SCORE: A SlotToCoeff Optimization for Real-Vector Encryption in CKKS

Tim Seuré

University of Luxembourg, Esch-sur-Alzette, Luxembourg
tim.seure@uni.lu

Abstract. We present SCORE, a tweaked version of the SlotToCoeff operation tailored to the context of encrypted real vectors in the CKKS scheme, where SCORE stands for *SlotToCoeff Optimization for Real-Vector Encryption*. This approach accelerates CKKS bootstrapping algorithms that employ the SlotToCoeff operation as their final step, provided the inputs are encryptions of real vectors. SCORE constitutes a conceptually simple refinement of the conventional SlotToCoeff step that can be seamlessly incorporated into existing bootstrapping algorithms. Despite its minimal nature, the technique yields tangible performance improvements. We demonstrate its utility through proof-of-concept implementations for two such algorithms: the conventional bootstrapping algorithm and the SPRU algorithm, which is particularly efficient for ciphertexts containing a small number of encrypted slots. Our results indicate notable performance gains of up to 2× compared to the original formulations of the algorithms.

Keywords: Homomorphic encryption · Bootstrapping · RLWE · Lattices.

1 Introduction

1.1 General Overview

Homomorphic Encryption. Homomorphic encryption (HE) enables computation directly on encrypted data, ensuring that sensitive information remains protected throughout processing. Fully homomorphic encryption (FHE) generalizes this capability by supporting the evaluation of arbitrarily complicated circuits on the ciphertexts; one says that the circuits are applied *homomorphically* on the encrypted data. Since Gentry's revolutionary idea of bootstrapping in 2009 [24], numerous FHE schemes have been proposed, all grounded in the hardness of lattice-based problems such as Ring Learning With Errors (RLWE) [29,31]. The schemes typically differ in their supported data types. For instance, BGV and BFV [7,8,22] operate over finite field vectors, FHEW and TFHE [16,20] are optimized for binary data, while CKKS [15] supports approximate arithmetic over complex vectors. The CKKS scheme is well-suited for practical applications requiring real-valued computations, as is for instance the case for privacy-preserving machine learning [23,21].

The CKKS Scheme. Among modern HE schemes, CKKS stands out for its support of approximate componentwise arithmetic on complex vectors. It enables operations such as addition, multiplication, conjugation and slot rotations, thereby making it well-suited for practical privacy-preserving applications.

A key limitation of CKKS is its reliance on rescaling after each homomorphic multiplication. Each rescaling step reduces the available *modulus*, and after a limited number of such steps, the ciphertext no longer has sufficient modulus for further homomorphic computations. This issue is addressed by *bootstrapping* [10,13,14,18,19,27], a technique that refreshes a ciphertext to restore its modulus, effectively enabling indefinite homomorphic computations.

Although CKKS is designed for approximate arithmetic on complex vectors $\boldsymbol{z} \in \mathbb{C}^{N/2}$ for some power of two N, its security relies on the RLWE problem. Therefore, before encryption, a complex vector $\boldsymbol{z}$ must be converted into an integer polynomial $m \in \mathbb{Z}[X]/(X^N+1)$. This conversion relies on a ring isomorphism $\tau_{N/2} : \mathbb{R}[X]/(X^N+1) \to \mathbb{C}^{N/2}$, a version of the discrete Fourier transform. Using the inverse map $\tau_{N/2}^{-1}$, the vector $\boldsymbol{z}$ is first mapped to a polynomial $p = \tau_{N/2}^{-1}(\boldsymbol{z}) \in \mathbb{R}[X]/(X^N+1)$. Then, it is multiplied by a large integer Δ, the *scaling factor*, to preserve numerical precision, and finally each coefficient is rounded to obtain an integer polynomial $m = \lfloor \Delta p \rceil \in \mathbb{Z}[X]/(X^N+1)$. The transformation from complex vectors to integer polynomials is known as *encoding*, and the reverse process is referred to as *decoding*.

Now, a *CKKS encryption* of a plaintext polynomial $m \in \mathbb{Z}[X]/(X^N+1)$ (and thus of the complex vector $\boldsymbol{z} \in \mathbb{C}^{N/2}$ it encodes) is a tuple $\mathbf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1)$ 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 some small error $e \in \mathbb{Z}[X]/(X^N+1)$, where the polynomial $s \in \mathbb{Z}[X]/(X^N+1)$ is the secret key and q is a large modulus. Consequently, the polynomial m (and therefore the complex vector $\boldsymbol{z}$ it encodes) cannot be recovered exactly due to the presence of the error e, which reflects the approximate nature of the CKKS scheme.

The CoeffToSlot and SlotToCoeff Operations. All existing CKKS bootstrapping algorithms rely on at least one of two fundamental operations: *CoeffToSlot* and *SlotToCoeff* [13]. Given an encryption of a complex vector $\boldsymbol{z} \in \mathbb{C}^{N/2}$, the *CoeffToSlot operation* extracts the coefficients p_i of the polynomial $p = \sum_{i=0}^{N-1} p_i X^i = \tau_{N/2}^{-1}(\boldsymbol{z})$ and maps them into the slots of an encrypted vector. This results in an encryption of the vector $\boldsymbol{p}_0 + \mathrm{I} \cdot \boldsymbol{p}_1 \in \mathbb{C}^{N/2}$, where $\boldsymbol{p}_0 = (p_i)_{i \in [0, N/2)}$ and $\boldsymbol{p}_1 = (p_{i+N/2})_{i \in [0, N/2)}$, with $\mathrm{I} = \sqrt{-1} \in \mathbb{C}$ denoting the imaginary unit. The coefficients of the polynomial p can then be obtained by taking real and imaginary parts. The *SlotToCoeff operation* reverses this transformation: starting from an encryption of the vector $\boldsymbol{p}_0 + \mathrm{I} \cdot \boldsymbol{p}_1 \in \mathbb{C}^{N/2}$ with $\boldsymbol{p}_0 = (p_i)_{i \in [0, N/2)} \in \mathbb{R}^{N/2}$ and $\boldsymbol{p}_1 = (p_{i+N/2})_{i \in [0, N/2)} \in \mathbb{R}^{N/2}$, it outputs an encryption of the plaintext polynomial $m = \lfloor \Delta p \rceil \in \mathbb{Z}[X]/(X^N+1)$, where $p = \sum_{i=0}^{N-1} p_i X^i$. The first half of the coefficients of m therefore stem from the real part of $\boldsymbol{p}_0 + \mathrm{I} \cdot \boldsymbol{p}_1$, and the second half from its imaginary part.

CKKS Bootstrapping. *Bootstrapping* in the context of CKKS refers to the process of refreshing a ciphertext by restoring its modulus. More formally, we consider a ciphertext $\mathbf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1)$ satisfying $\mathsf{ct}_0 + s \cdot \mathsf{ct}_1 = m \bmod q$ for a "small" modulus q (often times, q equals q_0, the base modulus of the scheme), where we treat the inherent error as part of the plaintext polynomial $m \in \mathbb{Z}[X]/(X^N + 1)$. Bootstrapping generates a new ciphertext $\mathbf{ct}' = (\mathsf{ct}'_0, \mathsf{ct}'_1)$ such that $\mathsf{ct}'_0 + s \cdot \mathsf{ct}'_1 = m + e \bmod q'$ for some larger modulus $q' \gg q$ and a small error $e \in \mathbb{Z}[X]/(X^N + 1)$.

A variety of bootstrapping techniques have been developed for CKKS, beginning with the conventional method proposed by Cheon *et al.* [13]. This approach initiates the process with the *ModRaise step*, which reinterprets the ciphertext under a much larger modulus. However, this reinterpretation introduces unwanted multiples of the original modulus q into the plaintext polynomial. To eliminate these unwanted terms, the reduction-modulo-q function is evaluated homomorphically. This is accomplished by first using the CoeffToSlot operation to map the polynomial coefficients into the slots of an encrypted vector. An approximation of the reduction-modulo-q function is then applied homomorphically, constituting the *EvalMod step*. Finally, the reduced coefficients are mapped back from the slots to the polynomial coefficients using the SlotToCoeff operation. This yields a bootstrapped ciphertext. One simple variation of this involves reordering the standard steps: performing SlotToCoeff before ModRaise, followed by CoeffToSlot and then EvalMod (see, *e.g.*, [2]). These two approaches are referred to as *CoeffToSlot-first bootstrapping* and *SlotToCoeff-first bootstrapping*, respectively.

Note that in the SlotToCoeff-first approach, the input ciphertext must still be at a sufficiently large modulus to accommodate the SlotToCoeff operation before ModRaise. Consequently, all homomorphic computations preceding bootstrapping must be carried out at this elevated modulus or higher, which in turn increases their cost. Since the complexity of homomorphic operations scales roughly linearly with the bit-length of the modulus, an operation performed at modulus $q' > q$ is about $\log q' / \log q$ times slower than at modulus q. For example, reserving three multiplicative levels for SlotToCoeff implies that an operation immediately before bootstrapping which consumes one level is executed at modulus $q_0 \cdot \Delta^4$ in the SlotToCoeff-first variant (where q_0 is the base modulus of the scheme and Δ is the scaling factor), compared to $q_0 \cdot \Delta$ in the CoeffToSlot-first variant. Assuming $q_0 \approx \Delta$, this results in an approximate slowdown factor of $\log(\Delta^5)/\log(\Delta^2) = 2.5$. On the other hand, because SlotToCoeff is performed at the lowest moduli, the bootstrapping phase itself is more efficient in the SlotToCoeff-first approach. Hence, the two variants offer complementary trade-offs.

There are other bootstrapping algorithms which target specific types of plaintexts. For instance, Bae *et al.* [3] proposed several techniques optimized for ciphertexts encoding binary data. In a related effort, Bae *et al.* introduced SI-BTS [4], a bootstrapping approach tailored to encrypted vectors whose slots contain small integers; their algorithm in particular enables functional bootstrapping.

More recently, new algorithms have emerged that adopt a design philosophy closer to that of other FHE schemes, where bootstrapping is realized by homomorphically replicating the decryption circuit by using roots of unity to emulate

addition modulo q_0 in the complex plane. Examples of such methods include the SHIP [14], SPRU [18], and PaCo [19] algorithms.

Our Contribution. We introduce SCORE, a variant of the SlotToCoeff operation for encrypted real vectors in the CKKS scheme. By exploiting a skew-symmetric structure of plaintext polynomials encoding real vectors, SCORE enables bootstrapping algorithms to process only half of the polynomials' coefficients, reducing computational overhead without affecting correctness. This approach yields notable speedups of up to $2\times$. While conventional methods achieve similar speedups by packing two real vectors into the real and imaginary parts of a complex vector, our technique provides substantial performance gains when bootstrapping only a single encrypted real vector. We note that the applicability of SCORE is limited to bootstrapping procedures whose final step is a SlotToCoeff operation, and the resulting speedup depends on the extent to which preceding steps can benefit from processing only half as many polynomial coefficients.

