have an "undesired" factor Δ. In general, the symbol $\times$ denotes a homomorphic multiplication that implicitly includes a rescaling step.

Table 1. Elementary homomorphic operations in CKKS.

Operation	**Notation**	**Encrypted vector**	**Consumed levels**
Rescaling	$\mathsf{RS}(\mathbf{ct})$	$\boldsymbol{z}/\Delta$	1
Addition	$\mathbf{ct}_0 + \mathbf{ct}_1$	$\boldsymbol{z}_0 + \boldsymbol{z}_1$	0
Subtraction	$\mathbf{ct}_0 - \mathbf{ct}_1$	$\boldsymbol{z}_0 - \boldsymbol{z}_1$	0
Plaintext-ciphertext mult.	$\mathsf{Ecd}(\boldsymbol{w}) \times \mathbf{ct}$	$\boldsymbol{w} \odot \boldsymbol{z}$	1
Scalar-ciphertext mult.	$\mathsf{Ecd}(\lambda) \times \mathbf{ct}$	$\lambda \cdot \boldsymbol{z}$	1
Ciphertext-ciphertext mult.	$\mathbf{ct}_0 \times \mathbf{ct}_1$	$\boldsymbol{z}_0 \odot \boldsymbol{z}_1$	1
Rotation by $j \in \mathbb{Z}$ slots	$\mathsf{Rot}_j(\mathbf{ct})$	$\mathsf{Rot}_j(\boldsymbol{z})$	0
Conjugation	$\mathsf{Conj}(\mathbf{ct})$	$\overline{\boldsymbol{z}}$	0

3.4 Trace and Product Operations

We consider two strict divisors $n \geqslant B$ of N. For a vector $\boldsymbol{z} = (z_i)_{i \in [0,n)} \in \mathbb{C}^n$, we introduce the *trace operation* (referred to as the *partial sum* in [12]) and the *product operation*:

$$\mathsf{Tr}_{n \to B}(\boldsymbol{z}) = \left(\sum_{j=0}^{n/B-1} z_{jB+i} \right)_{i \in [0,B)} \in \mathbb{C}^B,$$

$$\mathsf{Pr}_{n \to B}(\boldsymbol{z}) = \left(\prod_{j=0}^{n/B-1} z_{jB+i} \right)_{i \in [0,B)} \in \mathbb{C}^B.$$

As discussed in [10], these operations are extended versions of the trace and norm operations of the number field $\mathbb{Q}[X^{N/(2n)}]/(X^N+1)$ over $\mathbb{Q}[X^{N/(2B)}]/(X^N+1)$. Figure 4 visually demonstrates how these operations are performed by horizontally summing or multiplying vector slots.

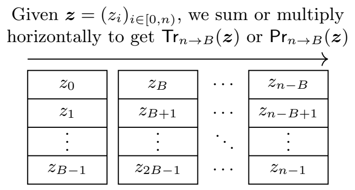
Given $\boldsymbol{z} = (z_i)_{i \in [0,n)}$, we sum or multiply horizontally to get $\mathsf{Tr}_{n \to B}(\boldsymbol{z})$ or $\mathsf{Pr}_{n \to B}(\boldsymbol{z})$
z_0
z_1
z_{B-1}
z_B
z_{B+1}
z_{2B-1}
z_{n-B}
z_{n-B+1}
z_{n-1}

Fig. 4. Illustration of how the trace and product are computed.

We can also express them using compositions of rotations and elementwise operations. Let id denote the identity map, and regard $\mathbb{C}^B$ (equipped with the Hadamard product) as a subring of $\mathbb{C}^n$ via the previously mentioned natural embedding. Then:

$$\mathsf{Tr}_{n \to B} = (\mathsf{id} + \mathsf{Rot}_B) \circ \cdots \circ (\mathsf{id} + \mathsf{Rot}_{n/4}) \circ (\mathsf{id} + \mathsf{Rot}_{n/2}),$$
$$\mathsf{Pr}_{n \to B} = (\mathsf{id} \odot \mathsf{Rot}_B) \circ \cdots \circ (\mathsf{id} \odot \mathsf{Rot}_{n/4}) \circ (\mathsf{id} \odot \mathsf{Rot}_{n/2}).$$

This decomposition enables efficient homomorphic evaluation of the trace and the product operations for a ciphertext $\mathsf{ct} = \mathsf{Enc}^*(\boldsymbol{z})$ with $\boldsymbol{z} \in \mathbb{C}^n$, see the first two operations in Table 2.

3.5 Matrix Multiplication

Consider a matrix $\boldsymbol{A} = (A_{i,j})_{i,j \in [0,n)} \in \mathbb{C}^{n \times n}$. We define for $j \in [0, n)$ the *j-th diagonal* of $\boldsymbol{A}$ as $\mathsf{Diag}_j(\boldsymbol{A}) = (A_{i,[i+j]_n})_{i \in [0,n)} \in \mathbb{C}^n$. Then, for every $\boldsymbol{z} \in \mathbb{C}^n$, the following holds true [22]:

$$\boldsymbol{A} \cdot \boldsymbol{z} = \sum_{j=0}^{n-1} \mathsf{Diag}_j(\boldsymbol{A}) \odot \mathsf{Rot}_j(\boldsymbol{z}).$$

This equality allows us to homomorphically compute a matrix-vector product. Specifically, given an encoding $\mathsf{Ecd}(\boldsymbol{A}) = (\mathsf{Ecd}(\mathsf{Diag}_j(\boldsymbol{A})))_{j \in [0,n)}$ of the matrix $\boldsymbol{A}$, and a ciphertext $\mathsf{ct} = \mathsf{Enc}^*(\boldsymbol{z})$ for some vector $\boldsymbol{z} \in \mathbb{C}^n$, we can homomorphically compute a ciphertext encrypting the product $\boldsymbol{A} \cdot \boldsymbol{z}$; see the final operation in Table 2.

A matrix $\boldsymbol{A} \in \mathbb{C}^{n \times n}$ is said to be *diagonally sparse* if most of its diagonals $\mathsf{Diag}_j(\boldsymbol{A})$ are zero. Matrix-ciphertext multiplications become particularly efficient in this case, as they require only a small number of rotations and plaintext-ciphertext multiplications.

Table 2. Further homomorphic operations in CKKS.

Operation	Notation	Encrypted vector	Consumed levels
Trace	$\mathsf{Tr}_{n \to B}(\mathsf{ct})$	$\mathsf{Tr}_{n \to B}(\boldsymbol{z})$	0
Product	$\mathsf{Pr}_{n \to B}(\mathsf{ct})$	$\mathsf{Pr}_{n \to B}(\boldsymbol{z})$	$\log(n/B)$
Matrix-ciphertext mult.	$\mathsf{Ecd}(\boldsymbol{A}) \times \mathsf{ct}$	$\boldsymbol{A} \cdot \boldsymbol{z}$	1

Some matrices $\boldsymbol{A}$ admit a decomposition of the form $\boldsymbol{A} = \prod_{\ell=0}^{k-1} \boldsymbol{D}_\ell$, where each $\boldsymbol{D}_\ell$ is diagonally sparse. This structure often leads to a more efficient homomorphic multiplication by evaluating $\mathsf{Ecd}(\boldsymbol{D}_{k-1}) \times \cdots \times \mathsf{Ecd}(\boldsymbol{D}_0) \times \mathsf{ct}$ (from right to left), rather than directly computing $\mathsf{Ecd}(\boldsymbol{A}) \times \mathsf{ct}$. For instance, if each $\boldsymbol{D}_\ell$ is *tridiagonal* — i.e., containing at most three non-zero diagonals — then the homomorphic multiplication by $\boldsymbol{A}$ can be carried out using at most $3k$ rotations and $3k$ plaintext-ciphertext multiplications. The downside, however, is that this approach consumes k levels instead of just one.

To reduce the number of levels used, we introduce a *grouping parameter* denoted by $g \in [1, k]$, which allows us to combine the matrices $\boldsymbol{D}_\ell$ into blocks of size g:

$$\prod_{\ell=0}^{k-1} \boldsymbol{D}_\ell = (\boldsymbol{D}_{k-1} \cdots \boldsymbol{D}_{k-1-[k-1]_g}) \cdots (\boldsymbol{D}_{2g-1} \cdots \boldsymbol{D}_g)(\boldsymbol{D}_{g-1} \cdots \boldsymbol{D}_0).$$

With this grouping, the number of levels consumed is reduced to $\lceil k/g \rceil$. This decreases as g increases, while the number of rotations and multiplications per block increases with g. Thus, the parameter g lets us balance computation time against level consumption. For simplicity, we will continue to denote the matrix-ciphertext multiplication as $\mathsf{Ecd}\left(\prod_{\ell=0}^{k-1} \boldsymbol{D}_\ell\right) \times \mathbf{ct}$, with the understanding that an appropriate choice of the grouping parameter g can be used to optimize efficiency. The value 2^g is commonly referred to as the *radix* [11,16] (specifically in the context of discrete Fourier transform matrices).

3.6 Conventional SlotToCoeff

We will start by revisiting the conventional $\mathsf{SlotToCoeff}$ operation. (We will not make use of the conventional $\mathsf{CoeffToSlot}$ operation.) Later on, we will show how to perform it more efficiently. Consider a complex vector $\boldsymbol{z} \in \mathbb{C}^n$ and the corresponding polynomial $p = \sum_{i=0}^{2n-1} p_i Y^i = \tau_n^{-1}(\boldsymbol{z}) \in \mathbb{R}[Y]/(Y^{2n}+1)$. Define the coefficient vectors $\boldsymbol{p}_0 = (p_i)_{i\in[0,n)} \in \mathbb{R}^n$ and $\boldsymbol{p}_1 = (p_{i+n})_{i\in[0,n)} \in \mathbb{R}^n$. For the Vandermonde matrix $\boldsymbol{U}_n = \left(\zeta_{n,i}^j\right)_{i,j\in[0,n)} \in \mathbb{C}^{n\times n}$, the vector $\boldsymbol{z}$ can be recovered from $\boldsymbol{p}_0$ and $\boldsymbol{p}_1$, as follows [12]:

$$\boldsymbol{z} = \boldsymbol{U}_n(\boldsymbol{p}_0 + \mathrm{I} \cdot \boldsymbol{p}_1). \tag{3}$$

Thus, given two ciphertexts $\mathbf{ct}_i = \mathsf{Enc}^*(\boldsymbol{p}_i)$ for $i \in [0,2)$, we can first homomorphically compute a ciphertext $\mathbf{ct} = \mathsf{Enc}^*(\boldsymbol{p}_0 + \mathrm{I}\cdot\boldsymbol{p}_1)$, and then apply the homomorphic counterpart of (3) to obtain a ciphertext $\mathsf{SlotToCoeff}_n(\mathbf{ct}) = \mathsf{Enc}^*(\boldsymbol{z})$.

This operation is referred to as $\mathsf{SlotToCoeff}$ because it transforms an encryption of a vector whose *slots* contain the coefficients of the polynomial p, into an encryption of a polynomial whose *coefficients* are given by those of p (multiplied by Δ).

The efficiency improvement from [11] arises from decomposing the Vandermonde matrix $\boldsymbol{U}_n$ into a product of diagonally sparse matrices. This decomposition, along with related new results, is presented in Section 4.

4 Decomposing the Vandermonde Matrix $\boldsymbol{U}_n$

We now examine the decomposition of the matrix $\boldsymbol{U}_n$, which serves two purposes: enabling the efficient implementation of the $\mathsf{SlotToCoeff}$ and $\mathsf{CoeffToSlot}$ operations via diagonally sparse matrices, and realizing the ring isomorphisms that are essential to PaCo.

To motivate this section, we briefly explain a subtlety that arises in the final step of our procedure, a $\mathsf{SlotToCoeff}$ operation based on the decomposition of the matrix $\boldsymbol{U}_n$. As we will see, this operation requires the encrypted input vector to be in *bit-reversed order*. Thus, when homomorphically applying the ring isomorphisms for PaCo, we must ensure that the encrypted output vector has its slots in a specific intermediate order — one that, after several trace and product operations, results in an encrypted vector whose slots are in the bit-reversed order required for the final SlotToCoeff operation. We refer to this intermediate order as *extended bit-reversed order*.

