Modular Reduction in CKKS

Jaehyung Kim[1,2] and Taeyeong Noh[1]

[1] CryptoLab Inc., Seoul, Republic of Korea
tynoh0219@cryptolab.co.kr
[2] Stanford University, Stanford, United States of America
jaehk@stanford.edu

Abstract. The Cheon–Kim–Kim–Song (CKKS) scheme is renowned for its efficiency in encrypted computing over real numbers. However, it lacks an important functionality that most exact schemes have, an efficient modular reduction. This derives from the fundamental difference in encoding structure. The CKKS scheme encodes messages to the least significant bits, while the other schemes encode to the most significant bits (or in an equivalent manner). As a result, CKKS could enjoy an efficient rescaling but lost the ability to modular reduce inherently.

Our key observation is that at the very bottom modulus, plaintexts encoded in the least significant bits can still enjoy the inherent modular reduction of RLWE. We suggest incorporating modular reduction as a primary operation for CKKS and exploring its impact on efficiency. We constructed a novel homomorphic modular reduction algorithm using the discrete bootstrapping from Bae et al. [Asiacrypt'24] and a new discretization algorithm from modulus switching. One of the key advantages of our modular reduction is that its computational complexity grows sublinearly ($O(\log k)$) as we increase the input range $[0, k)$, which is asymptotically better than the state-of-the-art with $\geq O(k)$.

We checked our algorithms with concrete experiments. Notably, our modulo 1 function for input range $[0, 2^{20})$ takes only 44.9 seconds with 13.3 bits of (mean) precision, in a single-threaded CPU. Recall that modular reduction over such a large range was almost infeasible in the previous works, as they need to evaluate a polynomial of degree $> 2^{20}$ (or equivalent). As an application of our method, we compared a bit decomposition based on our framework with the state-of-the-art method from Drucker et al. [J.Cryptol'24]. Our method is 7.1× faster while reducing the failure probability by more than two orders of magnitude.

Keywords: Homomorphic Encryption · CKKS · Modular Reduction

1 Introduction

Among the widely used fully homomorphic encryption (FHE) schemes, the Cheon–Kim–Kim–Song (CKKS) scheme [CKKS17] provides efficient real number computation which is essential in many applications like machine learning [LLL+22, CBH+22]. This characteristic stems from the unique functionality that CKKS provides called *rescaling*, which approximately divides the message by a constant. However, this prevents CKKS from supporting intrinsic modular reduction unlike other major schemes (BGV [BGV12], BFV [Bra12,FV12], CGGI [CGGI16], and DM [DM15]). As a result, CKKS relies on polynomial approximations to handle discrete operations (e.g. in [CKK$^+$19,CKK20,LLNK22,LLKN22] for homomorphic comparison) and suffers from relatively low performance.

This paper mainly focuses on exploiting the inherent modular reduction function on $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^N+1)$, providing efficient discrete operations for CKKS. Recall that we already have been using modular reduction in CKKS, especially in rescaling where we subtract the remainder $([b]_q, [a]_q)$ from the original ciphertext (b, a) to get an approximate division. In addition, some works [KDE$^+$24,CCKS23] partially used modular reduction for designing new features in CKKS. However, none of the works regarded modular reduction as a new elementary operation of CKKS and explored its impact. In this work, we suggest incorporating modular reduction as part of primary CKKS operations and using it to handle discrete computations.

1.1 Technical Overview

Given a power-of-two integer N and $Q \in \mathbb{Z}_{>0}$, let $\mathcal{R} = \mathbb{Z}[X]/(X^N+1)$ and $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R}$. We observe that modular reduction on $\mathcal{R}_Q$ is inherent: given $q \mid Q$ and $p(X) \in \mathcal{R}_Q$, $[p(X)]_q$ can be defined as taking modulo q for each coefficient. Given an RLWE ciphertext $\mathsf{ct} = (b, a) \in \mathcal{R}_Q^2$ that decrypts to $m \in \mathcal{R}$ (i.e. $[\mathsf{ct} \cdot \mathsf{sk}]_Q = m$ for the secret key sk), we may define

$$[\mathsf{ct}]_q = ([b]_q, [a]_q) \in \mathcal{R}_q^2,$$

which decrypts to $[m]_q$.