1.2 Overview of New Approach

Motivation. Our approach is motivated by two questions. First, why restrict ourselves to real vectors in CKKS, a scheme originally designed for complex vectors? Second, why focus exclusively on CoeffToSlot-first bootstrapping algorithms?

Regarding the first point, the primary strength of CKKS compared to other FHE schemes lies in its support for approximate arithmetic, which is particularly well suited for real-world data arising in machine learning, signal processing, and statistical analysis, see for instance [9,23,21,30]. In such applications, the data is naturally represented as real vectors and rarely carries a meaningful imaginary component. Restricting to real vectors is therefore not artificial, but reflects the dominant practical use cases of CKKS. We note that a real-only variant of CKKS was proposed in [28], which avoids wasting the imaginary part of the slots at the encoding level. However, this variant does not include a dedicated bootstrapping procedure; in practice, one must still rely on standard CKKS bootstrapping, which can therefore benefit from our optimization.

Concerning the second point, CoeffToSlot-first and SlotToCoeff-first bootstrapping exhibit complementary trade-offs: CoeffToSlot-first is typically more efficient for homomorphic computations performed prior to bootstrapping, whereas SlotToCoeff-first can reduce the cost of the bootstrapping phase itself. The SlotToCoeff-first paradigm already admits an optimization for real vectors, where the imaginary part is removed before the EvalMod step. In contrast, no analogous optimization is currently available for the CoeffToSlot-first setting, and we will shortly clarify the specific obstacle that arises in this context. Our work addresses this gap by providing a real-vector-optimized variant of CoeffToSlot-first bootstrapping.

How SCORE Works. Let $\boldsymbol{z} \in \mathbb{R}^{N/2}$ be a real-valued vector and let $p = \sum_{i=0}^{N-1} p_i X^i = \tau_{N/2}^{-1}(\boldsymbol{z}) \in \mathbb{R}[X]/(X^N + 1)$ be its associated polynomial. After

scaling and rounding, we obtain the plaintext polynomial $m = \sum_{i=0}^{N-1} m_i X^i = \lfloor \Delta p \rceil \in \mathbb{Z}[X]/(X^N + 1)$. A common misconception is that, because $\boldsymbol{z}$ is real-valued, the polynomials p and m have only their first $N/2$ coefficients non-zero. If this were true, one could apply the SlotToCoeff operation to an encryption of the vector $\boldsymbol{p}_0 = (p_i)_{i \in [0, N/2)} \in \mathbb{R}^{N/2}$ to recover an encryption of $m = \lfloor \Delta p \rceil$, and hence of $\boldsymbol{z} = \tau_{N/2}(p)$.

This, however, is not the case. In general, both halves of the coefficient vector of p are non-zero. What holds instead is a *skew-symmetric relation* between the coefficients:

$$p_{N-i} = -p_i \quad \text{for all } i \in [1, N/2],$$

see Figure 1. In particular, this implies that $p_{N/2} = 0$. Thus, p is completely determined by its first $N/2$ coefficients, not because the remaining ones vanish, but because they are fixed by this symmetry. Equivalently, we can express p as follows:

$$p = p_0 + \sum_{i=1}^{N/2-1} p_i \, (X^i - X^{N-i}).$$

SCORE exploits this structure to recover p from only the first half of its coefficients. More precisely, we consider an encryption of the vector $\boldsymbol{p}_0 = (p_i)_{i \in [0, N/2)} \in \mathbb{R}^{N/2}$ containing the first $N/2$ coefficients of p. Starting from an encryption of $\boldsymbol{p}_0$, the SCORE operation constructs an encryption of the plaintext polynomial $m = \lfloor \Delta p \rceil$ and hence of the vector $\boldsymbol{z} = \tau_{N/2}(p)$.

This is done as follows. First, we apply the conventional SlotToCoeff operation onto the encryption of $\boldsymbol{p}_0$; this yields an encryption of the polynomial $m' = \sum_{i=0}^{N/2-1} m_i X^i$. While the first half of m' matches that of m, the second half is zero (because $\boldsymbol{p}_0$ has no imaginary part), in contrast to m. Therefore, the conventional SlotToCoeff alone is insufficient to fully recover m from $\boldsymbol{p}_0$. To fix this, we apply the conjugation automorphism $\mathsf{Conj} : X \mapsto X^{-1}$ homomorphically, resulting in a ciphertext which encrypts the polynomial $\mathsf{Conj}(m') = m_0 - \sum_{i=1}^{N/2-1} m_i X^{N-i}$. Homomorphically adding m' and $\mathsf{Conj}(m')$ yields an encryption of the following polynomial:

$$m' + \mathsf{Conj}(m') = 2m_0 + \sum_{i=1}^{N/2-1} m_i \cdot (X^i - X^{N-i}) = m + m_0.$$

This nearly recovers m, but includes an extra m_0 in the constant term. To eliminate this extra $m_0 = \lfloor \Delta p_0 \rceil$, we first multiply the encrypted coefficient vector $\boldsymbol{p}_0 = (p_i)_{i \in [0, N/2)}$ by the vector $(1/2, 1, \ldots, 1) \in \mathbb{R}^{N/2}$ before applying the above procedure. This ensures that the constant term is scaled correctly. Moreover, this multiplication can be embedded directly into the SlotToCoeff operation, thereby avoiding any additional homomorphic multiplication.

Experimental Results. We integrate SCORE into two bootstrapping algorithms: the conventional method [13] and the SPRU method [18]. Our experimental

results demonstrate that SCORE yields notable performance improvements. In the case of the conventional algorithm, we observe modest speedups exceeding 1.25×. For the SPRU algorithm, the gains are more pronounced, with speedups approaching 2×. Our implementations are publicly available online via GitHub.[1] We emphasize that our technique is not limited to these two algorithms. It applies to any bootstrapping procedure whose final step is a SlotToCoeff operation, as is also the case for PaCo [19]. In this sense, SCORE does not compete with existing bootstrapping algorithms, but rather complements them as a general optimization layer that can be integrated into a broad range of designs. The precise speedup achieved depends on the given algorithm, but the underlying source of improvement remains the same: only half of the polynomial coefficients need to be processed.

2 Notation

All intervals are assumed to be over the integers, and all logarithms are considered base 2. For a real number x, we write $\lfloor x \rceil$ for the nearest integer to x. This notation extends coefficientwise to polynomials. For an integer $q \geqslant 1$ and a real number x, we write $[x]_q$ for the non-negative reduction of x modulo q. We write $\mathrm{I} \in \mathbb{C}$ for the imaginary unit (satisfying $\mathrm{I}^2 = -1$). Given $z = a + b\mathrm{I} \in \mathbb{C}$, we denote its complex conjugate by $\overline{z} = a - b\mathrm{I}$ and its imaginary part by $\mathrm{Im}(z) = b$. All vectors in this paper are assumed to be column vectors, even when written in tuple notation. For two complex vectors $\boldsymbol{z} = (z_i)_{i\in[0,n)}$ and $\boldsymbol{w} = (w_i)_{i\in[0,n)}$, their entrywise (Hadamard) product is written as $\boldsymbol{z} \odot \boldsymbol{w} = (z_i w_i)_{i\in[0,n)}$. For $j \in \mathbb{Z}$, the rotation of $\boldsymbol{z}$ by j positions is denoted by $\mathsf{Rot}_j(\boldsymbol{z}) = (z_{[i+j]_n})_{i\in[0,n)}$. Given complex numbers $z_0, \ldots, z_{n-1} \in \mathbb{C}$, we denote by $\mathsf{Diag}(z_0, \ldots, z_{n-1}) \in \mathbb{C}^{n\times n}$ the diagonal matrix with the z_i on the diagonal. The identity matrix of size $n \times n$ is denoted by $\boldsymbol{I}_n$. Throughout this paper, we fix a power-of-two integer $N \geqslant 1$, called the *ring dimension* for the CKKS scheme. Unless stated otherwise, $n \geqslant 1$ denotes a strict divisor of N.

3 Background on CKKS

We now provide a summary of the main components of the CKKS scheme.

3.1 Encoding and Decoding

The CKKS scheme enables the encryption of vectors $\boldsymbol{z} \in \mathbb{C}^n$, where n is a strict divisor of N. This is accomplished by representing the vector $\boldsymbol{z}$ as a polynomial $m \in \mathbb{Z}[Y]/(Y^{2n}+1)$. The correspondence between vectors and polynomials is established via the ring isomorphism:

$$\tau_n : \frac{\mathbb{R}[Y]}{(Y^{2n}+1)} \to \mathbb{C}^n, \quad p \mapsto \left(p\left(\zeta_{4n}^{5^i}\right)\right)_{i\in[0,n)},$$

[1] https://github.com/se-tim/SCORE-Implementations-in-Go.git

where $\zeta_{4n} = \exp(2\pi \mathrm{I}/(4n))$ is the primitive $(4n)$-th root of unity with argument $\pi/(2n)$.

Given a large positive integer Δ, called the *scaling factor* for the CKKS scheme, we *encode* a complex vector $\boldsymbol{z} \in \mathbb{C}^n$ as $\mathsf{Ecd}(\boldsymbol{z}) = \lfloor \Delta p \rceil \in \mathbb{Z}[Y]/(Y^{2n}+1)$, where $p = \tau_n^{-1}(\boldsymbol{z}) \in \mathbb{R}[Y]/(Y^{2n}+1)$. By identifying $Y = X^{N/(2n)}$, we treat $\mathsf{Ecd}(\boldsymbol{z})$ as an element of $\mathbb{Z}[X]/(X^N+1)$. To *decode* a polynomial $m \in \mathbb{Z}[Y]/(Y^{2n}+1)$, it is enough to compute $\mathsf{Dcd}_n(m) = \tau_n(p) \in \mathbb{C}^n$ for $p = (1/\Delta) \cdot m$. For any two vectors $\boldsymbol{z}_0, \boldsymbol{z}_1 \in \mathbb{C}^n$, we have $\mathsf{Ecd}(\boldsymbol{z}_0) + \mathsf{Ecd}(\boldsymbol{z}_1) \approx \mathsf{Ecd}(\boldsymbol{z}_0 + \boldsymbol{z}_1)$ (the approximation is due to rounding). If we define $m_0 \times m_1 = \lfloor (1/\Delta) \cdot m_0 m_1 \rceil$ for $m_0, m_1 \in \mathbb{Z}[X]/(X^N+1)$, then also $\mathsf{Ecd}(\boldsymbol{z}_0) \times \mathsf{Ecd}(\boldsymbol{z}_1) \approx \mathsf{Ecd}(\boldsymbol{z}_0 \odot \boldsymbol{z}_1)$.