To achieve this, we ensure that already the encrypted vector we apply the ring isomorphisms to has its slots in extended bit-reversed order. Then, we modify the matrices corresponding to the ring isomorphisms by conjugating them with the appropriate permutation matrices. This ensures that the slots of the encrypted output vector are in the required extended bit-reversed order.

4.1 Bit-Reversing

Given a power of two B and an integer $i \in [0, B)$, the *bit-reversing* of i under B is defined as the integer $\mathsf{br}_B(i)$ obtained by reversing the $\log B$ bits of i. More precisely, if i is written as $i = \sum_{j=0}^{\log B-1} b_j 2^j$ with $b_j \in [0, 2)$, then $\mathsf{br}_B(i) = \sum_{j=0}^{\log B-1} b_{\log B-1-j} 2^j$. We will extend the bit-reversing operation to all non-negative integers, as follows. Given an integer $i \geqslant 0$, we define the *(extended) bit-reversing* of i under B as the integer $\mathsf{br}_B(i)$ obtained by reversing the $\log B$ least significant bits of i. This means that $\mathsf{br}_B(i) = \lfloor i/B \rfloor \cdot B + \mathsf{br}_B([i]_B)$, where $\mathsf{br}_B([i]_B)$ can be computed using the initial definition.

Consider now two powers of two $n \geqslant B$. Note that the map $\mathsf{br}_B : [0, n) \to [0, n)$ is a permutation. Let $\boldsymbol{\Pi}_n^{(B)} \in [0, 2)^{n \times n}$ denote the permutation matrix corresponding to this permutation — the entry (i, j) of $\boldsymbol{\Pi}_n^{(B)}$ is therefore 1 if and only if $j = \mathsf{br}_B(i)$. Left-multiplying a vector $\boldsymbol{z} = (z_i)_{i\in[0,n)} \in \mathbb{C}^n$ by the matrix $\boldsymbol{\Pi}_n^{(B)}$ produces the vector $\boldsymbol{\Pi}_n^{(B)}\boldsymbol{z} = (z_{\mathsf{br}_B(i)})_{i\in[0,n)}$, whose slots are thus given in bit-reversed order; we say that the vector $\boldsymbol{z}$ has been *bit-reversed*. For simplicity, we write $\boldsymbol{\Pi}^{(B)}$ instead of $\boldsymbol{\Pi}_n^{(B)}$ if the size of the matrix is clear from the context — typically when it is right-multiplied by a length-n vector or an $(n \times n)$-matrix. We note that (extended) bit-reversing commutes with both the trace and the product operations. In other words, for powers of two $n_0 \geqslant n_1 \geqslant B$, the following diagram commutes (in which the trace operation could also be replaced by the product operation):

$$\begin{tikzcd}
\mathbb{C}^{n_0} \arrow[r, "\boldsymbol{\Pi}^{(B)}"] \arrow[d, "\mathsf{Tr}_{n_0\to n_1}"'] & \mathbb{C}^{n_0} \arrow[d, "\mathsf{Tr}_{n_0\to n_1}"] \\
\mathbb{C}^{n_1} \arrow[r, "\boldsymbol{\Pi}^{(B)}"'] & \mathbb{C}^{n_1}
\end{tikzcd}$$

We finish with two remarks. First, observe that the matrix $\boldsymbol{\Pi}^{(B)}$ is its own inverse because bit-reversing twice restores the original number. Second, keep in mind that we always avoid directly performing homomorphic multiplication by the matrix $\boldsymbol{\Pi}^{(B)}$ because it is not diagonally sparse (despite being sparse).

4.2 A Sequence of Ring Isomorphisms

To analyze the decomposition of the Vandermonde matrix $\boldsymbol{U}_n$ as introduced in [11], we first study the underlying ring isomorphism that $\boldsymbol{U}_n$ corresponds to, namely:

$$\sigma_n : \frac{\mathbb{C}[Z]}{(Z^n - \mathrm{I})} \ni p \mapsto \big(p(\zeta_{n,i})\big)_{i \in [0,n)} \in \mathbb{C}^n.$$

We recall that $\zeta_{n,i} = \exp(2\pi \mathrm{I} \cdot 5^i/(4n))$; they are half of the primitive $(4n)$-th roots of unity, and all the n-th roots of I. By representing any $p = \sum_{i=0}^{n-1} p_i Z^i \in \mathbb{C}[Z]/(Z^n - \mathrm{I})$ via its coefficient vector $[p] = (p_i)_{i \in [0,n)} \in \mathbb{C}^n$, it follows that $\boldsymbol{U}_n \cdot [p] = \sigma_n(p)$ for all $p \in \mathbb{C}[Z]/(Z^n - \mathrm{I})$. This relationship is captured in the following commutative diagram:

$$\begin{tikzcd}
\frac{\mathbb{C}[Z]}{(Z^n - \mathrm{I})} \arrow[d, "{[\cdot]}"'] \arrow[dr, "\sigma_n"] & \\
\mathbb{C}^n \arrow[r, "\boldsymbol{U}_n"'] & \mathbb{C}^n
\end{tikzcd}$$

We now factor σ_n into a sequence of ring isomorphisms through an application of the Cooley-Tukey algorithm [16]. Observe that the two square roots of I are $\zeta_{n,0}^{n/2}$ and $\zeta_{n,1}^{n/2}$, leading to the ring isomorphism:

$$\frac{\mathbb{C}[Z]}{(Z^n - \mathrm{I})} \to \frac{\mathbb{C}[Z]}{\Big(Z^{n/2} - \zeta_{n,0}^{n/2}\Big)} \times \frac{\mathbb{C}[Z]}{\Big(Z^{n/2} - \zeta_{n,1}^{n/2}\Big)},$$

given by projection onto the two components. Each ring in the Cartesian product can similarly be decomposed into a further product of two rings. Continuing recursively, this process ultimately results in n copies of $\mathbb{C}$. Formally, for each $\ell \in [0, \log n]$, define the following ring:

$$\mathcal{R}_{n,2^\ell} = \prod_{i=0}^{n/2^\ell - 1} \frac{\mathbb{C}[Z]}{\Big(Z^{2^\ell} - \zeta_{n,\mathsf{br}_{n/2^\ell}(i)}^{2^\ell}\Big)},$$

with $\mathcal{R}_{n,n} = \mathbb{C}[Z]/(Z^n - \mathrm{I})$ and $\mathcal{R}_{n,1} = \mathbb{C}^n$. We then obtain a sequence of ring isomorphisms $\phi_{n,2^\ell} : \mathcal{R}_{n,2^{\ell+1}} \to \mathcal{R}_{n,2^\ell}$, defined by projecting the polynomial in the i-th component of $\mathcal{R}_{n,2^{\ell+1}}$ onto the $(2i)$-th and $(2i+1)$-st components of $\mathcal{R}_{n,2^\ell}$. This is the reason why bit-reversing is necessary, since the two square roots of $\zeta_{n,\mathsf{br}_{n/2^{\ell+1}}(i)}^{2^{\ell+1}}$ are $\zeta_{n,\mathsf{br}_{n/2^\ell}(2i)}^{2^\ell}$ and $\zeta_{n,\mathsf{br}_{n/2^\ell}(2i+1)}^{2^\ell}$. The maps $\phi_{n,1} \circ \cdots \phi_{n,n/4} \circ \phi_{n,n/2}$ and σ_n produce the same vectors modulo a reordering of the slots according to the (non-extended) bit-reversing under n.

4.3 From Isomorphisms to Matrices

For each $\ell \in [0, \log n]$, we endow the ring $\mathcal{R}_{n,2^\ell}$ with the canonical basis:

$$\big(Z^0, 0, \ldots, 0\big), \ldots, \big(Z^{2^\ell - 1}, 0, \ldots, 0\big), \ldots, \big(0, \ldots, 0, Z^0\big), \ldots, \big(0, \ldots, 0, Z^{2^\ell - 1}\big).$$

With this basis, each isomorphism $\phi_{n,2^\ell}$ can be represented by a matrix $\boldsymbol{D}_{n,2^\ell} \in \mathbb{C}^{n\times n}$. If we denote by $[p]$ the coefficient vector of $p \in \mathcal{R}_{n,2^\ell}$ with respect to the canonical basis, then we obtain the following commutative diagram:

$$\begin{tikzcd}
\mathcal{R}_{n,2^{\ell+1}} \arrow[r, "\phi_{n,2^\ell}"] \arrow[d, "{[\cdot]}"'] & \mathcal{R}_{n,2^\ell} \arrow[d, "{[\cdot]}"] \\
\mathbb{C}^n \arrow[r, "\boldsymbol{D}_{n,2^\ell}"'] & \mathbb{C}^n
\end{tikzcd}$$

The entire situation is summarized in the below commutative diagram:

$$\begin{tikzcd}
\mathcal{R}_{n,n} \arrow[r, "\phi_{n,n/2}"'] \arrow[d, "{[\cdot]}"] \arrow[rrrrrr, bend left, "\sigma_n"] & \mathcal{R}_{n,n/2} \arrow[r, "\phi_{n,n/4}"'] \arrow[d, "{[\cdot]}"] & \cdots \arrow[r, "\phi_{n,2}"'] & \mathcal{R}_{n,2} \arrow[r, "\phi_{n,1}"'] \arrow[d, "{[\cdot]}"] & \mathcal{R}_{n,1} \arrow[r, "\boldsymbol{\Pi}^{(n)}"'] \arrow[d, equal] & \mathcal{R}_{n,1} \arrow[d, equal] \\
\mathbb{C}^n \arrow[r, "\boldsymbol{D}_{n,n/2}"] \arrow[rrrrrr, bend right, "\boldsymbol{U}_n"'] & \mathbb{C}^n \arrow[r, "\boldsymbol{D}_{n,n/4}"] & \cdots \arrow[r, "\boldsymbol{D}_{n,2}"] & \mathbb{C}^n \arrow[r, "\boldsymbol{D}_{n,1}"] & \mathbb{C}^n \arrow[r, "\boldsymbol{\Pi}^{(n)}"] & \mathbb{C}^n
\end{tikzcd}$$

In particular, it yields the following decomposition of $\boldsymbol{U}_n$:

$$\boldsymbol{U}_n = \boldsymbol{\Pi}^{(n)} \cdot \left(\prod_{\ell=0}^{\log n - 1} \boldsymbol{D}_{n,n/2^{\ell+1}} \right). \tag{4}$$

As detailed in [11], each matrix $\boldsymbol{D}_{n,2^\ell}$ is block-diagonal, with the following $n/2^{\ell+1}$ blocks of size $2^{\ell+1} \times 2^{\ell+1}$ on its diagonal:

$$\left(\begin{array}{c|c} \boldsymbol{I}_{2^\ell} & \zeta^{2^\ell}_{n,\mathsf{br}_{n/2^{\ell+1}}(i)} \boldsymbol{I}_{2^\ell} \\ \hline \boldsymbol{I}_{2^\ell} & -\zeta^{2^\ell}_{n,\mathsf{br}_{n/2^{\ell+1}}(i)} \boldsymbol{I}_{2^\ell} \end{array} \right), \quad i \in [0, n/2^{\ell+1}). \tag{5}$$

Thus, the only non-zero diagonals $\mathsf{Diag}_j(\boldsymbol{D}_{n,2^\ell})$ occur for $j \in \{0, 2^\ell, n - 2^\ell\}$. Note that the inverse matrices $\boldsymbol{D}^{-1}_{n,2^\ell} = 1/2 \cdot \overline{\boldsymbol{D}_{n,2^\ell}}^{\mathsf{T}}$ remain tridiagonal.