Recall that there are two types of encoding in CKKS, namely coefficients-encoding and slots-encoding. Coefficients-encoding puts a real vector $\boldsymbol{z} \in \mathbb{R}^N$ into $\mathcal{R}$ by scaling up and rounding. In other words, coefficients encoding $\mathsf{CoeffEcd} : \mathbb{R}^N \to \mathcal{R}$ is defined as

$$\mathsf{CoeffEcd}(\boldsymbol{z}) = \sum_{i=0}^{N-1} \lfloor \Delta \cdot z_i \rceil \cdot X^i \in \mathcal{R}$$

where $\boldsymbol{z} = (z_0, z_1, \dots, z_{N-1})$ and $\Delta \in \mathbb{R}_{>0}$ is a scaling factor. On the other hand, slots-encoding contains a canonical embedding (i.e. discrete Fourier transform (DFT)) in addition, efficiently supporting single instruction multiple data (SIMD) computations. Given $\boldsymbol{z} \in \mathbb{C}^{N/2}$, the slots-encoding is defined as

$$\mathsf{Ecd}(\boldsymbol{z}) = \lfloor \Delta \cdot \mathsf{iDFT}(\boldsymbol{z}) \rceil \in \mathcal{R}$$

where $\mathsf{iDFT} : \mathbb{C}^{N/2} \to \mathbb{R}[X]/(X^N+1)$ is an inverse DFT. The straightforward observation is that the RLWE modular reduction gives a modular reduction for coefficients-encoding but not for slots-encoding, as iDFT and modular reduction do not commute.

Despite its natural definition, the usage of modular reduction in CKKS has been limited. There are two main reasons for this. First, CKKS ciphertexts are usually in slots-encoded state which is not compatible with modular reduction. They are in coefficients-encoded state only at the very bottom modulus right before the $\mathsf{ModRaise}$ step in the context of bootstrapping[3]. Alternatively, one may convert slots-encoded ciphertext to coefficients-encoded ciphertext, but it involves an expensive homomorphic linear transform called slots-to-coefficients (StC). Second, unless we decrypt right after the modular reduction, the output of modular reduction requires bootstrapping to allow further computations. However, the output ciphertext of $[\cdot]_q$ could encrypt an arbitrary element of the plaintext space $\mathcal{R}_q$, which makes bootstrapping very difficult. Recall that the conventional CKKS bootstrapping (first instantiated in [CHK$^+$18]) does require a gap between underlying plaintext and ciphertext modulus, as it relies on polynomial approximations to remove the error term coming from $\mathsf{ModRaise}$.

To solve this problem, we incorporate bootstrapping into modular reduction and use what we call most significant bits (MSB) bootstrapping. Although we bootstrap every time we need modular reduction, the efficiency of the new method outperforms the typical approximation-based methods. For MSB bootstrapping, we follow the framework of [BKSS24] and extend it by using iterative algorithms and modulus switching. We first illustrate our MSB bootstrapping framework and describe modular reduction for discrete and approximate data.