3.2 Ciphertexts

A *CKKS ciphertext* encrypting a polynomial $m \in \mathbb{Z}[X]/(X^N+1)$ under a modulus q is a tuple $\mathbf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1)$ with $\mathsf{ct}_0, \mathsf{ct}_1 \in \mathbb{Z}[X]/(X^N+1)$, satisfying the decryption equation $\mathsf{ct}_0 + s \cdot \mathsf{ct}_1 = m + e \bmod q$, with $s \in \mathbb{Z}[X]/(X^N+1)$ being the secret key and $e \in \mathbb{Z}[X]/(X^N+1)$ a small error. We denote this as $\mathbf{ct} = \mathsf{Enc}(m)$, noting that decryption yields only an approximate recovery of m. When $m \approx \mathsf{Ecd}(\boldsymbol{z})$ for some $\boldsymbol{z} \in \mathbb{C}^n$, we may 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 $\boldsymbol{z}$.

In the CKKS scheme, ciphertexts are usually considered under a modulus belonging to a sequence of moduli $q_0, \ldots, q_L$, where each modulus is given by $q_\ell = q_0 \cdot \Delta^\ell$. Here, q_0 denotes the *base modulus* and Δ is the scaling factor from before. A ciphertext encrypted under the modulus q_ℓ is said to be at *level* ℓ. Operations involving homomorphic multiplication typically reduce a ciphertext's level. In practical implementations such as Lattigo [32], the residue number system (RNS) variant of CKKS is typically used, where ciphertext moduli are represented as products of distinct primes to improve efficiency [12]. For clarity in our discussions, we will continue using the standard (non-RNS) variant of CKKS, although our implementations utilize the RNS variant implemented in Lattigo, see Section 5.

3.3 Elementary Operations

We summarize the elementary homomorphic operations in CKKS that are relevant to this work. For details on their implementation, see [13,15]. Consider ciphertexts $\mathbf{ct} = \mathsf{Enc}^*(\boldsymbol{z})$ and $\mathbf{ct}_i = \mathsf{Enc}^*(\boldsymbol{z}_i)$ for $i \in [0, 2)$, where $\boldsymbol{z}, \boldsymbol{z}_i \in \mathbb{C}^n$. Let $\boldsymbol{w} \in \mathbb{C}^n$ be a plaintext vector and $\lambda \in \mathbb{C}$ a scalar. Provided the required evaluation keys, the operations in Table 1 can be executed homomorphically.

We note that in Table 1, all homomorphic multiplications are assumed to include the rescaling step, so that the resulting encrypted vectors do not accumulate an extra factor of Δ. Throughout, the symbol $\times$ denotes a multiplication with an implicit rescaling.

Also, observe that if $p = \tau_n^{-1}(\boldsymbol{z}) \in \mathbb{R}[Y]/(Y^{2n}+1)$, then $\mathsf{Conj}(p) = \tau_n^{-1}(\overline{\boldsymbol{z}})$, where $\mathsf{Conj}(p) = p(Y^{-1})$. Therefore, the ciphertext $\mathsf{Conj}(\mathbf{ct})$ encrypts the polynomial $\lfloor \Delta \mathsf{Conj}(p) \rceil \in \mathbb{Z}[Y]/(Y^{2n}+1)$. Naturally, homomorphic conjugation allows

Table 1. Some homomorphic operations in CKKS.

Operation	Notation	Encrypted vector	Consumed levels
Rescaling	$\mathsf{RS}(\mathbf{ct})$	$(1/\Delta) \cdot \boldsymbol{z}$	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

to homomorphically extract the real and imaginary parts of encrypted complex vectors.

3.4 Matrix–Ciphertext Multiplication

Given a matrix $\boldsymbol{A} = (A_{i,j})_{i,j\in[0,n)} \in \mathbb{C}^{n\times n}$, we define its *j-th diagonal* for $j \in [0, n)$ as $\mathsf{Diag}_j(\boldsymbol{A}) = (A_{i,[i+j]_n})_{i\in[0,n)} \in \mathbb{C}^n$. For any vector $\boldsymbol{z} \in \mathbb{C}^n$, we can then compute $\boldsymbol{A} \cdot \boldsymbol{z}$ using the *diagonal method* [25]:

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

This decomposition enables matrix–ciphertext multiplication. Specifically, if $\mathsf{Ecd}(\boldsymbol{A}) = (\mathsf{Ecd}(\mathsf{Diag}_j(\boldsymbol{A})))_{j\in[0,n)}$ encodes the diagonals of $\boldsymbol{A}$, and $\mathbf{ct} = \mathsf{Enc}^*(\boldsymbol{z})$ is a ciphertext encrypting $\boldsymbol{z}$, then:

$$\mathsf{Ecd}(\boldsymbol{A}) \times \mathbf{ct} = \sum_{j=0}^{n-1} \mathsf{Ecd}(\mathsf{Diag}_j(\boldsymbol{A})) \times \mathsf{Rot}_j(\mathbf{ct})$$

is a ciphertext encrypting $\boldsymbol{A}\cdot\boldsymbol{z}$. This operation consumes only one ciphertext level, but generally requires $\mathcal{O}(n)$ rotations and plaintext–ciphertext multiplications. However, matrix–ciphertext multiplications become particularly efficient in the case where the involved matrix $\boldsymbol{A}$ is *diagonally sparse*, meaning that most of its diagonals $\mathsf{Diag}_j(\boldsymbol{A})$ are zero. In that case, only a small number of rotations and plaintext–ciphertext multiplications are required. The method can also be subject to a *baby-step giant-step* optimization, see for instance [13]. In Section A, we describe how the matrix–ciphertext multiplication can be implemented more efficiently for some matrices by trading additional multiplicative levels for reduced computational cost.

3.5 Trace Operation

Let $n \geqslant B$ be two strict divisors of N, and let $\boldsymbol{z} \in \mathbb{C}^n$. We partition $\boldsymbol{z}$ into n/B blocks of length B, namely $\boldsymbol{z} = (\boldsymbol{z}_i)_{i\in[0,n/B)}$ with $\boldsymbol{z}_i \in \mathbb{C}^B$. The *trace operation* $\mathsf{Tr}_{n\to B}(\boldsymbol{z})$ is defined as the sum of these blocks: $\mathsf{Tr}_{n\to B}(\boldsymbol{z}) = \sum_{i=0}^{n/B-1} \boldsymbol{z}_i \in \mathbb{C}^B$. It

is possible to apply it homomorphically to a ciphertext $\mathbf{ct}$, yielding a ciphertext $\mathsf{Tr}_{n\to B}(\mathbf{ct})$. It requires $\log(n/B)$ homomorphic rotations and does not consume any levels. See Section B for more details.

3.6 The CoeffToSlot and SlotToCoeff Operations

Let $\boldsymbol{z} \in \mathbb{C}^n$ be a complex vector with associated 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 $\boldsymbol{p}_1 = (p_{i+n})_{i\in[0,n)} \in \mathbb{R}^n$. Then, the vector $\boldsymbol{z}$ can be recovered from $\boldsymbol{p}_0$ and $\boldsymbol{p}_1$ as follows [13]:

$$\boldsymbol{z} = \boldsymbol{U}_n(\boldsymbol{p}_0 + \mathrm{I}\cdot\boldsymbol{p}_1) \tag{1}$$

for the Vandermonde matrix $\boldsymbol{U}_n \in \mathbb{C}^{n\times n}$ whose entry in row $i \in [0,n)$ and column $j \in [0,n)$ is given by $\zeta_{4n}^{5^i j}$. Homomorphically, if $\mathbf{ct} = \mathsf{Enc}^*(\boldsymbol{p}_0 + \mathrm{I}\cdot\boldsymbol{p}_1)$ is a ciphertext encrypting the coefficients of p in its slots (in both the real and imaginary parts), we define the *SlotToCoeff operation* as:

$$\mathsf{SlotToCoeff}_n(\mathbf{ct}) = \mathsf{Ecd}(\boldsymbol{U}_n) \times \mathbf{ct};$$

this yields an encryption of the vector $\boldsymbol{z}$ according to (1). The SlotToCoeff operation thus transforms an encryption where the *slots* contain the coefficients of p into an encryption of the polynomial $m = \lfloor \Delta p \rceil$, whose *coefficients* correspond to those of p (scaled by Δ and rounded).

For the reverse transformation, we make use of the inverse of $\boldsymbol{U}_n$, namely $\boldsymbol{U}_n^{-1} = (1/n)\cdot\overline{\boldsymbol{U}}_n^{\mathsf{T}}$ [13]. From (1), we then have:

$$\boldsymbol{p}_0 + \mathrm{I}\cdot\boldsymbol{p}_1 = \tfrac{1}{n}\overline{\boldsymbol{U}}_n^{\mathsf{T}}\cdot\boldsymbol{z}.$$

Homomorphically, given a ciphertext $\mathbf{ct} = \mathsf{Enc}^*(\boldsymbol{z})$, the *CoeffToSlot operation* is defined as:

$$\mathsf{CoeffToSlot}_n(\mathbf{ct}) = \mathsf{Ecd}\left(\tfrac{1}{n}\overline{\boldsymbol{U}}_n^{\mathsf{T}}\right) \times \mathbf{ct},$$

which yields an encryption of the vector $\boldsymbol{p}_0 + \mathrm{I}\cdot\boldsymbol{p}_1$. The real and imaginary parts can then be extracted homomorphically to obtain separate encryptions of both coefficient vectors $\boldsymbol{p}_0$ and $\boldsymbol{p}_1$. Note that for $m = \sum_{i=0}^{2n-1} m_i Y^i = \lfloor \Delta p \rceil$, an encryption of $\boldsymbol{p}_0 + \mathrm{I}\cdot\boldsymbol{p}_1$ can also be seen as an encryption of the vector $(1/\Delta)\cdot(\boldsymbol{m}_0 + \mathrm{I}\cdot\boldsymbol{m}_1)$ with $\boldsymbol{m}_0 = (m_i)_{i\in[0,n)} \in \mathbb{Z}^n$ and $\boldsymbol{m}_1 = (m_{i+n})_{i\in[0,n)} \in \mathbb{Z}^n$.

The CoeffToSlot and SlotToCoeff operations each require $\mathcal{O}(n)$ homomorphic multiplications and homomorphic rotations. One can use a Cooley-Tukey decomposition [26,17] to reduce the computational cost of the CoeffToSlot and SlotToCoeff operations, though this increases the number of consumed ciphertext levels. More details are provided in Section D.