4.4 Interposing Permutation Matrices

Consider two powers of two $n \geqslant B$. Define, for each $\ell \in [0, \log n)$, the matrix $\boldsymbol{E}^{(B)}_{n,2^\ell} = \boldsymbol{\Pi}^{(B)} \boldsymbol{D}_{n,2^\ell} \boldsymbol{\Pi}^{(B)}$. It satisfies $\boldsymbol{E}^{(B)}_{n,2^\ell} \boldsymbol{\Pi}^{(B)} = \boldsymbol{\Pi}^{(B)} \boldsymbol{D}_{n,2^\ell}$, leading to the following commutative diagram:

$$\begin{tikzcd}
\mathbb{C}^n \arrow[r, "\boldsymbol{D}_{n,n/2}"] \arrow[d, "\boldsymbol{\Pi}^{(B)}"] & \mathbb{C}^n \arrow[r, "\boldsymbol{D}_{n,n/4}"] \arrow[d, "\boldsymbol{\Pi}^{(B)}"] & \cdots \arrow[r, "\boldsymbol{D}_{n,2}"] & \mathbb{C}^n \arrow[r, "\boldsymbol{D}_{n,1}"] \arrow[d, "\boldsymbol{\Pi}^{(B)}"] & \mathbb{C}^n \arrow[d, "\boldsymbol{\Pi}^{(B)}"] \\
\mathbb{C}^n \arrow[r, "\boldsymbol{E}^{(B)}_{n,n/2}"'] & \mathbb{C}^n \arrow[r, "\boldsymbol{E}^{(B)}_{n,n/4}"'] & \cdots \arrow[r, "\boldsymbol{E}^{(B)}_{n,2}"'] & \mathbb{C}^n \arrow[r, "\boldsymbol{E}^{(B)}_{n,1}"'] & \mathbb{C}^n
\end{tikzcd}$$

Thus, applying a sequence of the $\boldsymbol{D}_{n,2^\ell}$ matrices followed by a bit-reversing is equivalent to first bit-reversing and then applying the sequence of corresponding matrices $\boldsymbol{E}^{(B)}_{n,2^\ell}$. The same holds true if we replace the $\boldsymbol{D}_{n,2^\ell}$ matrices and the $\boldsymbol{E}^{(B)}_{n,2^\ell}$ matrices by their inverses. Crucially for homomorphic computations, the matrices $\boldsymbol{E}^{(B)}_{n,2^\ell}$ and their inverses are tridiagonal, as is proven in Section A. For later reference, we finish by rewriting (4) in terms of the matrices $\boldsymbol{E}_{n,2^\ell} = \boldsymbol{E}^{(n)}_{n,2^\ell}$:

$$\boldsymbol{U}_n = \left(\prod_{\ell=0}^{\log n - 1} \boldsymbol{E}_{n,n/2^{\ell+1}} \right) \cdot \boldsymbol{\Pi}^{(n)}. \tag{6}$$

4.5 Decomposed SlotToCoeff

We now detail the efficient execution of the SlotToCoeff operation by making use of the decomposition of the Vandermonde matrix $\boldsymbol{U}_n$. As previously, we consider a complex vector $\boldsymbol{z} \in \mathbb{C}^n$, the corresponding polynomial $p = \sum_{i=0}^{2n-1} p_i Y^i = \tau_n^{-1}(\boldsymbol{z}) \in \mathbb{R}[Y]/(Y^{2n}+1)$, and the coefficient vectors $\boldsymbol{p}_0 = (p_i)_{i\in[0,n)} \in \mathbb{R}^n$ and $\boldsymbol{p}_1 = (p_{i+n})_{i\in[0,n)} \in \mathbb{R}^n$. By applying the equations (3) and (6), we can recover the vector $\boldsymbol{z}$ from the bit-reversed coefficient vectors $\boldsymbol{\Pi}^{(n)}\boldsymbol{p}_0$ and $\boldsymbol{\Pi}^{(n)}\boldsymbol{p}_1$, as follows:

$$\boldsymbol{z} = \left(\prod_{\ell=0}^{\log n - 1} \boldsymbol{E}_{n,n/2^{\ell+1}} \right) \cdot \left(\boldsymbol{\Pi}^{(n)}\boldsymbol{p}_0 + \mathrm{I} \cdot \boldsymbol{\Pi}^{(n)}\boldsymbol{p}_1 \right). \tag{7}$$

Therefore, as before, given two ciphertexts $\mathbf{ct}_i = \mathsf{Enc}^*(\boldsymbol{\Pi}^{(n)}\boldsymbol{p}_i)$ for $i \in [0,2)$, we can first homomorphically compute a ciphertext $\mathbf{ct} = \mathsf{Enc}^*(\boldsymbol{\Pi}^{(n)}(\boldsymbol{p}_0 + \mathrm{I}\cdot\boldsymbol{p}_1))$, and then apply the homomorphic counterpart of (7) to obtain $\mathsf{SlotToCoeff}'_n(\mathbf{ct}) = \mathsf{Enc}^*(\boldsymbol{z})$. Equivalently, given a polynomial $m = \sum_{i=0}^{2n-1} m_i Y^i \in \mathbb{Z}[Y]/(Y^{2n}+1)$ with coefficient vectors $\boldsymbol{m}_0 = (m_i)_{i\in[0,n)} \in \mathbb{Z}^n$ and $\boldsymbol{m}_1 = (m_{i+n})_{i\in[0,n)} \in \mathbb{Z}^n$, then we have:

$$\mathsf{SlotToCoeff}'_n\left(\mathsf{Enc}^*\left(\frac{1}{\Delta} \cdot \boldsymbol{\Pi}^{(n)}(\boldsymbol{m}_0 + \mathrm{I}\cdot\boldsymbol{m}_1) \right) \right) = \mathsf{Enc}(m). \tag{8}$$

Note that the SlotToCoeff′ operation can be executed more efficiently than the original SlotToCoeff operation because it involves homomorphic multiplications by diagonally sparse matrices, see Section 3.5. However, it requires the encrypted coefficient vectors to be bit-reversed.

5 Sequential Bootstrapping

We now present the full details of PaCo. This section focuses on the sequential (non-parallel) version, which serves as the foundation for the parallel implementation discussed in the upcoming Section 6.

5.1 Setup

We fix the Hamming weight of the secret key as a power of two $h \in [1, N]$. In the introduction, we have assumed $B = N/h$, which implied that the costly partial CoeffToSlot operation would need to be executed four times. To optimize this, we instead take $B = N/(4h)$. We will further choose a power of two $C \in [2, B]$, the number of coefficients to be bootstrapped in the encrypted polynomial. We can then write $k \cdot C = B$ for some positive integer k.

5.2 Decryption Equations

Secret Key. The secret key follows the same regular structure as described in the introduction and depicted in Figure 1, though we express it here in a slightly modified form. Concretely, we choose the PaCo secret key as $s = \sum_{v=0}^{4h-1} d_v \cdot X^{u_v \cdot 4h+v} \in \mathbb{Z}[X]/(X^N + 1)$, where the integers u_v, referred to as *shift indices*, are chosen uniformly at random from $[0, B)$, except $u_0 = 0$, and where the binary $d_v \in [0, 2)$ act as *selectors*. These selectors are chosen so that for every $v \in [0, h)$, precisely one among the four values d_v, d_{h+v}, d_{2h+v} and d_{3h+v} is equal to 1, while the remaining ones are 0. Equivalently, for each $v \in [0, h)$, there exists a unique $t_v \in [0, 4)$ for which $d_{t_v h+v} = 1$. To guarantee that the constant term of s is zero, we take $d_0 = 1$. Thanks to the selectors, we can collapse a sum of the form $\sum_{t=0}^{3} d_{th+v} \cdot p_{th+v}$ into a single term p_{λ_v} for every $v \in [0, h)$, where $\lambda_v = t_v h + v$. Ultimately, this will reduce the number of ciphertexts we have to deal with initially. The secret key generation is given in Algorithm 1. In Section 8.1, we argue that the structure of the secret key should not compromise the security of the CKKS scheme.

Algorithm 1 `skGen` (generation of a PaCo secret key)

Input: A power of two $h \in [1, N]$.
Output: A PaCo secret key $s \in \mathbb{Z}[X]/(X^N + 1)$ with Hamming weight h, the shift indices $(u_v)_{v\in[0,4h)}$ and the selectors $(d_v)_{v\in[0,4h)}$.

```
1: $B \leftarrow N/(4h)$
2: $u_0 \leftarrow 0$ and $u_1, \dots, u_{4h-1} \xleftarrow{\$} [0, B)$        ▷ Uniform random sampling
3: $d_0 \leftarrow 1$ and $d_1, \dots, d_{4h-1} \leftarrow 0$
4: for $v \in [1, h)$ do
5:     $t \xleftarrow{\$} [0, 4)$ and $d_{th+v} \leftarrow 1$
6: $s_0, \dots, s_{N-1} \leftarrow 0$
7: for $v \in [0, 4h)$ do
8:     $s_{v+u_v \cdot 4h} \leftarrow d_v$
9: $s \leftarrow \sum_{i=0}^{N-1} s_i X^i \in \mathbb{Z}[X]/(X^N + 1)$
10: return $s$, $(u_v)_{v\in[0,4h)}$ and $(d_v)_{v\in[0,4h)}$
```

Decryption Equation for $\boldsymbol{m}$. Starting with a ciphertext $\mathbf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1)$ encrypting $m = \sum_{i=0}^{N-1} m_i X^i \in \mathbb{Z}[X]/(X^N + 1)$ with $|m_i| \ll q$, the goal is to obtain a bootstrapped encryption of the sparser $\tilde{m} = \sum_{i=0}^{C-1} m_{i \cdot N/C} Y^i$, where $Y = X^{N/C} = X^{4hk}$. For this, we first transform the decryption equation $m = \mathsf{ct}_0 + s \cdot \mathsf{ct}_1 \bmod q$ into $m = \sum_{v=0}^{4h-1} d_v \cdot X^{u_v \cdot 4h} \cdot a_v \bmod q$, where $a_0 = \mathsf{ct}_0 + \mathsf{ct}_1$ and $a_v = X^v \cdot \mathsf{ct}_1$ for $v \in [1, 4h)$.

Decryption Equation for $\boldsymbol{\hat{m}}$. Take the polynomial $\hat{m} = \sum_{i=0}^{B-1} m_{i \cdot 4h} X^{i \cdot 4h}$. In the transformed decryption expression for m, we substitute the polynomials $a_v = \sum_{i=0}^{N-1} a_{v,i} X^i$ with $\tilde{a}_v = \sum_{i=0}^{B-1} \tilde{a}_{v,i} X^{i \cdot 4h}$ for $\tilde{a}_{v,i} = a_{v,i \cdot 4h}$, resulting in a decryption expression for $\hat{m}$, namely $\hat{m} = \sum_{v=0}^{4h-1} d_v \cdot X^{u_v \cdot 4h} \cdot \tilde{a}_v \bmod q$.

Decryption Equation for $\boldsymbol{\tilde{m}}$. To obtain a decryption equation for $\tilde{m}$, we extract from the decryption equation for $\hat{m}$ only those terms of each $X^{u_v \cdot 4h} \cdot \tilde{a}_v$ that are powers of $Y = X^{4hk}$. For this, write each $\tilde{a}_v$ as $\tilde{a}_v = \sum_{r=0}^{k-1} X^{r \cdot 4h} \cdot \tilde{a}_v^{(r)}$, where $\tilde{a}_v^{(r)} = \sum_{i=0}^{C-1} \tilde{a}_{v,ik+r} X^{i \cdot 4hk}$ is a polynomial in $Y = X^{4hk}$. Substituting this into the decryption equation for $\hat{m}$ gives $\hat{m} = \sum_{v=0}^{4h-1} \sum_{r=0}^{k-1} d_v \cdot X^{4h(u_v + r)} \cdot \tilde{a}_v^{(r)} \bmod q$. Since $Y = X^{4hk}$, only terms where $4h(u_v + r)$ is divisible by $4hk$ contribute to $\tilde{m}$, *i.e.*, when $u_v + r = 0 \bmod k$. Thus, the decryption equation for $\tilde{m}$ becomes:

$$\begin{aligned}\tilde{m} &= \sum_{v=0}^{4h-1} \sum_{r=0}^{k-1} d_v \cdot \mathbb{1}_{u_v + r = 0 \bmod k} \cdot Y^{(u_v + r)/k} \cdot \tilde{a}_v^{(r)} \mod q \qquad (9)\\ &= \sum_{v=0}^{h-1} \sum_{r=0}^{k-1} \mathbb{1}_{u_{\lambda_v} + r = 0 \bmod k} \cdot Y^{(u_{\lambda_v} + r)/k} \cdot \tilde{a}_{\lambda_v}^{(r)} \mod q\\ &= \sum_{v=0}^{h-1} Y^{\lceil u_{\lambda_v}/k \rceil} \cdot \tilde{a}_{\lambda_v}^{([-u_{\lambda_v}]_k)} \mod q.\end{aligned}$$

If we set $a'_v = Y^{\lceil u_v/k \rceil} \cdot \tilde{a}_v^{([-u_v]_k)}$ for every $v \in [0, 4h)$, we conclude that $\tilde{m} = \sum_{v=0}^{h-1} a'_{\lambda_v} \bmod q$, or equivalently, if we write $\tilde{m} = \sum_{i=0}^{C-1} \tilde{m}_i Y^i$ and $a'_v = \sum_{i=0}^{C-1} a'_{v,i} Y^i$ for $v \in [0, h)$, then $\tilde{m}_i = \sum_{v=0}^{h-1} a'_{\lambda_v, i} \bmod q$.