Iterative MSB bootstrapping. We introduce an iterative MSB bootstrapping method which extends the MSB bootstrapping methods in [BGGJ20, BKSS24]. The main advantage of the iterative method is that it allows us to achieve arbitrary precision (as in [BCC$^+$22] for general CKKS bootstrapping) which is essential in our modular reduction framework. Let $\mathsf{ct} = (b, a) \in \mathcal{R}_{q_0}^2$ be a coefficients-encoded RLWE ciphertext encrypting a message $m \in \mathcal{R}_{t^\ell}$ in the most significant bits. To clarify, this means that $[\mathsf{ct} \cdot \mathsf{sk}]_{q_0} = (q_0/t^\ell) \cdot m$ for the secret sk. We sequentially extract the least significant digits as follows. In the first step, we multiply ct by $t^{\ell-1}$ and get a CKKS ciphertext encrypting $[m]_t$ in the most significant bits. We bootstrap this ciphertext (using integers-to-integers bootstrapping from [BKSS24]) and have $\mathsf{ct}' \in \mathcal{R}_Q^2$ encrypting $[m]_t$. Next, we put it into coefficients (via StC) and subtract it from the original ciphertext ct. At this stage, we have a slots-encoded CKKS ciphertext encrypting the least significant digit of m and a coefficients-encoded RLWE ciphertext encrypting all the other digits (i.e. $(m - [m]_t)/t$) in the most significant bits. Thus, we may extract all the digits by repeating this and combining the extracted digits if necessary.

MSB bootstrapping for reals. Based on the high-precision MSB bootstrapping for discrete data (described in the previous paragraph), we propose a novel reals-to-reals MSB bootstrapping using modulus switching. The key observation is

[3] Here one needs to use slots bootstrapping [BCC$^+$22] rather than coefficients bootstrapping [CHK$^+$18]. In other words, StC should be the very first step of bootstrapping.

that modulus switching to a small modulus converts a (not necessarily discrete) CKKS ciphertext into a discrete CKKS ciphertext. That is, given a conventional coefficients-encoded CKKS ciphertext $\mathsf{ct} = (b, a) \in \mathcal{R}_{q_0}^2$ encrypting a vector $\boldsymbol{z} \in [0, 1)^N$ with a scaling factor $\Delta_0 = q_0$, modulus-switching it to a modulus t^ℓ and returning back to q_0 can be regarded as discretizing the ciphertext while allowing some errors in the least significant bits, which makes ciphertext compatible with discrete bootstrapping. To be precise, the output ciphertext encrypts a plaintext $m \in \mathcal{R}_{t^\ell}$ that is approximately equal to $t^\ell \cdot \boldsymbol{z}$, with a scaling factor $\Delta_0 = q_0/t^\ell$. We then use the MSB bootstrapping to bootstrap the ciphertext. Since modulus switching was an approximate identity, the result of the bootstrapping should be a valid bootstrapping of the input ciphertext.

Homomorphic modular reduction. By using the (large-precision) MSB bootstrapping as a subroutine, we construct a novel homomorphic modular reduction. Let $\mathsf{ct} = (b, a) \in \mathcal{R}_q^2$ be a slots-encoded CKKS ciphertext encrypting a message $\boldsymbol{z} \in \mathbb{C}^{N/2}$. We choose the base level scaling factor (i.e. the scaling factor that corresponds to the base modulus q_0) as $\Delta_0 = q_0$. The main observation is that, performing $[\cdot]_{q_0}$ on a coefficients-encoded ciphertext results in performing $[\cdot]_1$ on the message. More precise, performing StC and $[\cdot]_{q_0}$, we obtain a coefficients-encoded RLWE ciphertext encrypting a plaintext $[z(X)]_{q_0} \in \mathcal{R}$ whose cofficients belong to either $[Re(\boldsymbol{z})]_1$ or $[Im(\boldsymbol{z})]_1$. We apply the MSB bootstrapping to recover the modulus and get the modular reduction as desired. This method works both for discrete and approximate data. Note that this method is not applicable to bootstrapping which requires a gap between the ciphertext and its modulus, such as general CKKS bootstrapping methods.

1.2 Contributions

Efficient MSB bootstrapping for reals. In the conventional CKKS bootstrapping methods, the underlying message of a ciphertext right before the $\mathsf{ModRaise}$ step needs to have a gap with the base modulus, in order to approximate a modular reduction function homomorphically. Such a gap causes extra modulus consumption during bootstrapping, as illustrated in [BCKS24]. To be precise, as the size of the underlying message enlarges through $\mathsf{ModRaise}$ (i.e. from m to $m + q_0 I$), one typically uses larger modulus per rescaling inside bootstrapping (e.g. [Cry22, lat23]). This results in performance degradation as modulus is a very limited resource in homomorphic encryption.