4 Bootstrapping Encrypted Real Vectors

4.1 Introducing SCORE

We introduce SCORE, a variant of the SlotToCoeff operation specifically designed for encrypted real vectors in the CKKS scheme. SCORE is a homomorphic

operation applied to ciphertexts. We fix a real vector $\boldsymbol{z} \in \mathbb{R}^n$ with associated polynomials $p = \tau_n^{-1}(\boldsymbol{z}) \in \mathbb{R}[Y]/(Y^{2n}+1)$ and $m = \lfloor \Delta p \rceil \in \mathbb{Z}[Y]/(Y^{2n}+1)$; an encryption of m is therefore also an encryption of $\boldsymbol{z}$. The SCORE operation has the following effect:

$$\mathsf{SCORE}_n(\mathsf{Enc}^*(\boldsymbol{p}_0)) = \mathsf{Enc}^*(\boldsymbol{z}),$$

where $\boldsymbol{p}_0 = (p_i)_{i\in[0,n)} \in \mathbb{R}^n$ is the coefficient vector of the first n coefficients of the polynomial $p = \sum_{i=0}^{2n-1} p_i Y^i$.

Skew-Symmetric Relation. SCORE is based on the following observation, depicted in Figure 1:

Proposition 1. *Consider the polynomial $p = \tau_n^{-1}(\boldsymbol{z}) \in \mathbb{R}[Y]/(Y^{2n}+1)$ for a real vector $\boldsymbol{z} \in \mathbb{R}^n$. If we write $p = \sum_{i=0}^{2n-1} p_i Y^i$, then its coefficients satisfy $p_{2n-i} = -p_i$ for every $i \in [1,n]$. In other words, it holds that $p = p_0 + \sum_{i=1}^{n-1} p_i \cdot (Y^i - Y^{2n-i})$.*

Proof. Since $\boldsymbol{z}$ is real-valued, we have $\overline{\boldsymbol{z}} = \boldsymbol{z}$. Therefore $\mathsf{Conj}(p) = p$, which means $p_0 - \sum_{i=1}^{2n-1} p_{2n-i} Y^i = p_0 + \sum_{i=1}^{2n-1} p_i Y^i$. Comparing coefficients of Y^i for $i \in [1, 2n)$ yields $-p_{2n-i} = p_i$. □

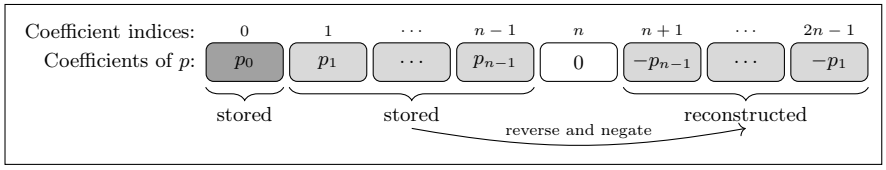


Fig. 1. Structure of $p = \sum_{i=0}^{2n-1} p_i Y^i = \tau_n^{-1}(\boldsymbol{z}) \in \mathbb{R}[Y]/(Y^{2n}+1)$ for $\boldsymbol{z} \in \mathbb{R}^n$.

Altered SlotToCoeff Operation. Recall that the conventional $\mathsf{SlotToCoeff}$ operation consists of a homomorphic multiplication by the matrix $\boldsymbol{U}_n$. In our approach, we modify this step by introducing a new matrix $\boldsymbol{U}'_n$, which is derived from $\boldsymbol{U}_n$ by scaling its first column by $1/2$. This means that left-multiplying a vector by $\boldsymbol{U}'_n$ is equivalent to halving its first entry, followed by a left-multiplication by $\boldsymbol{U}_n$. As a result, for the polynomial $m' = \lfloor \Delta p' \rceil \in \mathbb{Z}[Y]/(Y^{2n}+1)$ with $p' = p_0/2 + \sum_{i=1}^{n-1} p_i Y^i$, we have $\mathsf{Ecd}(\boldsymbol{U}'_n) \times \mathsf{Enc}^*(\boldsymbol{p}_0) = \mathsf{Enc}(m')$.

Homomorphic Conjugation. By construction, the polynomial p' satisfies $p' + \mathsf{Conj}(p') = p_0 + \sum_{i=1}^{n-1} p_i(Y^i - Y^{2n-i}) = p$, so that also $m' + \mathsf{Conj}(m') \approx m$ with $m = \lfloor \Delta p \rceil$ (the approximation is due to rounding). Homomorphically, this yields $\mathsf{Enc}(m') + \mathsf{Conj}(\mathsf{Enc}(m')) = \mathsf{Enc}(m) = \mathsf{Enc}^*(\boldsymbol{z})$.

The Algorithm. The entire SCORE procedure is concise and summarized in Algorithm 1. We recall that $\boldsymbol{z} \in \mathbb{R}^n$ is a real vector with associated polynomial $p = \tau_n^{-1}(\boldsymbol{z})$; the vector $\boldsymbol{p}_0 = (p_i)_{i \in [0,n)} \in \mathbb{R}^n$ contains the first n coefficients of $p = \sum_{i=0}^{2n-1} p_i Y^i$.

Algorithm 1: SCORE

Input: A ciphertext $\mathbf{ct} = \mathsf{Enc}^*(\boldsymbol{p}_0)$
Input: The power-of-two number of slots $n \in [1, N/2]$
Output: An encryption $\mathbf{ct}' = \mathsf{Enc}^*(\boldsymbol{z})$ of the real vector $\boldsymbol{z} \in \mathbb{R}^n$
1: $\mathbf{ct}' \leftarrow \mathsf{Ecd}(\boldsymbol{U}_n') \times \mathbf{ct}$ ▷ $\boldsymbol{U}_n' = \boldsymbol{U}_n \cdot \mathsf{Diag}(1/2, 1, \ldots, 1) \in \mathbb{C}^{n \times n}$
2: $\mathbf{ct}' \leftarrow \mathbf{ct}' + \mathsf{Conj}(\mathbf{ct}')$
3: **return** $\mathbf{ct}'$

SCORE can be accelerated by incorporating the Cooley-Tukey decomposition [26,17], which reduces the computational cost of the matrix–ciphertext multiplication present in SCORE at the expense of consuming additional ciphertext levels; further details can be found in Section D.

4.2 Application to Conventional Bootstrapping

We describe how to incorporate SCORE into the conventional bootstrapping algorithm for CKKS, originally proposed in [13]; henceforth, we refer to this algorithm simply as BOOT. Since its introduction, this algorithm has undergone numerous refinements and optimizations, *e.g.*, [5,6,10]. Our discussion is independent of these optimizations. The algorithm we study applies the CoeffToSlot operation before the SlotToCoeff operation, and is hence referred to as the *CoeffToSlot-first variant*. A *SlotToCoeff-first variant* exists as well, see for instance [2]. The main efficiency improvement of integrating SCORE comes from applying the (approximated) modular reduction function homomorphically to only half as many values compared to the conventional BOOT algorithm since the standard SlotToCoeff cannot natively recover the skew-symmetrically determined (redundant) coefficients of p from only its first half.

We refer to this modified BOOT algorithm as the *R-BOOT algorithm*, since it bootstraps encrypted *real* vectors. The complete R-BOOT algorithm is summarized in Algorithm 2. Note that the Cooley-Tukey decomposition can directly be incorporated into the R-BOOT algorithm, see Section D. We provide practical results in Section 5.

Let us briefly explain why R-BOOT is correct:

- As input, we have a ciphertext $\mathbf{ct} = \mathsf{Enc}^*(\boldsymbol{z})$ encrypting a real vector $\boldsymbol{z} \in \mathbb{R}^n$. Letting $p = \tau_n^{-1}(\boldsymbol{z})$ and $m = \sum_{i=0}^{2n-1} m_i Y^i = \lfloor \Delta p \rceil \in \mathbb{Z}[Y]/(Y^{2n}+1)$, we can also write $\mathbf{ct} = \mathsf{Enc}(m)$. We assume that the ciphertext $\mathbf{ct}$ is at modulus q_0.
- We first reinterpret $\mathbf{ct}$ as a ciphertext at the highest level $\ell = L$, yielding a ciphertext $\mathsf{ModRaise}(\mathbf{ct}) = \mathsf{Enc}(m + q_0 J)$ for some small $J \in \mathbb{Z}[X]/(X^N+1)$.

Algorithm 2: R-BOOT (BOOT algorithm for encrypted real vectors)

Input: An encryption $\mathbf{ct}$ of a real vector $\boldsymbol{z} \in \mathbb{R}^n$
Output: A bootstrapped encryption $\mathbf{ct}_{\mathsf{boot}}$ of $\boldsymbol{z}$

1: $\mathbf{ct}_{\mathsf{boot}} \leftarrow \mathsf{ModRaise}(\mathbf{ct})$ ▷ View $\mathbf{ct}$ as a ciphertext at level $\ell = L$
2: $\mathbf{ct}_{\mathsf{boot}} \leftarrow \mathsf{Tr}_{N/2\to n}(\mathbf{ct})$
3: $\mathbf{ct}_{\mathsf{boot}} \leftarrow (n/N) \cdot \mathsf{CoeffToSlot}_n(\mathbf{ct})$ ▷ Incorporate n/N into CoeffToSlot
4: $\mathbf{ct}_{\mathsf{boot}} \leftarrow \mathbf{ct}_{\mathsf{boot}} + \mathsf{Conj}(\mathbf{ct}_{\mathsf{boot}})$ ▷ Compute twice the real part
5: $\mathbf{ct}_{\mathsf{boot}} \leftarrow \mathsf{EvalMod}_{q_0/\Delta}(\mathbf{ct}_{\mathsf{boot}})$ ▷ Reduce modulo q_0/Δ
6: $\mathbf{ct}_{\mathsf{boot}} \leftarrow \mathsf{SCORE}_n(\mathbf{ct}_{\mathsf{boot}})$ ▷ Apply Algorithm 1
7: **return** $\mathbf{ct}_{\mathsf{boot}}$

- To bring the plaintext polynomial $m + q_0 J$ into the subring $\mathbb{Z}[Y]/(Y^{2n}+1)$, we apply the trace operation from $N/2$ to n, yielding an encryption of the polynomial $(N/(2n)) \cdot (m + q_0 J')$ for some $J' = \sum_{i=0}^{2n-1} J'_i Y^i \in \mathbb{Z}[Y]/(Y^{2n}+1)$.
- Next, we apply the CoeffToSlot operation with the incorporated factor n/N. This yields an encryption of a vector whose real part is $(1/(2\Delta)) \cdot (\boldsymbol{m}_0 + q_0 \boldsymbol{J}'_0)$ and whose imaginary part is $(1/(2\Delta)) \cdot (\boldsymbol{m}_1 + q_0 \boldsymbol{J}'_1)$, where $\boldsymbol{m}_0 = (m_i)_{i\in[0,n)}$, $\boldsymbol{m}_1 = (m_{i+n})_{i\in[0,n)}$ and $\boldsymbol{J}'_0 = (J'_i)_{i\in[0,n)}$, $\boldsymbol{J}'_1 = (J'_{i+n})_{i\in[0,n)}$.
- Since only the first n coefficients of m are relevant (see Figure 1), we are only interested in the real part of the result. Therefore, we add the resulting ciphertext to its homomorphic conjugate, which gives an encryption of the vector $(1/\Delta) \cdot (\boldsymbol{m}_0 + q_0 \boldsymbol{J}'_0)$. (Without SCORE, we would also need to keep track of the imaginary part and apply the upcoming step to it as well.)
- Applying a homomorphic reduction modulo q_0/Δ, the so-called *EvalMod*$_{q_0/\Delta}$ *operation* (typically based on the approximation of trigonometric functions), results in an encryption of the vector $(1/\Delta) \cdot \boldsymbol{m}_0 \approx \boldsymbol{p}_0$ with $\boldsymbol{p}_0 = (p_i)_{i\in[0,n)}$ containing the first n coefficients of $p = \tau_n^{-1}(\boldsymbol{z})$.
- The final step of the algorithm is to apply the SCORE operation. As discussed above, this yields an encryption of the real vector $\boldsymbol{z}$, which is the desired output.