5.3 Embedding in the Circle Group

Given the embedding $\psi : \mathbb{Z}_q \ni a \mapsto \exp(2\pi \mathrm{I} \cdot a/q) \in \mathbb{C}^\times$ and applying the small-angle approximation, we get an approximation for the coefficients of $\tilde{m}$:

$$\tilde{m}_i \approx \frac{q}{2\pi} \cdot \mathrm{Im}\left(\prod_{v=0}^{h-1} \psi(a'_{\lambda_v, i}) \right).$$

Consider the complex polynomials $\tilde{b}_v^{(r)} = \sum_{i=0}^{C-1} \psi\big(\tilde{a}_{v,i}^{(r)}\big) Z^i$ and $b'_v = Z^{\lceil u_v/k \rceil} \cdot \tilde{b}_v^{([-u_v]_k)}$ for each $v \in [0, 4h)$ and $r \in [0, k)$. Drawing on reasoning similar to that attached to Figure 3, and writing $b'_v = \sum_{i=0}^{2C-1} b'_{v,i} Z^i$, we get the identity $\psi(a'_{v,i}) = b'_{v,i} + \overline{b'_{v,i+C}}$. This leads us to the approximation:

$$\tilde{m}_i \approx \frac{q}{2\pi} \cdot \mathrm{Im}\left(\prod_{v=0}^{h-1} \left(b'_{\lambda_v,i} + \overline{b'_{\lambda_v,i+C}} \right) \right). \tag{10}$$

It is worth emphasizing, again, that the $\tilde{b}_v^{(r)}$ polynomials can be computed without access to secret information, whereas the b'_v polynomials will need to be computed homomorphically using bootstrapping keys. More details will be provided below.

5.4 Polynomial Packing

We now describe how the polynomials involved in PaCo will be packed.

Packing the Polynomials $\tilde{\boldsymbol{b}}_{\boldsymbol{v}}^{(\boldsymbol{r})}$. Consider the $4hk$ complex polynomials $\tilde{b}_v^{(r)}$ for $v \in [0, 4h)$ and $r \in [0, k)$, each with degree strictly bounded by C. We can pack hk polynomials $\tilde{b}_v^{(r)}$ into a single element of the ring $\mathcal{R}_{n,2C}^k = \mathcal{R}_{n,2C} \times \cdots \times \mathcal{R}_{n,2C}$, where $n = 2hC = N/(2k)$. Because there are $4hk$ polynomials in total, this packing results in four elements of $\mathcal{R}_{n,2C}^k$. Formally, for each $t \in [0, 4)$ and $r \in [0, k)$, we construct the element:

$$\left(\tilde{b}_{th+v}^{(r)} \right)_{v \in [0,h)} \in \mathcal{R}_{n,2C}.$$

To these elements, we apply the ring isomorphism $\phi_{n,1} \circ \cdots \circ \phi_{n,C/2} \circ \phi_{n,C}$ to obtain $4k$ vectors $\boldsymbol{\beta}_t^{(r)} \in \mathbb{C}^n$. Equivalently:

$$\boldsymbol{\beta}_t^{(r)} = \left(\prod_{\ell=0}^{\log C} \boldsymbol{D}_{n,C/2^\ell} \right) \cdot \left[\left(\tilde{b}_{th+v}^{(r)} \right)_{v \in [0,h)} \right] \in \mathbb{C}^n.$$

For each $t \in [0, 4)$, we combine the k vectors $\boldsymbol{\beta}_t^{(r)}$ for $r \in [0, k)$ into one vector, yielding $\boldsymbol{\beta}_t = \left(\boldsymbol{\beta}_t^{(r)} \right)_{r \in [0,k)} \in \mathbb{C}^{N/2}$.

Packing the Secret Key Monomials. Following (9), we will similarly pack the $4hk$ monomials $d_v \cdot \mathbb{1}_{u_v + r = 0 \bmod k} \cdot Z^{(u_v + r)/k}$ with $v \in [0, 4h)$ and $r \in [0, k)$. Thus, during key generation, the following vector is formed for every $t \in [0, 4)$ and every $r \in [0, k)$:

$$\boldsymbol{\sigma}_t^{(r)} = \left(\prod_{\ell=0}^{\log C} \boldsymbol{D}_{n,C/2^\ell} \right) \cdot \left[\left(d_{th+v} \cdot \mathbb{1}_{u_{th+v} + r = 0 \bmod k} \cdot Z^{(u_{th+v} + r)/k} \right)_{v \in [0,h)} \right] \in \mathbb{C}^n.$$

Then, for each $t \in [0, 4)$, the k vectors $\boldsymbol{\sigma}_t^{(r)}$ are packed into a single vector, yielding the vector $\boldsymbol{\sigma}_t = \left(\boldsymbol{\sigma}_t^{(r)} \right)_{r \in [0,k)} \in \mathbb{C}^{N/2}$.

Packing Relation. The vectors $\boldsymbol{\beta}_t$ and $\boldsymbol{\sigma}_t$ satisfy the following identity, which we refer to as the *packing relation*:

$$\left(\prod_{\ell=0}^{\log C}\left(\boldsymbol{E}_{n,2^\ell}^{(C/2)}\right)^{-1}\right)\cdot \mathsf{Tr}_{N/2\to n}\left(\sum_{t=0}^{3}\left(\boldsymbol{\Pi}^{(C/2)}\boldsymbol{\beta}_t\right)\odot\left(\boldsymbol{\Pi}^{(C/2)}\boldsymbol{\sigma}_t\right)\right)$$
$$=\left(\left(b'_{\lambda_v,\mathsf{br}_{C/2}(i)}\right)_{i\in[0,2C)}\right)_{v\in[0,h)}. \qquad (11)$$

This relation is derived algebraically in Section B; however, we also provide a high-level explanation here for intuition. The key idea is that the permutation matrices $\boldsymbol{\Pi}^{(C/2)}$ can be factored out and merged with the matrices $\left(\boldsymbol{E}_{n,2^\ell}^{(C/2)}\right)^{-1}$. This composition corresponds to applying the inverse ring isomorphism $(\phi_{n,1}\circ\cdots\circ\phi_{n,C/2}\circ\phi_{n,C})^{-1}$, followed by bit-reversing the slots. This accounts for the appearance of polynomial coefficients directly in the vector slots, but in bit-reversed order. Next, the trace operation can be replaced by a simple summation over $r\in[0,k)$, provided that the vectors $\boldsymbol{\beta}_t$ and $\boldsymbol{\sigma}_t$ are replaced with their respective components $\boldsymbol{\beta}_t^{(r)}$ and $\boldsymbol{\sigma}_t^{(r)}$. The final result is a concatenation of h vectors, indexed by $v\in[0,h)$, each constructed as a double sum over t and r. Notably, for every v, only a single term contributes to the double sum, namely the one where $t=t_v$ and $r=[-u_{\lambda_v}]_k$. This term is the polynomial b'_{λ_v}.

The packing relation allows us to derive (an encryption of) the coefficients of b'_{λ_v} from (encodings of) $\boldsymbol{\Pi}^{(C/2)}\boldsymbol{\beta}_t$ and (encryptions of) $\boldsymbol{\Pi}^{(C/2)}\boldsymbol{\sigma}_t$. These coefficients, in bit-reversed order, are used — via (10) — to compute encryptions of the coefficients of $\tilde{m}$, also in bit-reversed order. This enables the final application of the decomposed $\mathsf{SlotToCoeff}'$ operation to get a bootstrapped encryption of $\tilde{m}$.

5.5 Bootstrapping Keys and Coefficient Encodings

The bootstrapping procedure starts by homomorphically replicating the circuit defined by the packing relation. This requires generating *bootstrapping keys* during key generation, defined as $\mathsf{bsk}_t=\mathsf{Enc}^*(\boldsymbol{\Pi}^{(C/2)}\boldsymbol{\sigma}_t)$ for $t\in[0,4)$. The generation process is outlined in Algorithm 2.

Furthermore, from the ciphertext that is to be bootstrapped, four *coefficient encodings* are generated, which are defined as $\mathsf{cEcd}_t=\mathsf{Ecd}(\boldsymbol{\Pi}^{(C/2)}\boldsymbol{\beta}_t)$ for $t\in[0,4)$. Algorithm 3 summarizes the process of generating them.

5.6 PaCo Algorithm

With the preparatory steps in place, we now turn to the core of PaCo: the bootstrapping algorithm itself. The method is formally described in Algorithm 4, and an explanation of the steps now follows.

Forming the Coefficient Encodings. First, the four coefficient encodings cEcd_t with $t\in[0,4)$ are formed. This is described in line 2 of Algorithm 4.

Algorithm 2 `bskGen` (generation of bootstrapping keys)

```
Input: Powers of two $h \in [1, N]$ and $C \in [2, N/(4h)]$, the shift indices $(u_v)_{v\in[0,4h)}$ and
    the selectors $(d_v)_{v\in[0,4h)}$.
Output: Bootstrapping keys $(\mathsf{bsk}_t)_{t\in[0,4)}$.
 1: $k \leftarrow N/(4hC)$ and $n \leftarrow 2hC$
 2: for $t \in [0,4)$ do
 3:     for $r \in [0,k)$ do
 4:         $z_0, \dots, z_{n-1} \leftarrow 0$
 5:         for $v \in [0,h)$ do
 6:             $i \leftarrow u_{th+v} + r$
 7:             if $i = 0 \bmod k$ then
 8:                 $z_{v\cdot 2C+i/k} \leftarrow d_{th+v}$
 9:         $\boldsymbol{z} \leftarrow (z_i)_{i\in[0,n)} \in \mathbb{C}^n$
10:         $\boldsymbol{\sigma}_t^{(r)} \leftarrow \left(\prod_{\ell=0}^{\log C} \boldsymbol{D}_{n,C/2^\ell}\right) \cdot \boldsymbol{z} \in \mathbb{C}^n$
11:     $\boldsymbol{\sigma}_t \leftarrow \left(\boldsymbol{\sigma}_t^{(r)}\right)_{r\in[0,k)} \in \mathbb{C}^{N/2}$        ▷ Vector concatenation
12:     $\mathsf{bsk}_t \leftarrow \mathsf{Enc}^*\left(\boldsymbol{\Pi}^{(C/2)}\boldsymbol{\sigma}_t\right)$
13: return $(\mathsf{bsk}_t)_{t\in[0,4)}$
```

Homomorphic Packing Relation. Next, we proceed by following the steps prescribed by the packing relation, as outlined in the lines 3 to 7 in Algorithm 4. This results in an encryption of the vector $\left(\left(b'_{\lambda_v,\mathsf{br}_{C/2}(i)}\right)_{i\in[0,2C)}\right)_{v\in[0,h)}$.

Extracting the Coefficients of $\tilde{\boldsymbol{m}}$. The next step involves extracting the coefficients of $\tilde{m}$, ensuring that the resulting ciphertext is properly prepared for the final $\mathsf{SlotToCoeff}'$ operation. This is achieved using the approximation provided in (10). In Algorithm 4, this corresponds to the lines 8 to 15. We relegate the technical details to Section C. The outcome is a ciphertext encrypting the vector $1/\Delta \cdot \boldsymbol{\Pi}^{(C/2)}(\tilde{\boldsymbol{m}}_0 + \mathrm{I}\cdot\tilde{\boldsymbol{m}}_1)$ for the two coefficient vectors $\tilde{\boldsymbol{m}}_0 = (\tilde{m}_i)_{i\in[0,C/2)}$ and $\tilde{\boldsymbol{m}}_1 = (\tilde{m}_{i+C/2})_{i\in[0,C/2)}$. We note that in CKKS, the combination of a rotation and a conjugation (line 8) can be executed as a single homomorphic operation.