In this regard, finding an efficient MSB bootstrapping is an interesting research question (and necessary for our modular reduction framework). Although bootstrapping the most significant bits has been partially explored in [BGGJ20], it did not specify the bootstrapping instantiation concerning the encoding structure of CKKS. In [BCKS24], the authors adapted the CKKS bootstrapping for bits, providing an MSB bootstrapping for bits. This has been further improved in [BKSS24], from integers to integers.

In this work, we efficiently extend the MSB bootstrapping to arbitrary precision and real numbers. One of the biggest limitations of discrete bootstrapping in [BKSS24] is that its computational complexity grows exponentially with respect to the precision (in bits). To solve this problem, we introduce an iterative method (as in [BCC$^+$22, BZP$^+$23]) to efficiently support high-precision discrete bootstrapping. In addition, we introduce an extra modulus switching step at the bottom modulus to extend the MSB bootstrapping to reals. With these two improvements, we propose a full-fledged MSB bootstrapping framework for the first time.

Asymptotically faster homomorphic modular reduction. All the existing approaches [CHK$^+$18, LLL$^+$21, JM22, LLK$^+$22, HMWW24] for homomorphic modular reduction (with CKKS) rely on polynomial approximations. In order to approximate the modulo 1 function over the interval $[0, k)$, the polynomial degree for approximation is at least linear in k. To check this, one can simply count the number of roots (i.e. the number of integer points) which is k. As homomorphic polynomial evaluation of degree k needs $O(k)$ constant ciphertext multiplications, the running time should also be $O(k)$. As a result, homomorphic modular reduction on large intervals has been extremely expensive and considered almost impossible.

In contrast, our modular reduction method is far less dependent on the input interval $[0, k)$. Instead of approximating the modular reduction function over the large interval $[0, k)$, we just take the inherent modular reduction which is independent from the input interval. As we increase k, we only need to increase the precision of StC, leading to $O(\log(k))$ bits of modulus consumption. Overall, the running time of modular reduction should be at most $O(\log(k))$, which is somewhat negligible in practice. To summarize, we provide a very efficient modular reduction over large intervals, which is asymptotically better than the existing methods.

Discretization for CKKS. As observed in the previous section, the modulus switching at the bottom modulus enables a discretization from a coefficients-encoded CKKS ciphertext into a coefficients-encoded discrete CKKS ciphertext. This allows us to *discretize* a typical CKKS ciphertext, and to use discrete CKKS afterwards. One of the most important advantages of discrete CKKS over conventional CKKS is that it supports arbitrary function evaluation via look-up tables [CKKL24]. In addition, we may bit-decompose a CKKS ciphertext using the framework in [BKSS24], which provides an alternative solution for comparison-like functions.

Look-up table-based approaches can accelerate many functions that are difficult to approximate. For instance, non-polynomial functions such as exponential, inverse, square root, and ReLU can be evaluated in this manner. Recall that these functions are essential in machine learning, which is one of the primary applications of homomorphic encryption. Furthermore, this approach can be considered as an analogue of quantization in the context of machine learning, as we reduce the computation cost by shrinking the data type of the underlying message.

Bit-decomposed CKKS ciphertexts can be used to accelerate comparison-like functions such as argmax and sorting. Recall that the existing homomorphic comparison approaches use polynomial approximations, and thus they require large multiplicative depths and time for evaluation. Hence, functions that contain homomorphic comparison as a subroutine have been considered very expensive. On the other hand, comparisons for bit-decomposed ciphertexts are relatively cheap, and we may use this comparison to build more complex functions like argmax and sorting.