4.3 Application to SPRU Bootstrapping

Similarly, SCORE can be incorporated into the SPRU bootstrapping algorithm [18], which is tailored for CKKS ciphertexts encrypting a small number of slots. This restriction arises because SPRU uses the available slots to parallelize the homomorphic evaluation of the decryption circuit; supporting more encrypted slots is possible, but comes at a substantial efficiency cost. The small-slot regime is particularly relevant in machine-learning applications [23,21], where the encrypted data is often real-valued. Integrating SCORE into SPRU is therefore especially advantageous such settings.

Let us briefly explain how the SPRU algorithm works. The central idea of SPRU is to homomorphically evaluate the decryption equation directly over the complex unit circle. Starting from a ciphertext $\mathbf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1)$ satisfying $\mathsf{ct}_0 + \mathsf{ct}_1 \cdot s = m \bmod q_0$, the algorithm extracts each coefficient m_i of the

plaintext polynomial by isolating the corresponding coefficient in the decryption equation. Instead of recovering m_i directly, SPRU embeds the additive group $\mathbb{Z}_{q_0}$ into the complex unit circle via the map $x \mapsto \zeta_{q_0}^x$ with $\zeta_{q_0} = \exp(2\pi \mathrm{I}/q_0)$, thereby transforming the modular linear relation into a multiplicative identity in $\mathbb{C}$. At this stage, the coefficients m_i appear in the slots of the ciphertext encoded in the exponent, that is, in the form $\zeta_{q_0}^{m_i}$. Using the *small-angle approximation*, we have $\mathrm{Im}(\zeta_{q_0}^{m_i}) = \sin(2\pi m_i/q_0) \approx 2\pi m_i/q_0$ under the assumption $|m_i| \ll q_0$. Hence, the coefficients m_i can be recovered by homomorphically extracting the imaginary part of the ciphertext and applying an appropriate rescaling. The algorithm then concludes with a final SlotToCoeff operation.

Using a structured secret key, the overall complexity of SPRU is $\mathcal{O}(n)$ homomorphic operations, where n denotes the number of slots in the input ciphertext. When the ciphertext encrypts a real vector, however, only the first half of the coefficients of m are required. In this case, SPRU can be executed on half as many coefficients and the final SlotToCoeff step can be replaced by SCORE, resulting in an approximate $2\times$ speedup.

We provide a detailed derivation of the modified version in the appendix, see Section C. The resulting algorithm is referred to as the *R-SPRU algorithm*. It supports bootstrapping encrypted real vectors $\boldsymbol{z} \in \mathbb{R}^n$ with a power-of-two number of slots $n \in [1, N/(2h)]$. Practical results are given in Section 5.

5 Practical Results

5.1 Implementation Details

Our proof-of-concept implementations were developed in Lattigo [32], including both bootstrapping algorithms BOOT and SPRU, as well as their variants utilizing SCORE, namely R-BOOT and R-SPRU. To promote transparency and reproducibility, our implementations are publicly available on GitHub.[2]

5.2 Parameter Sets

We provide two distinct parameter sets: one for the BOOT/R-BOOT context, and another for the SPRU/R-SPRU context, as summarized in Table 2.

We now provide further explanations for these parameter sets:

- The ring dimension is denoted by N. For SPRU/R-SPRU, a smaller N is used due to lower multiplicative depth requirements.
- The Hamming weight of the secret key $s \in \mathbb{Z}[X]/(X^N + 1)$ is h. In BOOT/R-BOOT, the key is ternary; in SPRU/R-SPRU, it is binary and meets the structural constraints described in Section C. For BOOT/R-BOOT, the second value for h in Table 2 refers to the Hamming weight of the *ephemeral secret key* [6].
- The parameter q_0 denotes the smallest ciphertext modulus, which is equal to the modulus of a ciphertext prior to bootstrapping.

[2] `https://github.com/se-tim/SCORE-Implementations-in-Go.git`

Table 2. Parameter sets for the two bootstrapping contexts.

Algorithm	BOOT/R-BOOT	SPRU/R-SPRU
$\log N$	16	15
h	192 and 32	64
$\log q_0$	55	53
$\log \Delta$	40	40
$\log(PQ/q')$	$3 \cdot 39 + 8 \cdot 60 + 4 \cdot 56 + 3 \cdot 61$	$39 + 7 \cdot 55 + 61$
$\log(PQ)$	$\leqslant 1\,259$	578
Precision	$\geqslant 27$ bits	$\geqslant 25$ bits

- The scaling factor (outside bootstrapping) is denoted by Δ.
- The post-bootstrapping ciphertext modulus is denoted by q'. For BOOT/R-BOOT, q' is chosen as $q' \approx q_0 \cdot \Delta^\ell$ with varying $\ell \in [1, 5]$; for SPRU/R-SPRU, we take a single $q' \approx q_0 \cdot \Delta$.
- Q is the largest ciphertext modulus encountered during bootstrapping, and PQ is the modulus used for evaluation keys.
- The value $\log(PQ/q')$ indicates the total number of bits allocated for bootstrapping. For BOOT/R-BOOT, the terms in Table 2 for $\log(PQ/q')$, like $3 \cdot 39$, $8 \cdot 60$, *etc.*, correspond to the modulus size requirements for SlotToCoeff, EvalMod, CoeffToSlot, and P. Both SlotToCoeff and CoeffToSlot use the Cooley-Tukey decomposition [26,17] with radices 2^5 and 2^4, respectively (see Section D). For SPRU/R-SPRU, the terms for $\log(PQ/q')$ correspond to SlotToCoeff, the product operator with two plaintext–ciphertext multiplications, and P. A Cooley-Tukey decomposition is not used for SPRU/R-SPRU due to a lower computational cost of SCORE when dealing with fewer slots, but its integration is still explained in Section D for completeness.
- The bound on $\log(PQ)$ in Table 2 ensures at least 128 bits of security, based on standard lattice estimators such as [1], considering attacks such as primal uSVP, primal hybrid, dual, and dual hybrid.
- The precision row indicates the average number of bits preserved in the real part of the bootstrapped encrypted vectors.

5.3 Benchmarks

We now summarize our experimental findings, demonstrating the performance gains achieved by integrating SCORE into the two chosen bootstrapping algorithms for encrypted real vectors. Results are presented in Table 3 and Table 4, and both are represented as comparative graphs in Figure 2. All timings have been obtained by averaging over 30 runs on a 2023 MacBook Air with an Apple M2 chip. For reproducibility, we decided to only use a single thread.

For the SPRU/R-SPRU algorithms, we benchmark varying the number of slots from $n = 2^1$ up to $n = 2^7$. On the other hand, for the BOOT/R-BOOT algorithms, we only consider the case of full packing, namely $n = N/2$ slots. This is because, for fewer slots, the conventional BOOT algorithm can be optimized by repacking the real and imaginary parts after the CoeffToSlot operation into a single ciphertext, allowing the EvalMod operation to be applied only once,

just as in R-BOOT. Therefore, the main advantage of SCORE in BOOT, namely reducing the number of EvalMod evaluations from two to one, does not apply when bootstrapping ciphertexts with fewer slots. Accordingly, instead of varying the number of slots for BOOT/R-BOOT, we vary the post-bootstrapping ciphertext modulus from $q' \approx 2^{95}$ to $q' \approx 2^{255}$.

Conventional Bootstrapping. Table 3 and Figure 2 show comparative timings for R-BOOT versus BOOT, using the parameter set from Table 2. We vary the post-bootstrapping ciphertext modulus q' while keeping all other parameters fixed. The data indicate that R-BOOT consistently achieves faster runtimes than BOOT, with speedups of approximately 1.25×. This improvement is due to R-BOOT applying the EvalMod operation to only one encrypted vector, while BOOT processes two. The other steps in both algorithms are similar in cost, resulting in a moderate overall speedup.

SPRU Bootstrapping. Similarly, we show in Table 4 and Figure 2 comparative timings for R-SPRU versus SPRU, using the parameter set from Table 2. The results indicate that R-SPRU consistently achieves better performance than SPRU. The speedup for R-SPRU grows with the number of slots n, approaching a factor of 2× for large n. This is due to the fact that the main computational cost in SPRU for larger n stems from $2n$ plaintext–ciphertext multiplications, while R-SPRU reduces this to n such multiplications.

Table 3. Average runtime of BOOT and R-BOOT for $N/2$ slots with the parameter set from Table 2, to bootstrap a ciphertext from modulus $q_0 \approx 2^{55}$ to modulus $q' > q_0$.

$\log q'$	95	135	175	215	255
BOOT	5.83 s	6.56 s	7.28 s	8.15 s	8.99 s
R-BOOT	4.70 s	5.32 s	5.91 s	6.57 s	7.13 s

Table 4. Average runtime of SPRU and R-SPRU for n slots with the parameter set from Table 2, to bootstrap a ciphertext from modulus $q_0 \approx 2^{55}$ to modulus $q' \approx 2^{95}$.