Applying a Conventional $\mathsf{SlotToCoeff}'$. The final step is described in 16, and involves applying a conventional (decomposed) $\mathsf{SlotToCoeff}'$ to the ciphertext. According to (8), the resulting ciphertext is an encryption of the polynomial $\tilde{m} = \sum_{i=0}^{C-1} \tilde{m}_i Y^i$, as required. With this, we conclude the description of the sequential bootstrapping procedure.

5.7 Multiplicative Depth

Table 3 summarizes the number of levels consumed in Algorithm 4. We use $g_0 \in [1, \log(2C)]$ as the grouping parameter for the partial $\mathsf{CoeffToSlot}$ operation (line 7), and $g_1 \in [1, \log(C/2)]$ for the full $\mathsf{SlotToCoeff}'$ operation (line 16). These correspond to radices of 2^{g_0} and 2^{g_1}, respectively. The multiplicative depth of

Algorithm 3 `getCoeffEnc` (generation of coefficient encodings)

```
Input: A ciphertext $\mathbf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1)$, the Hamming weight $h$ of the secret key, and a
    power of two $C \in [2, N/(4h)]$.
Output: Coefficient encodings $(\mathsf{cEcd}_t)_{t\in[0,4)}$.
 1: $k \leftarrow N/(4hC)$ and $n \leftarrow 2hC$
 2: for $v \in [0, 4h)$ do
 3:     for $r \in [0, k)$ do
 4:         for $i \in [0, C)$ do
 5:             if $v = 0$ then
 6:                 $\tilde{b}^{(r)}_{v,i} \leftarrow \psi(\mathsf{ct}_{0,4h(ik+r)} + \mathsf{ct}_{1,4h(ik+r)})$    ▷ $\mathsf{ct}_j = \sum_{i=0}^{N-1} \mathsf{ct}_{j,i} X^i$
 7:             else if $i = 0 = r$ then
 8:                 $\tilde{b}^{(r)}_{v,i} \leftarrow \psi(-\mathsf{ct}_{1,N-v})$
 9:             else
10:                 $\tilde{b}^{(r)}_{v,i} \leftarrow \psi(\mathsf{ct}_{1,4h(ik+r)-v})$
11:             $\tilde{b}^{(r)}_{v,i+C} \leftarrow 0$
12:         $\big[\tilde{b}^{(r)}_v\big] \leftarrow \big(\tilde{b}^{(r)}_{v,i}\big)_{i\in[0,2C)} \in \mathbb{C}^n$
13: for $t \in [0, 4)$ do
14:     for $r \in [0, k)$ do
15:         $\boldsymbol{z} \leftarrow \big(\big[\tilde{b}^{(r)}_{th+v}\big]\big)_{v\in[0,h)} \in \mathbb{C}^n$
16:         $\boldsymbol{\beta}^{(r)}_t \leftarrow \big(\prod_{\ell=0}^{\log C} \boldsymbol{D}_{n,C/2^\ell}\big) \cdot \boldsymbol{z} \in \mathbb{C}^n$
17:     $\boldsymbol{\beta}_t \leftarrow \big(\boldsymbol{\beta}^{(r)}_t\big)_{r\in[0,k)} \in \mathbb{C}^{N/2}$
18:     $\mathsf{cEcd}_t \leftarrow \mathsf{Ecd}(\boldsymbol{\Pi}^{(C/2)} \boldsymbol{\beta}_t)$
19: return $(\mathsf{cEcd}_t)_{t\in[0,4)}$
```

the entire bootstrapping procedure is therefore given by $L = \lceil \log(2C)/g_0 \rceil + \lceil \log(C/2)/g_1 \rceil + \log h + 3$.

Table 3. Level consumption for relevant lines in Algorithm 4.

Line	Operation	Consumed levels
4	Plaintext-ciphertext mult.	1
7	Partial CoeffToSlot	$\lceil \log(2C)/g_0 \rceil$
10	Plaintext-ciphertext mult.	1
11	Product operation	$\log h$
14	Plaintext-ciphertext mult.	1
16	Conventional SlotToCoeff′	$\lceil \log(C/2)/g_1 \rceil$

6 Parallel Bootstrapping

We now show how to parallelize the sequential bootstrapping procedure, allowing us to efficiently bootstrap an encrypted polynomial with up to N coefficients.

Algorithm 4 `seqPaCo` (sequential bootstrapping procedure)

```
Input: An encryption $\mathbf{ct}$ of $m = \sum_{i=0}^{N-1} m_i X^i$, the Hamming weight $h$ of the PaCo
    secret key, a power of two $C \in [2, N/(4h)]$, and corresponding bootstrapping keys
    $(\mathsf{bsk}_t)_{t\in[0,4)}$.
Output: Bootstrapped ciphertext $\tilde{\mathbf{ct}}$ encrypting $\tilde{m} = \sum_{i=0}^{C-1} m_{i\cdot N/C} X^{i\cdot N/C}$.
 1: $n \leftarrow 2hC$
 2: $(\mathsf{cEcd}_t)_{t\in[0,4)} \leftarrow \mathtt{getCoeffEnc}(\mathbf{ct}, h, C)$          ▷ Apply Algorithm 3
 3: for $t \in [0,4)$ do
 4:     $\tilde{\mathbf{ct}}_t \leftarrow \mathsf{cEcd}_t \times \mathsf{bsk}_t$
 5: $\tilde{\mathbf{ct}} \leftarrow \sum_{t=0}^{3} \tilde{\mathbf{ct}}_t$
 6: $\tilde{\mathbf{ct}} \leftarrow \mathsf{Tr}_{N/2\rightarrow n}(\tilde{\mathbf{ct}})$
 7: $\tilde{\mathbf{ct}} \leftarrow \mathsf{Ecd}\big(\prod_{\ell=0}^{\log C} \big(\boldsymbol{E}_{n,2^\ell}^{(C/2)}\big)^{-1}\big) \times \tilde{\mathbf{ct}}$          ▷ Partial CoeffToSlot
 8: $\tilde{\mathbf{ct}}' \leftarrow \mathsf{Conj}(\mathsf{Rot}_C(\tilde{\mathbf{ct}}))$          ▷ Can be done in a single homomorphic operation
 9: $\tilde{\mathbf{ct}} \leftarrow \tilde{\mathbf{ct}} + \tilde{\mathbf{ct}}'$
10: $\tilde{\mathbf{ct}} \leftarrow \mathsf{Ecd}(\boldsymbol{\mu}) \times \tilde{\mathbf{ct}}$          ▷ $\boldsymbol{\mu} \leftarrow (\mathbb{1}_{[i]_{2C}<C})_{i\in[0,n)} \in \mathbb{C}^n$
11: $\tilde{\mathbf{ct}} \leftarrow \mathsf{Pr}_{n\rightarrow 2C}(\tilde{\mathbf{ct}})$
12: $\tilde{\mathbf{ct}} \leftarrow \mathsf{Tr}_{2C\rightarrow C}(\tilde{\mathbf{ct}})$
13: $\tilde{\mathbf{ct}} \leftarrow \tilde{\mathbf{ct}} - \mathsf{Conj}(\tilde{\mathbf{ct}})$
14: $\tilde{\mathbf{ct}} \leftarrow \mathsf{Ecd}(\boldsymbol{\eta}) \times \tilde{\mathbf{ct}}$          ▷ $\boldsymbol{\eta} \leftarrow -q\mathrm{I}/(4\Delta\pi) \cdot (\mathbb{1}_{i<C/2} + \mathrm{I}\cdot \mathbb{1}_{i\geqslant C/2})_{i\in[0,C)} \in \mathbb{C}^C$
15: $\tilde{\mathbf{ct}} \leftarrow \mathsf{Tr}_{C\rightarrow C/2}(\tilde{\mathbf{ct}})$
16: $\tilde{\mathbf{ct}} \leftarrow \mathsf{SlotToCoeff}'_{C/2}(\tilde{\mathbf{ct}})$          ▷ Conventional SlotToCoeff′
17: return $\tilde{\mathbf{ct}}$
```

6.1 Setup

As in the sequential case, we fix the Hamming weight of the secret key to a power of two $h \in [1, N]$, and we take $B = N/(4h)$. The secret key s is structured as described in Section 5, and generated via Algorithm 1. The goal is to fully bootstrap an encrypted polynomial lying in $\mathbb{Z}[T]/(T^D + 1)$, where $D \in [1, N]$ is a power of two and $T = X^{N/D}$. We assume a power-of-two number of processors $\kappa \in [\lceil D/B \rceil, D/2]$ is available. Each processor will then be responsible for bootstrapping $C = D/\kappa$ coefficients. The bootstrapping keys $(\mathsf{bsk}_t)_{t\in[0,4)}$ are generated using Algorithm 2.

6.2 Procedure Description

Consider an encryption $\mathbf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1)$ of the polynomial $m = \sum_{i=0}^{D-1} m_i T^i \in \mathbb{Z}[T]/(T^D+1)$. We write $m = \sum_{r=0}^{\kappa-1} T^r \cdot \tilde{m}^{(r)}$, where each $\tilde{m}^{(r)} = \sum_{i=0}^{C-1} m_{i\kappa+r} T^{i\cdot\kappa}$ is a polynomial in the variable $Y = T^\kappa = X^{N/C}$. For each $r \in [0, \kappa)$, we define the shifted ciphertext $\mathbf{ct}^{(r)} = T^{-r} \cdot \mathbf{ct} = (T^{-r}\mathsf{ct}_0, T^{-r}\mathsf{ct}_1)$, which encrypts $T^{-r} \cdot m$. Applying the sequential bootstrapping procedure from Algorithm 4 to each $\mathbf{ct}^{(r)}$ yields a bootstrapped ciphertext $\tilde{\mathbf{ct}}^{(r)}$ encrypting $\tilde{m}^{(r)}$. These κ ciphertexts can be recombined to recover a bootstrapped encryption of m by computing $\mathbf{ct} = \sum_{r=0}^{\kappa-1} T^r \cdot \tilde{\mathbf{ct}}^{(r)}$, since $m = \sum_{r=0}^{\kappa-1} T^r \cdot \tilde{m}^{(r)}$. Importantly, this recombination step involves only additions and monomial multiplications, which do not consume any levels in the CKKS scheme and are computationally cheap. The entire parallel bootstrapping method is summarized in Algorithm 5.

Algorithm 5 `parallelPaCo` (parallel bootstrapping procedure)

```
Input: A power of two $D \in [1, N]$, an encryption $\mathsf{ct}$ of $m = \sum_{i=0}^{D-1} m_i T^i$, the Hamming
    weight $h$ of the secret key, a power-of-two number of processors $\kappa \in [\lceil 4hD/N \rceil, D/2]$,
    and bootstrapping keys $(\mathsf{bsk}_t)_{t \in [0,4)}$ for $C = D/\kappa$ coefficients.
Output: Bootstrapped ciphertext $\mathsf{ct}_{\mathsf{boot}}$ encrypting $m$.
1: $C \leftarrow D/\kappa$
2: for $r \in [0, \kappa)$ in parallel do
3:     $\tilde{\mathsf{ct}}^{(r)} \leftarrow T^{-r} \cdot \mathsf{ct}$
4:     $\tilde{\mathsf{ct}}^{(r)} \leftarrow \mathtt{seqPaCo}(\tilde{\mathsf{ct}}^{(r)}, h, C, (\mathsf{bsk}_t)_{t \in [0,4)})$
5:     $\tilde{\mathsf{ct}}^{(r)} \leftarrow T^{r} \cdot \tilde{\mathsf{ct}}^{(r)}$
6: $\mathsf{ct}_{\mathsf{boot}} \leftarrow \sum_{r=0}^{\kappa-1} \tilde{\mathsf{ct}}^{(r)}$
7: return $\mathsf{ct}_{\mathsf{boot}}$
```

7 Practical Results

7.1 Implementation Details

We implemented a proof of concept using SageMath [17] relying on the NTL library [30], which is based on multiple-precision arithmetic. The goal of this implementation is not speed optimization, but rather to validate the correctness and practicality of our approach. For a fair comparison, we also re-implemented the original CKKS bootstrapping procedure [12] using the same tools. While our running times are not optimized, this setup should still allow for a meaningful and fair comparison that highlights the feasibility of our method. In support of transparency, reproducibility and accountability, we have decided to make our implementation publicly available online via GitHub.[2]

7.2 Parameter Selection

We provide parameter sets for both the PaCo and original CKKS bootstrapping procedures, targeting ring dimensions $N \in \{2^{15}, 2^{16}\}$. The secret key $s \in \mathbb{Z}[X]/(X^N + 1)$ is chosen with a Hamming weight $h = 64$, ternary for the original CKKS procedure, and structured as in Figure 1 for PaCo.