Concrete experiments. We experimented with our algorithms to support the correctness and efficiency of our algorithm. All experiments are performed in a single-thread CPU and implemented with ring degree $N = 2^{16}$. Table 1 and 2 summarizes the performance of iterative MSB bootstrapping and homomorphic modular reduction for both integers and reals. The mean precision indicates $-\log_2 \mathbb{E}(\|e\|_1)$. We denote $\mathsf{IntBoot}$ (resp. IntMod) as the high-precision iterative MSB bootstrapping (resp. homomorphic modular reduction) with input and output are ciphertexts which encrypt 15 bits integers, and Boot (resp. Mod) as the iterative MSB bootstrapping (resp. homomorphic modular reduction) with real inputs.

Notably, our homomorphic modular reduction over an input range $[0, 2^{20})$ achieved 13.3 bits of (mean) precision in 44.9 seconds. Note that this is the first implementation of homomorphic modular reduction for such a large interval, as the previous approaches require significantly large degree polynomials to approximate the mod function. For application, we proposed a novel bit decomposition method using our framework. We compared the performance to the state-of-the-art method in Drucker et al [DMPS24] (see Table 3). In terms of running time, our method is 7.1× faster for 10-bit decomposition while maintaining precision. Please see Section 6 for further details.

Table 1. Performance of our integer MSB bootstrapping and homomorphic modular reduction algorithms.

	Input/Output bits	Mean precision	Total time	Amortized time
$\mathsf{IntBoot}$	15	28.5	42.6s	2.60ms
IntMod	15	28.1	43.3s	2.64ms

1.3 Related Works

We summarize the important related works to specify the works that are either used as key building blocks or needed to better assess the efficiency of our work. Some of them will be explained in more detail in Section 2 and 3.

Table 2. Performance of our real MSB bootstrapping with input range $[0,1)$ and homomorphic modular reduction algorithms with input range $[0,2^{20})$.

	Mean precision	Total time	Amortized time
Boot	13.3	43.7s	2.66ms
Mod	13.3	44.9s	2.74ms

Table 3. Comparing performance of our 10-bit extraction algorithm with [DMPS24].

	T_{avg} (sec)
Ours	53.8
[DMPS24]	385
gain	7.1×

Discrete CKKS. The message space of the original CKKS [CKKS17] is $\mathbb{C}^{N/2}$, which can be described as *approximate* data. In [DMPS24], the authors suggested using discrete message space, by introducing an embedding to $\mathbb{C}^{N/2}$. For instance, one may embed bits (i.e. $\{0,1\}$) into $\mathbb{C}^{N/2}$ with an identity embedding. One of the key differences between approximate and discrete CKKS is that discrete CKKS has a notion of cleaning, which removes the noise of the underlying plaintext. In order to improve the efficiency of look-up table evaluations, [CKKL24] introduced a roots-of-unity embedding and explored multivariate interpolations. Independently, some works [BCKS24, BKSS24] adapted CKKS bootstrapping specific to discrete data, leading to greatly improving its performance.

Modular Reduction. As one of the key steps of CKKS bootstrapping called EvalMod is a modular reduction. Modular reduction has been extensively studied throughout the CKKS literature [CHK$^+$18, LLL$^+$21, JM22, LLK$^+$22]. However, due to the limit of polynomial approximation (see [HMWW24]), these works could only evaluate modular reductions for a small input range (e.g. 25 periods). On the other hand, our work uses an inherent modular reduction functionality that RLWE ciphertexts already have. Although several works [KDE$^+$24, CCKS23] used such a feature to build some advanced functionalities upon the CKKS scheme, they did not explore the use of modular reduction as a primary operation of CKKS. One of the key techniques we needed to enable efficient modular reduction can be described as Most Significant Bit (MSB) bootstrapping, which bootstraps a ciphertext whose underlying message does not have a gap with the modulus. MSB bootstrapping has been partially covered in [BCKS24, BGGJ20, BKSS24], but none of these works provided an efficient bootstrapping for reals that could allow modular reduction in a practical sense. To tackle the problem, we follow the philosophy of iterative bootstrapping [BCC+22,BZP+23] and rely on modulus switching.

2 Preliminaries

Given a power-of-two integer $N > 1$, let $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ and $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R}$ for $Q \in \mathbb{Z}_{>0}$. For $m \in \mathbb{Z}$, $[m]_q$ denotes the signed modular reduction whose output lives in $(-q/2, q/2]$.