$\log n$	1	2	3	4	5	6	7
SPRU	0.79 s	0.81 s	0.91 s	1.12 s	1.61 s	2.63 s	4.79 s
R-SPRU	0.63 s	0.66 s	0.71 s	0.80 s	1.03 s	1.52 s	2.60 s

6 Conclusion

We introduced SCORE, a variant of the SlotToCoeff operation for the CKKS scheme, designed for bootstrapping encrypted real vectors. By using the skew-symmetric structure of the plaintext polynomials encoding real vectors, SCORE

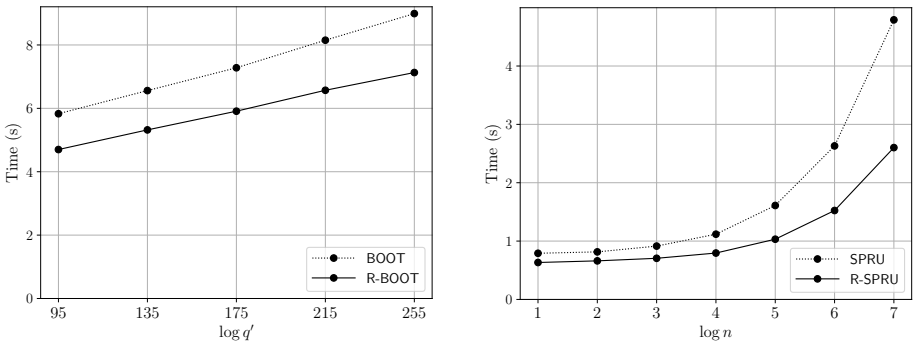


Fig. 2. Plots corresponding to the timings from Table 3 and Table 4.

enables bootstrapping algorithms to process only half of the coefficients, thereby reducing computational overhead. Our experiments demonstrate that integrating SCORE yields notable performance gains, achieving speedups approaching $2\times$ in some contexts.

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

Disclosure of Interests. The author has 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 (2025)
4. Bae, Y., Kim, J., Stehlé, D., Suvanto, E.: Bootstrapping Small Integers with CKKS. In: Advances in Cryptology – ASIACRYPT 2024. Springer (2025)
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.: (Leveled) Fully Homomorphic Encryption without Bootstrapping. In: ITCS 2012: Innovations in Theoretical Computer Science. ACM (2012)
9. Chang, R.I., Wang, C.H., Chang, Y.T., Wei, L.C.: Approximate Homomorphic Encryption for MLaaS by CKKS with Operation-Error-Bound. Computers, Materials and Continua **85** (2025)
10. Chen, H., Chillotti, I., Song, Y.: Improved Bootstrapping for Approximate Homomorphic Encryption. In: Advances in Cryptology – EUROCRYPT 2019. Springer (2019)
11. Chen, H., Dai, W., Kim, M., Song, Y.: Efficient Homomorphic Conversion Between (Ring) LWE Ciphertexts. In: Applied Cryptography and Network Security. Springer (2021)
12. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A Full RNS Variant of Approximate Homomorphic Encryption. In: Selected Areas in Cryptography – SAC 2018. Springer (2018)
13. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: Bootstrapping for Approximate Homomorphic Encryption. In: Advances in Cryptology – EUROCRYPT 2018. Springer (2018)
14. Cheon, J.H., Hanrot, G., Kim, J., Stehlé, D.: SHIP: A Shallow and Highly Parallelizable CKKS Bootstrapping Algorithm. In: Advances in Cryptology – EUROCRYPT 2025. Springer (2025)
15. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic Encryption for Arithmetic of Approximate Numbers. In: Advances in Cryptology – ASIACRYPT 2017. Springer (2017)
16. 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)
17. Cooley, J.W., Tukey, J.: An Algorithm for the Machine Calculation of Complex Fourier Series. Mathematics of Computation **19** (1965)
18. Coron, J.S., Köstler, R.: Low-Latency Bootstrapping for CKKS using Roots of Unity (2025), `https://eprint.iacr.org/2025/651`
19. Coron, J.S., Seuré, T.: PaCo: Bootstrapping for CKKS via Partial CoeffToSlot. In: Advances in Cryptology – ASIACRYPT 2025. Springer (2026)
20. Ducas, L., Micciancio, D.: FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In: Advances in Cryptology – EUROCRYPT 2015. Springer (2015)
21. Ebel, A., Garimella, K., Reagen, B.: Orion: A Fully Homomorphic Encryption Framework for Deep Learning. In: Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. ACM (2025)
22. Fan, J., Vercauteren, F.: Somewhat Practical Fully Homomorphic Encryption (2012), `https://eprint.iacr.org/2012/144`
23. Garimella, K., Ebel, A., De Micheli, G., Reagen, B.: HE-LRM: Encrypted Deep Learning Recommendation Models using Fully Homomorphic Encryption (2025), `https://arxiv.org/abs/2506.18150v1`
24. Gentry, C.: Fully Homomorphic Encryption Using Ideal Lattices. In: Proceedings of the Annual ACM Symposium on Theory of Computing. ACM (2009)
25. Halevi, S., Shoup, V.: Algorithms in HElib. In: Advances in Cryptology – CRYPTO 2014. Springer (2014)
26. Han, K., Hhan, M., Cheon, J.H.: Improved Homomorphic Discrete Fourier Transforms and FHE Bootstrapping. IEEE Access **7** (2019)

27. 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)
28. Kim, D., Song, Y.: Approximate Homomorphic Encryption over the Conjugate-Invariant Ring. In: Information Security and Cryptology – ICISC 2018. Springer (2019)
29. Lyubashevsky, V., Peikert, C., Regev, O.: On Ideal Lattices and Learning with Errors over Rings. In: Advances in Cryptology – EUROCRYPT 2010. Springer (2010)
30. Rahulamathavan, Y.: Privacy-Preserving Similarity Calculation of Speaker Features Using Fully Homomorphic Encryption (2022), `https://arxiv.org/abs/2202.07994`
31. Stehlé, D., Steinfeld, R., Tanaka, K., Xagawa, K.: Efficient Public Key Encryption Based on Ideal Lattices. In: Advances in Cryptology – ASIACRYPT 2009. Springer (2009)
32. Tune Insight: Lattigo v6 (2024), `https://github.com/tuneinsight/lattigo`

A Details on Matrix–Ciphertext Multiplication

Some matrices $\boldsymbol{A}$ can be decomposed as $\boldsymbol{A} = \boldsymbol{D}_{k-1} \cdots \boldsymbol{D}_0 = \prod_{\ell=0}^{k-1} \boldsymbol{D}_\ell$ for a small k, where each $\boldsymbol{D}_\ell$ is diagonally sparse. Homomorphic multiplication of $\mathbf{ct}$ by $\boldsymbol{A}$ can then be efficiently performed as $\mathsf{Ecd}(\boldsymbol{D}_{k-1}) \times \cdots \times \mathsf{Ecd}(\boldsymbol{D}_0) \times \mathbf{ct}$ (from right to left). The downside of this approach is that this consumes k multiplicative levels. To reduce level consumption, we can introduce a *grouping parameter* $g \in [1, k]$, combining the $\boldsymbol{D}_\ell$ into blocks of size g:

$$\prod_{\ell=0}^{k-1} \boldsymbol{D}_\ell = \prod_{i=0}^{\lceil k/g \rceil - 1} \left(\prod_{j=0}^{g-1} \boldsymbol{D}_{ig+j} \right),$$

where we set $\boldsymbol{D}_\ell = \boldsymbol{I}_n$ for $\ell \geqslant k$. Homomorphically, we can then multiply by the $\lceil k/g \rceil$ blocks instead of the k individual matrices. This reduces the number of consumed levels to $\lceil k/g \rceil$ at the cost of more rotations and plaintext–ciphertext multiplications per block as g increases. Therefore, g can be used to balance the trade-off between computational cost and level usage. In the context of discrete Fourier transform matrices, the value 2^g is commonly referred to as the *radix* [26,17].

B Details on the Trace and Product Operations

Let $n \geqslant B$ be two strict divisors of N, and let $\boldsymbol{z} \in \mathbb{C}^n$. We divide $\boldsymbol{z}$ into n/B consecutive blocks of size B, namely $\boldsymbol{z} = (\boldsymbol{z}_i)_{i \in [0, n/B)}$ with each $\boldsymbol{z}_i \in \mathbb{C}^B$. The *trace operation* $\mathsf{Tr}_{n \to B}(\boldsymbol{z})$ computes the sum of these blocks:

$$\mathsf{Tr}_{n \to B}(\boldsymbol{z}) = \sum_{i=0}^{n/B-1} \boldsymbol{z}_i \in \mathbb{C}^B.$$

The *product operation* $\mathsf{Pr}_{n\to B}(\boldsymbol{z})$ is defined analogously, with the sum replaced by the Hadamard product. As discussed in [11], the trace and product operations are extended versions of the trace and norm operations of the number field $\mathbb{Q}[X^{N/(2n)}]/(X^N+1)$ over $\mathbb{Q}[X^{N/(2B)}]/(X^N+1)$. By viewing $\mathbb{C}^{n/2}$ (equipped with the Hadamard product) as a subring of $\mathbb{C}^n$ via the embedding $\boldsymbol{z} \mapsto (\boldsymbol{z}, \boldsymbol{z})$, we have the equality $\mathsf{Tr}_{n\to n/2} = \mathsf{id} + \mathsf{Rot}_{n/2}$, where id is the identity map. Therefore, since $\mathsf{Tr}_{n\to B} = \mathsf{Tr}_{2B\to B} \circ \cdots \circ \mathsf{Tr}_{n/2\to n/4} \circ \mathsf{Tr}_{n\to n/2}$, we can define the trace operation on a ciphertext $\mathbf{ct} = \mathsf{Enc}^*(\boldsymbol{z})$ as:

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

yielding an encryption of the vector $\mathsf{Tr}_{n\to B}(\boldsymbol{z})$ and requiring $\log(n/B)$ homomorphic rotations. Similarly, the product operation for $\mathbf{ct} = \mathsf{Enc}^*(\boldsymbol{z})$ can be defined as:

$$\mathsf{Pr}_{n\to B}(\mathbf{ct}) = (\mathsf{id} \times \mathsf{Rot}_B) \circ \cdots \circ (\mathsf{id} \times \mathsf{Rot}_{n/4}) \circ (\mathsf{id} \times \mathsf{Rot}_{n/2})(\mathbf{ct}),$$

an encryption of the vector $\mathsf{Pr}_{n\to B}(\boldsymbol{z})$. This requires $\log(n/B)$ homomorphic rotations and $\log(n/B)$ consecutive homomorphic multiplications, thereby consuming $\log(n/B)$ levels.

C Details on the R-SPRU Algorithm

Here, we elaborate on the integration of SCORE into the SPRU bootstrapping algorithm [18] for encrypted real vectors. As SPRU is less widely known than the conventional BOOT algorithm, we include a more comprehensive overview by revisiting its main steps in detail.

Secret Key. The SPRU algorithm assumes that the secret key $s = \sum_{i=0}^{N-1} s_i X^i \in \mathbb{Z}[X]/(X^N+1)$ is binary and has a small, power-of-two Hamming weight h. Additionally, it requires a constant term of $s_0 = 1$ and that, for each $b \in [0, h)$, exactly one coefficient s_{bB+j} with $j \in [0, B)$ equals one, where $B = N/h$. We refer to such a key as a *SPRU secret key*.