Two Different Scaling Factors *p* and *Δ*. To support flexible parameterization, we distinguish between two scaling factors: p and Δ. The factor $p \leqslant q$ is used for homomorphic operations outside bootstrapping, whereas $\Delta \geqslant q$ applies during

[2] `https://github.com/se-tim/PaCo-Implementation.git`

bootstrapping. Starting from a ciphertext under the modulus q, we bootstrap over the moduli $qp \cdot \Delta^\ell$ for $\ell \in [0, L]$ to finally output a ciphertext under the modulus qp, supporting one additional multiplication with scaling factor p. Only ciphertexts that encrypt polynomials with coefficients having absolute values at most p are used during our testings.

Multiplicative Depth. For all SlotToCoeff and CoeffToSlot operations, we use a grouping parameter equal to $g = 3$. The EvalMod operation of the original bootstrapping algorithm depends on two parameters d and r [12]; we set $d = 7$, and choose $r = 8$ for $N = 2^{15}$, and $r = 9$ for $N = 2^{16}$. For both bootstrapping procedures, the maximum level L is chosen to match the multiplicative depth required to bootstrap $B = N/(4h) = N/256$ coefficients. It is computed as follows:

- For PaCo, it is the sum of the multiplicative depths for the partial CoeffToSlot, the product operation, the plaintext–ciphertext multiplications, and the SlotToCoeff.
- For the original CKKS bootstrapping, it is the sum of the multiplicative depths for the CoeffToSlot, the EvalMod operation without successive squaring, the r successive squarings, and the SlotToCoeff.

Moduli for Switching Keys. In PaCo, all switching keys (for rotations, multiplications, *etc.*) are only used at $\ell = L - 1$ and below, and can thus be provided under a modulus $(pq \cdot \Delta^{L-1})^2$. In contrast, the original CKKS bootstrapping applies switching keys at the maximum level $\ell = L$, which necessitates a larger modulus of $(pq \cdot \Delta^L)^2$. These are the moduli we use to estimate the security of our parameter sets, applying the security estimator from [1]. Following conventional notation, we denote these moduli as PQ.

Parameter Sets. For each procedure, we provide two parameter sets, summarized in Table 4. Precision is quantified as the average number of most significant bits retained in the plaintext polynomial coefficients after bootstrapping. This level of precision is attained for bootstrapping $B = N/(4h) = N/256$ coefficients, using $\kappa = 1$ processor. All parameter sets achieve an estimated security level of approximately 100 bits or higher [1], based on standard estimators for the primal uSVP attack, the primal hybrid attack, the dual attack, and the dual hybrid attack.

Table 4. Parameter sets for the two bootstrapping procedures.

Set	$\log N$	$\log q$	$\log p$	$\log \Delta$	L	$\log(PQ)$	Precision
PaCo I	15	29	22	32	$3+6+3+2=14$	934	12 bits
Original I					$2+4+8+2=16$	1 126	
PaCo II	16	44	32	48	$3+6+3+3=15$	1 496	22 bits
Original II					$3+4+9+3=19$	1 976	

7.3 Benchmarks

We now present the results of our experiments. Our results clearly demonstrate that PaCo achieves substantial gains in efficiency, even without making use of parallelization. The results are shown in the form of graphs. Precise numerical values of the timings are omitted, as they would not offer meaningful insights given that our implementation only serves as a proof of concept.

Sequential Bootstrapping. We provide in Figure 5 the bootstrapping timings for the two parameter sets listed in Table 4. All timings are reported in seconds, and are averaged over 10 runs. The timings were obtained from a 2023 MacBook Air with an Apple M2 chip.

Even without parallelization, PaCo consistently outperforms the original CKKS bootstrapping method roughly by a constant term, regardless of the number of coefficients C. Therefore, the relative speedup of PaCo is more pronounced for smaller values of C, but remains modest even for larger values. Overall, the observed speedups range from approximately 1.5× to 2.6× compared to the original CKKS procedure.

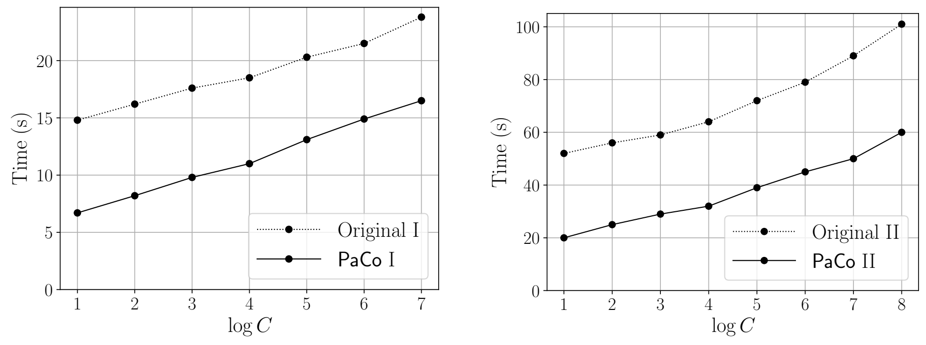

Fig. 5. Bootstrapping timings with the parameter sets listed in Table 4 to bootstrap C coefficients with $\log C \in [1, \log N - 8]$.

Parallel Bootstrapping. Thanks to the straightforward parallelization of PaCo, its runtime can be predicted when executed in parallel. Specifically, when using a power-of-two number of processors $\kappa \in [1, N/2]$ to bootstrap a power-of-two number of coefficients $D \in [2\kappa, \min(B\kappa, N)]$, the parallel runtime is approximately equal to the sequential runtime for $C = D/\kappa$ coefficients.

Still, to test this, we report timings for the PaCo II parameter set in Figure 6, using $\kappa = 32$ processors. These timings are not from actual parallel execution; instead, we ran the loop in Algorithm 5 sequentially and recorded the time of the longest iteration to simulate parallel performance. As expected, the timings for

$\log D \in [6, 13]$ with $\kappa = 32$ processors align closely with those for $\log C \in [1, 8]$ using a single processor. For comparison, we also include timings for sequential PaCo (where applicable) and the original CKKS procedure.

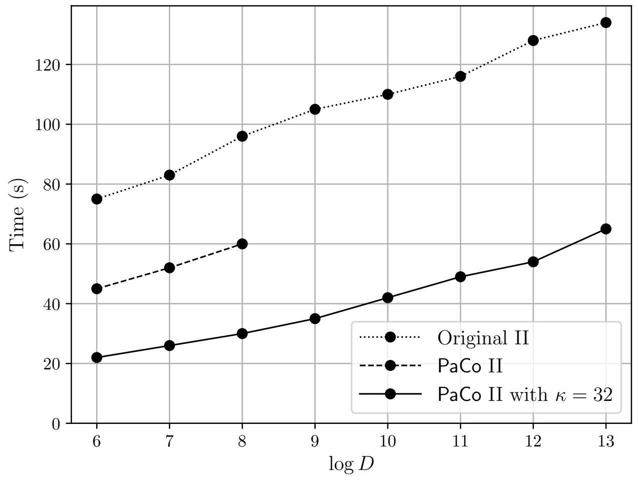

Fig. 6. Bootstrapping timings for PaCo II using $\kappa = 32$ processors, compared to PaCo II and Original II using $\kappa = 1$ processor, to bootstrap D coefficients with $\log D \in [6, 13]$.

8 Miscellaneous

We conclude with additional considerations that clarify the security foundations of PaCo and highlight practical aspects of its deployment.

8.1 Security

Let us justify why using a structured secret key for PaCo should not compromise the security of the overall scheme. Consider a secret key $s = \sum_{i=0}^{N-1} s_i X^i$ with a power-of-two Hamming weight $h \in [1, N]$, generated according to Algorithm 1, and let $B = N/(4h)$. The secret key is constructed such that for each $v \in [0, h)$, exactly one of the values $s_{u \cdot 4h + th + v}$ with $u \in [0, B)$ and $t \in [0, 4)$ is set to 1 (namely, $u = u_{t_v h + v}$ and $t = t_v$ in the notation of Section 5), while all others are set to 0; both $u \in [0, B)$ and $t \in [0, 4)$ are sampled uniformly at random, except if $v = 0$, in which case $u = 0 = t$. Equivalently, for each $v \in [0, h)$, exactly one of the values s_{uh+v} with $u \in [0, 4B)$ is set to 1, with u sampled uniformly at random from $[0, 4B)$, except if $v = 0$, in which case $u = 0$. Since $4B = N/h$, the secret key s takes the form illustrated in Figure 1, *i.e.*, the N coefficients of the secret key are partitioned into h blocks of size N/h, each containing exactly one entry equal to 1 and the rest equal to 0. The position of the 1 within each block is chosen uniformly at random, except in one block where the position is fixed. Therefore, the secret key s is uniformly sampled from a set S of cardinality $\#S = (N/h)^{h-1}$. This structured form does not weaken security: the analysis in Section 3.1 of [26] still applies. In particular, even advanced attacks such as May's meet-in-the-middle approach [29] would require at least $(\#S)^{0.25} = (N/h)^{0.25(h-1)}$ time, which is larger than 2^{128} for $N \geqslant 2^{15}$ and $h \in [64, N)$. Note that when using automated security estimators such as the one from [1], it is required to adjust the parameters accordingly: the ring dimension should be set to $N' = N(1-1/h)$, and the effective Hamming weight to $h' = h - 1$.

8.2 Sparse Secret Encapsulation

In our implementation, we use a secret key with a small Hamming weight $h = 64$, as this increases the number of coefficients $B = N/(4h)$ that can be bootstrapped efficiently with PaCo. However, a lower Hamming weight affects security. To mitigate this, one can use the concept of *sparse-secret encapsulation* [6]. More precisely, one can start with a general secret key (without the structured form of Figure 1, with higher Hamming weight and more general coefficients), and then perform a key switching to a PaCo secret key immediately before bootstrapping. The required switching key only needs to be provided for a small modulus, say q^2, thereby not affecting security.

8.3 Size of Bootstrapping Material

Recently, the SHIP bootstrapping algorithm [13] was introduced, which shares conceptual similarities with PaCo. While both approaches overlap in their use of blind rotations, a notable practical difference lies in the size of the bootstrapping material required. In contrast to SHIP, PaCo is notably lightweight in this regard. To illustrate this, we estimate the total bootstrapping material size for the lightest parameter sets of both methods. We use the estimation that a ciphertext at modulus Q occupies $2N \log Q \cdot 2^{-33}$ gigabytes, using the convention that one gigabyte equals 2^{33} bits.

For PaCo I, our implementation requires 29 switching keys for Galois automorphisms (conjugation and rotations) at modulus $(pq \cdot \Delta^{L-1})^2 = 2^{934}$, and four bootstrapping keys at modulus $pq \cdot \Delta^L = 2^{499}$, resulting in a total key size below 0.25 gigabytes.

On the other hand, the size of the bootstrapping material for SHIP is dominated by the blind rotation keys (see Algorithm 4 in [13]). For their smallest parameter set LL13 ($\log N = 13$, $\log(PQ) = 218$, $\theta = 6$, $h = 31$), a ciphertext at modulus $PQ = 2^{218}$ is needed for each j in an h-element set, for each $k \in [0, 4)$ and for each $j_1 \in [0, (N-1)/\theta]$, resulting in a total size exceeding 70 gigabytes.