2.1 CKKS Basics

Let $m \in \mathcal{R}$ be a plaintext. Given a secret key $\mathsf{sk} \in \mathcal{R}$, an RLWE ciphertext ct that encrypts m is a pair $(b, a) \in \mathcal{R}_Q^2$ such that $[\mathsf{ct} \cdot \mathsf{sk}]_Q = m$.

Encoding structure. Let $m(x) \in \mathcal{R}$ be a plaintext and $\Delta \in \mathbb{R}$ be a scaling factor. Let $\mathsf{DFT} : \mathbb{R}[X]/(X^N + 1) \to \mathbb{C}^{N/2}$ be defined as

$$\mathsf{DFT}(p(x)) = (p(\zeta_i))_{0 \le i < N/2}$$

where $\zeta_i = \zeta^{5^i}$ for a $2N$-th root of unity ζ. The decoding map $\mathsf{Dcd} : \mathcal{R} \to \mathbb{C}^{N/2}$ is defined as

$$\mathsf{Dcd}(m) = \frac{1}{\Delta}\mathsf{DFT}(m).$$

Let $\mathsf{iDFT} : \mathbb{C}^{N/2} \to \mathbb{R}[X]/(X^N + 1)$ be an inverse of DFT. The encoding map $\mathsf{Ecd} : \mathcal{R} \to \mathbb{C}^{N/2}$ is defined as

$$\mathsf{Ecd}(\boldsymbol{z}) = \lfloor \Delta \cdot \mathsf{iDFT}(\boldsymbol{z}) \rceil.$$

Encoding and decoding give a correspondence between $\mathcal{R}$ and $\mathbb{C}^{N/2}$, allowing us to use $\mathbb{C}^{N/2}$ as a message space of RLWE ciphertexts. Note that DFT and iDFT are 2-norm isometry with constant factor:

$$\|\mathsf{DFT}(p(x))\|_2 = \sqrt{\frac{N}{2}} \cdot \|p(x)\|_2$$

for all $p \in \mathcal{R}$.

Rescaling. Let $\mathsf{ct} = (b, a) \in \mathcal{R}_Q^2$ be a CKKS ciphertext encrypting $\boldsymbol{z} \in \mathbb{C}^{N/2}$ with a scaling factor Δ. Rescaling of ct by $q \mid Q$ is defined as

$$\mathsf{RS}_q(\mathsf{ct}) = \left(\frac{b - [b]_q}{q}, \frac{a - [a]_q}{q} \right)$$

which decreases the scaling factor from Δ to Δ/q. Note that rescaling is not an exact division by q, and it generates a small error (denoted as rescaling error) in the least significant bits.

Inverse rescaling. Let $\mathsf{ct} = (b, a) \in \mathcal{R}_Q^2$ be a CKKS ciphertext encrypting $\boldsymbol{z} \in \mathbb{C}^{N/2}$ with a scaling factor Δ. Inverse rescaling of ct by q' is defined as

$$\mathsf{Inv\text{-}RS}_{q'}(\mathsf{ct}) = (q' \cdot b, q' \cdot a) \in \mathcal{R}_{q'Q}^2$$

which increases the scaling factor from Δ to $q' \cdot \Delta$.

Modulus switching. Let $\mathsf{ct} \in \mathcal{R}_Q^2$ be an RLWE ciphertext and $Q' > 0$ be an integer. We define modulus switching from Q to Q' (denoted as $\mathsf{ModSwitch}_Q^{Q'}$) as first inverse rescaling to $\mathsf{lcm}(Q, Q')$, followed by a rescaling to Q'. That is:

$$\mathsf{ModSwitch}_Q^{Q'}(\mathsf{ct}) = \mathsf{RS}_{\mathsf{lcm}(Q,Q')/Q'}(\mathsf{Inv\text{-}RS}_{\mathsf{lcm}(Q,Q')/Q}(\mathsf{ct})) \in \mathcal{R}_{Q'}$$

This reformulation is inspired from *Rational Rescale* in [CCKK24, Definition 1].