Decryption Equation for Each Coefficient. We aim to bootstrap a ciphertext $\mathbf{ct} = \mathsf{Enc}^*(\boldsymbol{z}) = \mathsf{Enc}(m)$ where $\boldsymbol{z} \in \mathbb{R}^n$ is a real vector and $m = \mathsf{Ecd}(\boldsymbol{z})$, with n a strict divisor of B. If we write $\mathbf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1)$ with $\mathsf{ct}_\ell = \sum_{i=0}^{N-1} \mathsf{ct}_{\ell,i} X^i$ for $\ell \in [0, 2)$, the decryption equation $m \approx \mathsf{ct}_0 + s \cdot \mathsf{ct}_1 \bmod q_0$ can be expanded as:

$$m \approx \sum_{i=0}^{N-1} \mathsf{ct}_{0,i} X^i + \sum_{i,j=0}^{N-1} s_i \mathsf{ct}_{1,j} X^{i+j} \mod q_0.$$

Therefore, if we write $m = \sum_{a=0}^{2n-1} m_a Y^a$ with $Y = X^{N/(2n)}$, then each coefficient m_a can be approximated by the scalar product $m_a \approx \langle \boldsymbol{s}, \boldsymbol{c}_a \rangle \bmod q_0$, where $\boldsymbol{s} = (s_i)_{i\in[0,N)} \in [0,2)^N$ is the coefficient vector of s, and $\boldsymbol{c}_a = (c_{a,i})_{i\in[0,N)} \in \mathbb{Z}^N$ is a vector defined in terms of the polynomials ct_0 and ct_1, namely $c_{a,0} = \mathsf{ct}_{0,aN/(2n)} + \mathsf{ct}_{1,aN/(2n)}$, and $c_{a,i} = \mathsf{ct}_{1,aN/(2n)-i}$ for $i \in [1, aN/(2n)]$, as well as $c_{a,i} = -\mathsf{ct}_{1,N+aN/(2n)-i}$ for $i \in (aN/(2n), N)$.

CKKS-Compatible Reformulation. To homomorphically evaluate the scalar product $\langle \boldsymbol{s}, \boldsymbol{c}_a \rangle \bmod q_0$ in the CKKS scheme, we embed $\mathbb{Z}_{q_0}$ into the complex plane $\mathbb{C}$ via the following group morphism:

$$\psi : \mathbb{Z}_{q_0} \to \mathbb{C}, \quad x \mapsto \exp\left(\tfrac{2\pi \mathrm{I} \cdot x}{q_0}\right).$$

If $|m_a| \ll q_0$ for all $a \in [0, 2n)$, then the small-angle approximation yields $(2\pi/q_0) \cdot m_a \approx \mathrm{Im}(\psi(m_a))$. Thus, to obtain encryptions of the coefficients m_a, we homomorphically evaluate $\psi(m_a)$, which can be achieved by homomorphically evaluating the embedded scalar product $\psi(\langle \boldsymbol{s}, \boldsymbol{c}_a \rangle)$. Since ψ is a group morphism, we have:

$$\psi(m_a) \approx \prod_{i=0}^{N-1} \psi(s_i c_{a,i}) = \prod_{b=0}^{h-1} \prod_{j=0}^{B-1} \psi(s_{bB+j} c_{a,bB+j}).$$

Recall that the values of the s_i are binary, and for each $b \in [0, h)$, there is exactly one coefficient s_{bB+j} with $j \in [0, B)$ equal to one. Therefore, the product over j can be replaced by a sum:

$$\psi(m_a) \approx \prod_{b=0}^{h-1} \sum_{j=0}^{B-1} s_{bB+j} \cdot \psi(c_{a,bB+j}).$$

To use the SCORE operation as the final step, we need to obtain an encryption of the coefficient vector $\boldsymbol{p}_0 = (p_a)_{a \in [0,n)}$, where $p = \sum_{a=0}^{2n-1} p_a Y^a = \tau_n^{-1}(\boldsymbol{z})$. Thus, compared to the standard SPRU algorithm, only the first n coefficients of p are needed. Since $p_a \approx (1/\Delta) \cdot m_a$, we have:

$$p_a \approx \tfrac{q_0}{2\pi\Delta} \cdot \mathrm{Im}(\psi(m_a)) \approx 2\mathrm{Im}\left(\prod_{b=0}^{h-1} \sum_{j=0}^{B-1} s_{bB+j} \cdot \psi(c_{a,bB+j}) \cdot \delta \right),$$

where $\delta = \sqrt[h]{q_0/(4\pi\Delta)}$. (Twice the imaginary part is used because it is easier to compute homomorphically than the imaginary part itself.) If we write each $j \in [0, B)$ as $j = uB/(2n) + k$ with $k \in [0, B/(2n))$ and $u \in [0, 2n)$, we get for the vector $\boldsymbol{p}_0$:

$$\boldsymbol{p}_0 \approx 2\mathrm{Im}\left(\prod_{b=0}^{h-1} \sum_{k=0}^{B/(2n)-1} \sum_{u=0}^{2n-1} s_{bB+uB/(2n)+k} \cdot \psi(c_{a,bB+uB/(2n)+k}) \cdot \delta \right)_{a \in [0,n)}.$$

More compactly, we can write:

$$\boldsymbol{p}_0 \approx 2\mathrm{Im} \circ \mathsf{Pr}_{hn \to n} \circ \mathsf{Tr}_{N/2 \to hn} \left(\sum_{u=0}^{2n-1} \boldsymbol{e}^{(u)} \odot \boldsymbol{s}^{(u)} \right),$$

where the vectors $\boldsymbol{e}^{(u)}$ and $\boldsymbol{s}^{(u)}$ are defined as follows:

$$\boldsymbol{e}^{(u)} = \Big(\big((\psi(c_{a,bB+uB/(2n)+k}) \cdot \delta)_{a \in [0,n)} \big)_{b \in [0,h)} \Big)_{k \in [0, B/(2n))} \in \mathbb{C}^{N/2},$$

$$\boldsymbol{s}^{(u)} = \Big(\big((s_{bB+uB/(2n)+k})_{a \in [0,n)} \big)_{b \in [0,h)} \Big)_{k \in [0, B/(2n))} \in \mathbb{C}^{N/2}.$$

Bootstrapping Equation and Algorithms. During the key generation phase, the vectors $\boldsymbol{s}^{(u)}$ are precomputed and encrypted to form the *bootstrapping keys*, denoted as $\mathsf{cs}^{(u)} = \mathsf{Enc}^*(\boldsymbol{s}^{(u)})$ ("ciphered secrets"). These ciphertexts are used throughout the bootstrapping procedure. The final bootstrapping equation is:

$$\mathsf{Enc}(m) = \mathsf{SCORE}_n \circ 2\mathrm{Im} \circ \mathsf{Pr}_{hn\to n} \circ \mathsf{Tr}_{N/2\to hn}\left(\sum_{u=0}^{2n-1} \mathsf{Ecd}\left(\boldsymbol{e}^{(u)}\right) \times \mathsf{cs}^{(u)}\right).$$

The homomorphic multiplications with the $\mathsf{cs}^{(u)}$ constitute the primary computational bottleneck, as $2n$ such multiplications are required. Since the original SPRU algorithm must process twice as many coefficients of p, it requires $4n$ such homomorphic multiplications. Thus, employing the SCORE operation reduces the number of multiplications by half, yielding a potential speedup of up to $2\times$.

We refer to this modified version of the SPRU algorithm as the *R-SPRU algorithm*. Algorithm 3 presents the procedure for generating the bootstrapping keys (using explicit indexing rather than vector concatenation, following the approach in [18]). The complete R-SPRU algorithm is detailed in Algorithm 4. The proof of correctness follows from the above derivation.

Algorithm 3: bskGen (generation of bootstrapping keys for R-SPRU)

Input: A SPRU secret key $s = \sum_{i=0}^{N-1} s_i X^i \in \mathbb{Z}[X]/(X^N+1)$
Input: The Hamming weight h of s
Input: The power-of-two number of slots $n \in [1, N/(2h)]$
Output: Corresponding bootstrapping keys $(\mathsf{cs}^{(u)})_{u\in[0,2n)}$

$B \leftarrow N/h$
for $u \in [0, 2n)$ **do**
  for $a \in [0, n)$, $b \in [0, h)$, $k \in [0, B/(2n))$ **do**
    $s^{(u)}_{knh+bn+a} \leftarrow s_{bB+uB/(2n)+k}$
  $\boldsymbol{s}^{(u)} \leftarrow \left(s_i^{(u)}\right)_{i\in[0,N/2)}$
  $\mathsf{cs}^{(u)} \leftarrow \mathsf{Enc}^*(\boldsymbol{s}^{(u)})$
return $(\mathsf{cs}^{(u)})_{u\in[0,2n)}$

In Section D, we explain how the Cooley-Tukey decomposition, as presented in [26], can be integrated into Algorithm 4.

D Cooley-Tukey Decomposition

We explain here how to perform the SlotToCoeff, CoeffToSlot and SCORE operations more efficiently by using the Cooley-Tukey decomposition [26,17].

Algorithm 4: R-SPRU (SPRU algorithm for encrypted real vectors)

Input: An encryption $\mathbf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1)$ of a real vector $\boldsymbol{z} \in \mathbb{R}^n$
Input: The power-of-two number of slots $n \in [1, N/(2h)]$
Input: The Hamming weight h of the SPRU secret key
Input: The bootstrapping keys $(\mathsf{cs}^{(u)})_{u\in[0,2n)}$
Input: The factor $\delta = \sqrt[h]{q_0/(4\pi\Delta)}$
Output: A bootstrapped encryption $\mathbf{ct}_{\mathsf{boot}}$ of $\boldsymbol{z}$

$B \leftarrow N/h$
for $a \in [0, n)$ **do**
  $c_{a,0} \leftarrow \mathsf{ct}_{0,aN/(2n)} + \mathsf{ct}_{1,aN/(2n)}$ ▷ $\mathsf{ct}_\ell = \sum_{i=0}^{N-1} \mathsf{ct}_{\ell,i} X^i$
  for $i \in [1, aN/(2n)]$ **do**
    $c_{a,i} \leftarrow \mathsf{ct}_{1,aN/(2n)-i}$
  for $i \in (aN/(2n), N)$ **do**
    $c_{a,i} \leftarrow -\mathsf{ct}_{1,N+aN/(2n)-i}$
for $u \in [0, 2n)$ **do**
  for $a \in [0, n)$, $b \in [0, h)$, $k \in [0, B/(2n))$ **do**
    $e^{(u)}_{knh+bn+a} \leftarrow \psi(c_{a,bB+uB/(2n)+k}) \cdot \delta$ ▷ $\psi : x \mapsto \exp(2\pi\mathrm{I} \cdot x/q_0)$
  $\boldsymbol{e}^{(u)} \leftarrow \left(e^{(u)}_i\right)_{i\in[0,N/2)}$
  $E^{(u)} \leftarrow \mathsf{Ecd}(\boldsymbol{e}^{(u)})$
  $\mathbf{ct}^{(u)} \leftarrow E^{(u)} \times \mathsf{cs}^{(u)}$
$\mathbf{ct}_{\mathsf{boot}} \leftarrow \sum_{u=0}^{2n-1} \mathbf{ct}^{(u)}$
$\mathbf{ct}_{\mathsf{boot}} \leftarrow \mathsf{Tr}_{N/2\to hn}(\mathbf{ct}_{\mathsf{boot}})$
$\mathbf{ct}_{\mathsf{boot}} \leftarrow \mathsf{Pr}_{hn\to n}(\mathbf{ct}_{\mathsf{boot}})$
$\mathbf{ct}_{\mathsf{boot}} \leftarrow \mathbf{ct}_{\mathsf{boot}} - \mathsf{Conj}(\mathbf{ct}_{\mathsf{boot}})$
$\mathbf{ct}_{\mathsf{boot}} \leftarrow (-X^{N/2}) \cdot \mathsf{Conj}(\mathbf{ct}_{\mathsf{boot}})$ ▷ Division by I without level consumption
$\mathbf{ct}_{\mathsf{boot}} \leftarrow \mathsf{SCORE}_n(\mathbf{ct}_{\mathsf{boot}})$ ▷ Apply Algorithm 1
return $\mathbf{ct}_{\mathsf{boot}}$