While we do not claim that the two parameter sets are directly comparable, this estimation still demonstrates that the two approaches have fundamentally different resource requirements, highlighting a core distinction in their design.

9 Conclusion and Future Work

We presented PaCo, an efficient and parallelizable bootstrapping method for CKKS with zero failure probability. This approach is enabled by reformulating the decryption equation using blind rotations and modular additions, made possible by adopting a structured secret key. The decryption circuit is evaluated homomorphically within the CKKS framework by using the circle group to approximate modular reductions, and polynomial ring structures to perform blind rotations, the latter enabled by a partial CoeffToSlot operation.

Future directions include exploring if PaCo can be adapted to bootstrap all N coefficients without parallelization, as well as developing an optimized implementation using a residue number system (RNS). A slight reordering of the algorithmic steps may allow PaCo to be integrated with small-integer bootstrapping techniques such as those in [4], which could be an interesting direction for future work.

Also, with increasing interest in blind rotation techniques for CKKS, it would be important to assess which strategies are best suited for various applications, considering recent developments such as those introduced in [13].

Acknowledgments. Tim Seuré acknowledges the support of the Luxembourgish "Fonds National de la Recherche" (FNR) through an Individual Grant (reference number: 17936291).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Albrecht, M., Player, R., Scott, S.: On the Concrete Hardness of Learning with Errors. Journal of Mathematical Cryptology **9** (2015)
2. Bae, Y., Cheon, J.H., Cho, W., Kim, J., Kim, T.: META-BTS: Bootstrapping Precision Beyond the Limit. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. ACM (2022)
3. Bae, Y., Cheon, J.H., Kim, J., Stehlé, D.: Bootstrapping Bits with CKKS. In: Advances in Cryptology – EUROCRYPT 2024. Springer (2024)
4. Bae, Y., Kim, J., Stehlé, D., Suvanto, E.: Bootstrapping Small Integers with CKKS. In: Advances in Cryptology – ASIACRYPT 2024. Springer (2024)
5. Bossuat, J.P., Mouchet, C., Troncoso-Pastoriza, J., Hubaux, J.P.: Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-Sparse Keys. In: Advances in Cryptology – EUROCRYPT 2021. Springer (2021)
6. Bossuat, J.P., Troncoso-Pastoriza, J., Hubaux, J.P.: Bootstrapping for Approximate Homomorphic Encryption with Negligible Failure-Probability by Using Sparse-Secret Encapsulation. In: Applied Cryptography and Network Security. Springer (2022)
7. Brakerski, Z.: Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In: Advances in Cryptology – CRYPTO 2012. Springer (2012)

8. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully Homomorphic Encryption without Bootstrapping. In: ITCS 2012: Innovations in Theoretical Computer Science. ACM (2012)
9. Chen, H., Chillotti, I., Song, Y.: Improved Bootstrapping for Approximate Homomorphic Encryption. In: Advances in Cryptology – EUROCRYPT 2019. Springer (2019)
10. Chen, H., Dai, W., Kim, M., Song, Y.: Efficient Homomorphic Conversion Between (Ring) LWE Ciphertexts. In: Applied Cryptography and Network Security. Springer (2021)
11. Cheon, J.H., Han, K., Hhan, M.: Improved Homomorphic Discrete Fourier Transforms and FHE Bootstrapping. IEEE Access **7** (2019)
12. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: Bootstrapping for Approximate Homomorphic Encryption. In: Advances in Cryptology – EUROCRYPT 2018. Springer (2018)
13. Cheon, J.H., Han, K., Kim, J., Stehlé, D.: SHIP: A Shallow and Highly Parallelizable CKKS Bootstrapping Algorithm. In: Advances in Cryptology – EUROCRYPT 2025. Springer (2025)
14. Cheon, J.H., Kim, D., Kim, D., Song, Y.: Homomorphic Encryption for Approximate Numbers. In: Advances in Cryptology – ASIACRYPT 2017. Springer (2017)
15. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In: Advances in Cryptology – ASIACRYPT 2016. Springer (2016)
16. Cooley, J.W., Tukey, J.: An Algorithm for the Machine Calculation of Complex Fourier Series. Mathematics of Computation **19** (1965)
17. Developers, S.: SageMath, the Sage Mathematics Software System (Version 10.2) (2023), `https://www.sagemath.org`
18. Ducas, L., Micciancio, D.: FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In: Advances in Cryptology – EUROCRYPT 2015. Springer (2015)
19. Fan, J., Vercauteren, F.: Somewhat Practical Fully Homomorphic Encryption (2012), `https://eprint.iacr.org/2012/144`
20. Gentry, C.: Fully Homomorphic Encryption Using Ideal Lattices. In: Proceedings of the Annual ACM Symposium on Theory of Computing. ACM (2009)
21. Gentry, C., Sahai, A., Waters, B.: Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In: Advances in Cryptology – CRYPTO 2013. Springer (2013)
22. Halevi, S., Shoup, V.: Algorithms in HElib. In: Advances in Cryptology – CRYPTO 2014. Springer (2014)
23. Han, K., Ki, D.: Better Bootstrapping for Approximate Homomorphic Encryption. In: Topics in Cryptology – CT-RSA 2020. Springer (2020)
24. Jutla, C., Manohar, N.: Sine Series Approximation of the Mod Function for Bootstrapping of Approximate HE. In: Advances in Cryptology – EUROCRYPT 2022. Springer (2022)
25. Kim, A., Deryabin, M., Eom, J., Choi, R., Lee, Y., Ghang, W., Yoo, D.: General Bootstrapping Approach for RLWE-Based Homomorphic Encryption. IEEE Transactions on Computers **73** (2024)
26. Lee, C., Min, S., Seo, J., Song, Y.: Faster TFHE Bootstrapping with Block Binary Keys. In: Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security. ACM (2023)
27. Lee, E., Lee, J.W., Kim, Y.S., Kim, Y., No, J.S.: High-Precision Bootstrapping for Approximate Homomorphic Encryption by Error Variance Minimization. In: Advances in Cryptology – EUROCRYPT 2022. Springer (2022)

28. Lyubashevsky, V., Peikert, C., Regev, O.: On Ideal Lattices and Learning with Errors over Rings. In: Advances in Cryptology – EUROCRYPT 2010. Springer (2010)
29. May, A.: How to Meet Ternary LWE Keys. In: Advances in Cryptology – CRYPTO 2021. Springer (2021)
30. Shoup, V.: NTL: A Library for doing Number Theory (Version 11.5.1) (2021), `https://www.shoup.net/ntl/`
31. Smart, N.P., Vercauteren, F.: Fully Homomorphic SIMD Operations. Designs, Codes and Cryptography **71** (2014)
32. Stehlé, D., Steinfeld, R., Tanaka, K., Xagawa, K.: Efficient Public Key Encryption Based on Ideal Lattices. In: Advances in Cryptology – ASIACRYPT 2009. Springer (2009)

A Tridiagonality Proof for the Matrices $\boldsymbol{E}^{(B)}_{n,2^\ell}$

In this section, we provide a proof of the fact that the matrices $\boldsymbol{E}^{(B)}_{n,2^\ell}$ are tridiagonal. We begin with the following preliminary result:

Lemma 1. *Let $n \geqslant B$ be powers-of-two and let $\ell \in [0, \log B)$. Also fix two integers $i \in [0, 2^\ell n/B)$ and $j \in [0, B/2^{\ell+1})$. Then the following equality holds true:*

$$\mathsf{br}_{n/2^{\ell+1}}\left(\left\lfloor \frac{\mathsf{br}_B(iB/2^\ell + j)}{2^{\ell+1}} \right\rfloor\right) = \frac{jn}{B} + \mathsf{br}_{n/B}\left(\left\lfloor \frac{i}{2^\ell} \right\rfloor\right).$$

Proof. Let us introduce the integers $i_0 = [i]_{2^\ell}$ and $i_1 = \lfloor i/2^\ell \rfloor$, so that we can write $i = 2^\ell i_1 + i_0$. We then have:

$$\begin{aligned}\mathsf{br}_B(iB/2^\ell + j) &= \mathsf{br}_B(i_1 B + i_0 B/2^\ell + j)\\ &= i_1 B + \mathsf{br}_B(i_0 B/2^\ell + j)\\ &= i_1 B + \mathsf{br}_{B/2^{\ell+1}}(j) \cdot 2^{\ell+1} + \mathsf{br}_{2^\ell}(i_0).\end{aligned}$$

Dividing by $2^{\ell+1}$ and rounding down gives:

$$\left\lfloor \frac{\mathsf{br}_B(iB/2^\ell + j)}{2^{\ell+1}} \right\rfloor = \frac{i_1 B}{2^{\ell+1}} + \mathsf{br}_{B/2^{\ell+1}}(j).$$

Applying $\mathsf{br}_{n/2^{\ell+1}}$ then yields the announced equality. □

Proposition 1. *Let $n \geqslant B$ be powers of two and let $\ell \in [0, \log n)$. Then the matrix $\boldsymbol{E}^{(B)}_{n,2^\ell}$ is equal to $\boldsymbol{D}_{n,2^\ell}$ in case $2^\ell \geqslant B$; otherwise, it is a block-diagonal matrix consisting of the following $2^\ell n/B$ blocks of size $(B/2^\ell) \times (B/2^\ell)$ on its diagonal:*

$$\left(\begin{array}{c|c} \boldsymbol{I}_{B/2^{\ell+1}} & \mathsf{Diag}\big(\zeta^{2^\ell}_{n,jn/B+\mathsf{br}_{n/B}(\lfloor i/2^\ell \rfloor)}\big)_{j\in[0,B/2^{\ell+1})} \\ \hline \boldsymbol{I}_{B/2^{\ell+1}} & -\mathsf{Diag}\big(\zeta^{2^\ell}_{n,jn/B+\mathsf{br}_{n/B}(\lfloor i/2^\ell \rfloor)}\big)_{j\in[0,B/2^{\ell+1})} \end{array}\right), \quad i \in [0, 2^\ell n/B).$$

Further, the inverse of $\boldsymbol{E}^{(B)}_{n,2^\ell}$ is given by $1/2 \cdot \overline{\boldsymbol{E}^{(B)}_{n,2^\ell}}^{\mathsf{T}}$. In particular, both $\boldsymbol{E}^{(B)}_{n,2^\ell}$ and its inverse are tridiagonal matrices.

Proof. Concerning the statement about the inverse, we have:

$$\begin{aligned}
\boldsymbol{E}^{(B)}_{n,2^\ell} \cdot \frac{1}{2}\overline{\boldsymbol{E}^{(B)}_{n,2^\ell}}^{\mathsf{T}} &= \boldsymbol{\Pi}^{(B)}\boldsymbol{D}_{n,2^\ell}\boldsymbol{\Pi}^{(B)} \cdot \frac{1}{2}\boldsymbol{\Pi}^{(B)}\overline{\boldsymbol{D}_{n,2^\ell}}^{\mathsf{T}}\boldsymbol{\Pi}^{(B)} \\
&= \boldsymbol{\Pi}^{(B)} \cdot \left(\boldsymbol{D}_{n,2^\ell} \cdot \frac{1}{2}\overline{\boldsymbol{D}_{n,2^\ell}}^{\mathsf{T}}\right) \cdot \boldsymbol{\Pi}^{(B)} \\
&= \boldsymbol{\Pi}^{(B)}\boldsymbol{I}_n\boldsymbol{\Pi}^{(B)} \\
&= \boldsymbol{I}_n.
\end{aligned}$$

Let us now treat the structure of the matrix $\boldsymbol{E}^{(B)}_{n,2^\ell}$. Assume first that $2^\ell \geqslant B$. The matrix $\boldsymbol{\Pi}^{(B)}_n$ is block-diagonal with $n/2^\ell$ copies of the block $\boldsymbol{\Pi}^{(B)}_{2^\ell}$ on its diagonal. The statement that $\boldsymbol{E}^{(B)}_{n,2^\ell} = \boldsymbol{D}_{n,2^\ell}$ now follows from the fact that simultaneously left- and right-multiplying any of the blocks $\boldsymbol{A}$ of size $2^{\ell+1} \times 2^{\ell+1}$ appearing in (5) by the matrix:

$$\left(\begin{array}{c|c} \boldsymbol{\Pi}^{(B)}_{2^\ell} & \boldsymbol{0}_{2^\ell} \\ \hline \boldsymbol{0}_{2^\ell} & \boldsymbol{\Pi}^{(B)}_{2^\ell} \end{array}\right) \in \mathbb{C}^{2^{\ell+1}\times 2^{\ell+1}}$$

gives back the original block $\boldsymbol{A}$. Let us now assume that $B \geqslant 2^{\ell+1}$. We fix $\boldsymbol{z} = (z_i)_{i\in[0,n)} \in \mathbb{C}^n$, and introduce the two vectors $\boldsymbol{v} = (v_i)_{i\in[0,n)} = \boldsymbol{D}_{n,2^\ell}\boldsymbol{\Pi}^{(B)}\boldsymbol{z}$ and $\boldsymbol{w} = (w_i)_{i\in[0,n)} = \boldsymbol{\Pi}^{(B)}\boldsymbol{v}$. The goal will be to prove that the entries of $\boldsymbol{w}$ satisfy the following equalities, for every $i \in [0, 2^\ell n/B)$, $j \in [0, B/2^{\ell+1})$ and $\delta \in [0,2)$:

$$w_{iB/2^\ell+\delta B/2^{\ell+1}+j} = z_{iB/2^\ell+j} + (-1)^\delta \cdot \zeta^{2^\ell}_{n,jn/B+\mathsf{br}_{n/B}(\lfloor i/2^\ell\rfloor)} \cdot z_{iB/2^\ell+B/2^{\ell+1}+j}.$$

Let us first consider the vector $\boldsymbol{v}$. For every $\hat{\imath} \in [0, n/2^{\ell+1})$, $\hat{\jmath} \in [0, 2^\ell)$ and $\delta \in [0,2)$, we have:

$$\begin{aligned}
v_{\hat{\imath}2^{\ell+1}+\delta 2^\ell+\hat{\jmath}} &= z_{\mathsf{br}_B(\hat{\imath}2^{\ell+1}+\hat{\jmath})} + (-1)^\delta \cdot \zeta^{2^\ell}_{n,\mathsf{br}_{n/2^{\ell+1}}(\hat{\imath})} \cdot z_{\mathsf{br}_B(\hat{\imath}2^{\ell+1}+2^\ell+\hat{\jmath})} \\
&= z_{\mathsf{br}_B(\hat{\imath}2^{\ell+1}+\hat{\jmath})} + (-1)^\delta \cdot \zeta^{2^\ell}_{n,\mathsf{br}_{n/2^{\ell+1}}(\hat{\imath})} \cdot z_{\mathsf{br}_B(\hat{\imath}2^{\ell+1}+\hat{\jmath})+B/2^{\ell+1}}.
\end{aligned}$$

Let us now consider the vector $\boldsymbol{w}$. Fixing $i \in [0, 2^\ell n/B)$, $j \in [0, B/2^{\ell+1})$ and $\delta \in [0,2)$, and writing $\mathsf{br}_B(iB/2^\ell + j) = \hat{\imath}2^{\ell+1} + \hat{\jmath}$ with $\hat{\imath} \in [0, n/2^{\ell+1})$ and $\hat{\jmath} \in [0, 2^\ell)$, we get:

$$\begin{aligned}
w_{iB/2^\ell+\delta B/2^{\ell+1}+j} &= v_{\mathsf{br}_B(iB/2^\ell+\delta B/2^{\ell+1}+j)} \\
&= v_{\mathsf{br}_B(iB/2^\ell+j)+\delta 2^\ell} \\
&= v_{\hat{\imath}2^{\ell+1}+\delta 2^\ell+\hat{\jmath}} \\
&= z_{\mathsf{br}_B(\hat{\imath}2^{\ell+1}+\hat{\jmath})} + (-1)^\delta \cdot \zeta^{2^\ell}_{n,\mathsf{br}_{n/2^{\ell+1}}(\hat{\imath})} \cdot z_{\mathsf{br}_B(\hat{\imath}2^{\ell+1}+\hat{\jmath})+B/2^{\ell+1}} \\
&= z_{iB/2^\ell+j} + (-1)^\delta \cdot \zeta^{2^\ell}_{n,\mathsf{br}_{n/2^{\ell+1}}(\hat{\imath})} \cdot z_{iB/2^\ell+B/2^{\ell+1}+j}.
\end{aligned}$$

Since $\hat{\imath} = \lfloor \mathsf{br}_B(iB/2^\ell + j)/2^{\ell+1}\rfloor$, we can apply Lemma 1 to obtain $\mathsf{br}_{n/2^{\ell+1}}(\hat{\imath}) = jn/B + \mathsf{br}_{n/B}(\lfloor i/2^\ell\rfloor)$. □

B Proof of the Packing Relation

In this section, we prove the packing relation (11), which enables the first step of the PaCo algorithm.

Proposition 2. *The vectors $\boldsymbol{\beta}_t$ and $\boldsymbol{\sigma}_t$ satisfy the following equality:*

$$\left(\prod_{\ell=0}^{\log C}\left(\boldsymbol{E}_{n,2^\ell}^{(C/2)}\right)^{-1}\right)\cdot \mathsf{Tr}_{N/2\to n}\left(\sum_{t=0}^{3}\left(\boldsymbol{\Pi}^{(C/2)}\boldsymbol{\beta}_t\right)\odot\left(\boldsymbol{\Pi}^{(C/2)}\boldsymbol{\sigma}_t\right)\right)$$
$$=\left(\left(b'_{\lambda_v,\mathsf{br}_{C/2}(i)}\right)_{i\in[0,2C)}\right)_{v\in[0,h)}.$$

Proof. Denote the vector on the left-hand side by $\boldsymbol{z}$. We begin by factoring the permutation matrix $\boldsymbol{\Pi}^{(C/2)}$ out of the summation, use that bit-reversing commutes with the trace operation, and move the trace operation inside the summation over t:

$$\boldsymbol{z}=\left(\prod_{\ell=0}^{\log C}\left(\boldsymbol{E}_{n,2^\ell}^{(C/2)}\right)^{-1}\right)\cdot\boldsymbol{\Pi}^{(C/2)}\sum_{t=0}^{3}\mathsf{Tr}_{N/2\to n}(\boldsymbol{\beta}_t\odot\boldsymbol{\sigma}_t).$$

Recall that first bit-reversing and then applying a sequence of matrices $\left(\boldsymbol{E}_{n,2^\ell}^{(C/2)}\right)^{-1}$ is equivalent to applying the sequence of the corresponding $\boldsymbol{D}_{n,2^\ell}^{-1}$ matrices followed by bit-reversing. Using this, and evaluating the trace, we obtain:

$$\boldsymbol{z}=\boldsymbol{\Pi}^{(C/2)}\cdot\left(\prod_{\ell=0}^{\log C}\boldsymbol{D}_{n,2^\ell}^{-1}\right)\cdot\sum_{t=0}^{3}\sum_{r=0}^{k-1}\boldsymbol{\beta}_t^{(r)}\odot\boldsymbol{\sigma}_t^{(r)}.$$

Therefore, it suffices to show:

$$\sum_{r=0}^{k-1}\sum_{t=0}^{3}\boldsymbol{\beta}_t^{(r)}\odot\boldsymbol{\sigma}_t^{(r)}=\left(\prod_{\ell=0}^{\log C}\boldsymbol{D}_{n,C/2^\ell}\right)\cdot\left[\left(b'_{\lambda_v}\right)_{v\in[0,h)}\right].$$

Towards this, recall that the composition $\phi_{n,1}\circ\cdots\circ\phi_{n,C}$ defines a ring morphism, and it is expressed by the product of matrices $\boldsymbol{D}_{n,C/2^\ell}$. Furthermore, a sum of the form $\sum_{t=0}^{3}d_{th+v}\cdot p_{th+v}$ simplifies to p_{λ_v}. With these, we obtain from the definition of $\boldsymbol{\beta}_t^{(r)}$ and $\boldsymbol{\sigma}_t^{(r)}$ that the following holds, for each $r\in[0,k)$:

$$\sum_{t=0}^{3}\boldsymbol{\beta}_t^{(r)}\odot\boldsymbol{\sigma}_t^{(r)}=\left(\prod_{\ell=0}^{\log C}\boldsymbol{D}_{n,C/2^\ell}\right)\cdot\left[\left(\mathbb{1}_{u_{\lambda_v}+r=0\bmod k}\cdot Z^{(u_{\lambda_v}+r)/k}\cdot\tilde{b}_{\lambda_v}^{(r)}\right)_{v\in[0,h)}\right].$$

Summing over $r\in[0,k)$ yields the result. □

C More Details on Extracting the Coefficients of $\tilde{\boldsymbol{m}}$

We have claimed that the complex vector underlying the ciphertext in line 15 of Algorithm 4 is approximately equal to $1/\Delta \cdot \boldsymbol{\Pi}^{(C/2)}(\tilde{\boldsymbol{m}}_0 + \mathrm{I} \cdot \tilde{\boldsymbol{m}}_1)$. To justify this claim, let us denote by $\boldsymbol{z}_\ell$ the complex vector associated with the ciphertext produced in line $\ell \in [7, 15]$ of the algorithm. Then we have:

$$\boldsymbol{z}_7 \approx \boldsymbol{\Pi}^{(C/2)}\Big(\big(b'_{\lambda_v,i}\big)_{i\in[0,2C)}\Big)_{v\in[0,h)}, \qquad \boldsymbol{z}_{10} \approx \boldsymbol{\mu} \odot \Big(\boldsymbol{z}_7 + \overline{\mathsf{Rot}_C(\boldsymbol{z}_7)}\Big).$$

Therefore:

$$\begin{aligned}
\boldsymbol{z}_{10} &\approx \boldsymbol{\Pi}^{(C/2)}\left(\left(\mathbb{1}_{i<C} \cdot \left(b'_{\lambda_v,i} + \overline{b'_{\lambda_v,i+C}}\right)\right)_{i\in[0,2C)}\right)_{v\in[0,h)} \\
\boldsymbol{z}_{11} &\approx \boldsymbol{\Pi}^{(C/2)}\left(\mathbb{1}_{i<C} \cdot \left(\prod_{v=0}^{h-1}\left(b'_{\lambda_v,i} + \overline{b'_{\lambda_v,i+C}}\right)\right)\right)_{i\in[0,2C)} \\
\boldsymbol{z}_{12} &\approx \boldsymbol{\Pi}^{(C/2)}\left(\prod_{v=0}^{h-1}\left(b'_{\lambda_v,i} + \overline{b'_{\lambda_v,i+C}}\right)\right)_{i\in[0,C)} \\
\boldsymbol{z}_{13} &\approx \frac{4\pi\mathrm{I}}{q} \cdot \boldsymbol{\Pi}^{(C/2)}(\tilde{m}_i)_{i\in[0,C)} \\
\boldsymbol{z}_{14} &\approx \frac{1}{\Delta} \cdot \boldsymbol{\Pi}^{(C/2)}(\mathbb{1}_{i<C/2} \cdot \tilde{m}_i + \mathrm{I} \cdot \mathbb{1}_{i\geqslant C/2} \cdot \tilde{m}_i)_{i\in[0,C)} \\
\boldsymbol{z}_{15} &\approx \frac{1}{\Delta} \cdot \boldsymbol{\Pi}^{(C/2)}(\tilde{\boldsymbol{m}}_0 + \mathrm{I} \cdot \tilde{\boldsymbol{m}}_1).
\end{aligned}$$

For $\boldsymbol{z}_{10}$, we have applied that $\boldsymbol{\Pi}^{(C/2)}\boldsymbol{\mu} = \boldsymbol{\mu}$. The approximation for $\boldsymbol{z}_{13}$ relies on the approximation for the coefficients of $\tilde{m}$, see (10). For $\boldsymbol{z}_{14}$, we used $\boldsymbol{\Pi}^{(C/2)}\boldsymbol{\eta} = \boldsymbol{\eta}$. In general, we have repeatedly made use of the fact that bit-reversing commutes with both the trace and the product operations.