D.1 Decomposition of the Vandermonde Matrix $\boldsymbol{U_n}$

The matrix $\boldsymbol{U}_n \in \mathbb{C}^{n\times n}$ can be decomposed using the Cooley-Tukey method. This approach expresses $\boldsymbol{U}_n$ as a product of $\log n$ diagonally sparse matrices and the bit-reversal permutation matrix.

We start by defining the bit-reversal operation. Given an integer $i \in [0, n)$, its *bit reversal* $\mathsf{br}_n(i)$ with respect to the power of two n is the integer obtained by reversing the order of the $\log n$ bits in the binary representation of i. Explicitly, if $i = \sum_{\ell=0}^{\log n-1} b_\ell 2^\ell$ with $b_\ell \in [0, 2)$, then $\mathsf{br}_n(i) = \sum_{\ell=0}^{\log n-1} b_{\log n-1-\ell} 2^\ell$. The *bit-reversal permutation matrix* is then the binary matrix $\boldsymbol{\Pi}_n \in [0, 2)^{n\times n}$ whose entry (i, j) with $i, j \in [0, n)$ is equal to 1 if and only if $j = \mathsf{br}_n(i)$. Left-multiplying a vector $\boldsymbol{z} = (z_i)_{i\in[0,n)} \in \mathbb{C}^n$ by this matrix has the effect of permuting the slots of $\boldsymbol{z}$ according to the bit-reversal operation. In other words, it holds that $\boldsymbol{\Pi}_n \cdot \boldsymbol{z} = (z_{\mathsf{br}_n(i)})_{i\in[0,n)}$.

With this notation, the decomposition of $\boldsymbol{U}_n$ can be written as follows [26]:

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

where each matrix $\boldsymbol{D}_{n,n/2^{\ell+1}}$ is diagonally sparse. More precisely, for each $\ell \in [0, \log n)$, the matrix $\boldsymbol{D}_{n,2^\ell} \in \mathbb{C}^{n\times n}$ is block diagonal, consisting of 2^ℓ copies of the following block of size $(n/2^\ell) \times (n/2^\ell)$:

$$\left(\begin{array}{c|c} \boldsymbol{I}_{n/2^{\ell+1}} & \mathsf{Diag}(\zeta_{4n}^{5^i\cdot 2^\ell})_{i\in[0,n/2^{\ell+1})} \\ \hline \boldsymbol{I}_{n/2^{\ell+1}} & -\mathsf{Diag}(\zeta_{4n}^{5^i\cdot 2^\ell})_{i\in[0,n/2^{\ell+1})} \end{array}\right);$$

we recall that $\zeta_{4n} = \exp(2\pi \mathrm{I}/(4n))$. Notice that the only non-zero diagonals $\mathsf{Diag}_j(\boldsymbol{D}_{n,2^\ell})$ are those corresponding to $j \in \{0, n/2^{\ell+1}, n - n/2^{\ell+1}\}$, and so $\boldsymbol{D}_{n,2^\ell}$ is indeed diagonally sparse.

D.2 Application to SlotToCoeff, CoeffToSlot and SCORE

The SlotToCoeff operation is implemented as a matrix–ciphertext multiplication with the matrix $\boldsymbol{U}_n$. Although (2) does not directly decompose $\boldsymbol{U}_n$ into diagonally sparse matrices, it does provide such a decomposition for the product $\boldsymbol{U}_n\boldsymbol{\Pi}_n$, namely:

$$\boldsymbol{U}_n\boldsymbol{\Pi}_n = \prod_{\ell=0}^{\log n-1} \boldsymbol{D}_{n,n/2^{\ell+1}}.$$

Therefore, the Cooley-Tukey version of the SlotToCoeff operation is a homomorphic matrix–ciphertext multiplication with the matrix $\boldsymbol{U}_n\boldsymbol{\Pi}_n$ rather than $\boldsymbol{U}_n$. This version allows balancing computational cost against level consumption, as detailed in Section A.

Similarly, the CoeffToSlot operation corresponds to a homomorphic matrix–ciphertext multiplication with the matrix $(1/n)\cdot\overline{\boldsymbol{U}}_n^{\mathsf{T}}$. Using (2) and the fact that $\boldsymbol{\Pi}_n$ is a symmetric matrix, we obtain the decomposition:

$$\tfrac{1}{n}\boldsymbol{\Pi}_n\overline{\boldsymbol{U}}_n^{\mathsf{T}} = \prod_{\ell=0}^{\log n-1} \tfrac{1}{2}\overline{\boldsymbol{D}}_{n,2^\ell}^{\mathsf{T}}.$$

The Cooley-Tukey version of the CoeffToSlot operation is therefore a homomorphic matrix–ciphertext multiplication with the matrix $(1/n)\cdot\boldsymbol{\Pi}_n\overline{\boldsymbol{U}}_n^{\mathsf{T}}$, again offering the same trade-off between cost and level usage as discussed in Section A.

Likewise, SCORE admits a Cooley-Tukey version. Recall that SCORE is based on a homomorphic matrix–ciphertext multiplication with the matrix $\boldsymbol{U}'_n = \boldsymbol{U}_n \cdot \mathsf{Diag}(1/2, 1, \ldots, 1)$. By using (2), we can write:

$$\boldsymbol{U}'_n\boldsymbol{\Pi}_n = \prod_{\ell=0}^{\log n-1} \boldsymbol{D}'_{n,n/2^{\ell+1}},$$

where $\boldsymbol{D}'_{n,n/2} = \boldsymbol{D}_{n,n/2}\cdot\mathsf{Diag}(1/2, 1, \ldots, 1)$ and $\boldsymbol{D}'_{n,2^\ell} = \boldsymbol{D}_{n,2^\ell}$ for $\ell \in [0, \log n-1)$. Thus, the Cooley-Tukey version of SCORE uses a matrix–ciphertext multiplication with the matrix $\boldsymbol{U}'_n\boldsymbol{\Pi}_n$, and is subject to the techniques discussed in Section A.

Note, however, that the introduction of the bit-reversal permutation matrix $\boldsymbol{\Pi}_n$ into the above operations requires some care. Specifically, the Cooley-Tukey version of the CoeffToSlot operation produces an encrypted vector whose slots are in bit-reversed order. Analogously, when applying the Cooley-Tukey version of the SlotToCoeff or SCORE operation, the encrypted input vector must have its slots in bit-reversed order. Otherwise, the resulting plaintext polynomial will have its coefficients permuted accordingly, which is typically not what is desired.

D.3 Application to R-BOOT

To integrate the Cooley-Tukey decomposition into the R-BOOT algorithm (Algorithm 2), it suffices to replace the CoeffToSlot and SCORE operations with their Cooley-Tukey variants. Although the Cooley-Tukey version of CoeffToSlot yields an encrypted vector with slots in bit-reversed order, this is again corrected by the final application of the Cooley-Tukey SCORE operation. The intermediate steps between CoeffToSlot and SCORE (lines 4 to 5 in Algorithm 2) are unaffected by the bit-reversed order: they operate slotwise on the encrypted vector.

D.4 Application to R-SPRU

Unlike R-BOOT, the R-SPRU algorithm only applies the SCORE operation at the end and does not make use of the CoeffToSlot operation. As a result, we do not benefit from the same "cancellation effect" of bit reversal seen in R-BOOT, where the bit-reversed output of the Cooley-Tukey CoeffToSlot is corrected by a subsequent Cooley-Tukey SCORE. Consequently, to use the Cooley-Tukey variant of SCORE in R-SPRU, we must ensure that the input ciphertext to SCORE has its slots already arranged in bit-reversed order.

In R-SPRU, we homomorphically operate on the vectors $\boldsymbol{e}^{(u)}$ and $\mathsf{cs}^{(u)}$, which are of length $N/2$ rather than n. To ensure the final encrypted vector passed into SCORE is bit-reversed with respect to n, we divide each length-$(N/2)$ vector $\boldsymbol{e}^{(u)}$ and $\boldsymbol{s}^{(u)}$ into $N/(2n)$ contiguous blocks of size n, and reorder the slots within each individual block according to bit reversal with respect to n.

This reordering of vector slots can be expressed as a left-multiplication by a certain permutation matrix $\boldsymbol{\Pi}^{(n)}_{N/2}$ of size $(N/2) \times (N/2)$. This matrix is block-diagonal, consisting of $N/(2n)$ copies of the $(n \times n)$-permutation matrix $\boldsymbol{\Pi}^{(n)}$ along its diagonal. Left-multiplying a vector $\boldsymbol{z} \in \mathbb{C}^{N/2}$ by $\boldsymbol{\Pi}^{(n)}_{N/2}$ has the intended effect: it partitions $\boldsymbol{z}$ into $N/(2n)$ contiguous blocks of length n, and reorders the slots within each block according to bit reversal with respect to n.

To modify the R-SPRU algorithm to support Cooley-Tukey SCORE, we therefore proceed as follows. During the generation of bootstrapping keys (see Algorithm 3), we replace line 6 with:

$$\mathsf{cs}^{(u)} \leftarrow \mathsf{Enc}^*\big(\boldsymbol{\Pi}^{(n)}_{N/2}\boldsymbol{s}^{(u)}\big).$$

Similarly, in Algorithm 4, we update line 12 as follows:

$$E^{(u)} \leftarrow \mathsf{Ecd}\big(\boldsymbol{\Pi}^{(n)}_{N/2}\boldsymbol{e}^{(u)}\big).$$

These changes ensure that the encrypted input vector to SCORE is in bit-reversed order with respect to n, as required.