CKKS Bootstrapping. The modulus of a CKKS ciphertext decreases through homomorphic computations. In order to recover the modulus, we need bootstrapping which increases the modulus while approximately preserving the underlying message. The conventional StC-first CKKS bootstrapping [BCC$^+$22] works as follows[4]:

1. **Slots-to-Coefficients** (StC): We homomorphically evaluate DFT to convert a ciphertext encrypting a message $\boldsymbol{z}$ to a ciphertext encrypting a plaintext $z(x)$ whose coefficients are the slots of $\boldsymbol{z}$.
2. **Modulus Raising** ($\mathsf{ModRaise}$): Given a coefficients-encoded ciphertext $\mathsf{ct} \in \mathcal{R}_{q_0}^2$ at the bottom modulus q_0 encrypting a plaintext $m(x) \in \mathcal{R}$, we embed it to $\mathcal{R}_Q^2$ for larger Q. That is, we first embed it to $\mathcal{R}^2$ with natural embedding from $\mathbb{Z}_q \to \mathbb{Z}$ and take modulo Q. In terms of underlying plaintext, it changes from m to $m + q_0 I$ where I is a small integer polynomial in $\mathcal{R}$.
3. **Coefficients-to-Slots** (CtS): We homomorphically evaluate iDFT to convert a ciphertext encrypting a plaintext $m(x)$ to a ciphertext encrypting a message $\boldsymbol{m}$ whose slots are the coefficients of $m(x)$.
4. **Approximate Modular Reduction** ($\mathsf{EvalMod}$): In order to remove the small multiple of q_0 introduced during $\mathsf{ModRaise}$, we approximate the modular reduction function and evaluate homomorphically. A typical choice is to use a polynomial that approximates some trigonometric function. Since we need some level of continuity to approximate modular reduction, one needs a gap between the message m and the base modulus q_0 during $\mathsf{ModRaise}$.

[4] Note that one can also use the bootstrapping that works in the order of $\mathsf{ModRaise} - \mathsf{CtS} - \mathsf{EvalMod} - \mathsf{StC}$ [CHK$^+$18]. However, this type of bootstrapping is not directly compatible with our framework because it does not have coefficients-encoded ciphertexts during bootstrapping. For this reason, we stick to the bootstrapping that starts with StC.

2.2 Discrete Computations in CKKS

Recently, a sequence of works developed a different philosophy than the original CKKS, which is to use CKKS for discrete (rather than approximate) data. As these works are relatively new and have brought significant changes in many building blocks of CKKS, we provide a detailed overview of the new philosophy, namely discrete CKKS.

Discrete Encoding. In [DMPS24], the authors suggested the first concrete framework to use CKKS for computing discrete data. The main idea of discrete CKKS is very simple: we choose a finite set in $\mathbb{C}$ and restrict the message space accordingly. One advantage is that we can use interpolation instead of approximation, allowing one to evaluate arbitrary functions. In this context, the specific choice of representatives in $\mathbb{C}$ for computation is influential in terms of efficiency.

The most straightforward encoding is to encode using an inclusion $\mathbb{Z} \subset \mathbb{C}$ as in [DMPS24]. To be precise, any vector $\boldsymbol{z} \in \mathbb{Z}^{N/2}$ is regarded as a complex vector $\boldsymbol{z} \in \mathbb{C}^{N/2}$ and go through CKKS encoding (i.e. canonical embedding) as usual. The advantage of this encoding is that it naturally inherits arithmetic operations like addition and multiplication directly from the homomorphic property of CKKS. Another type of encoding is the roots of unity encoding, as proposed in [CKKL24]. Here they focus on primitive t-th roots of unity for some t, which can be identified with $\mathbb{Z}_t$ with the map $x \mapsto e^{2\pi i x/t} : \mathbb{Z}_t \to \mathbb{C}$. As illustrated in [CKKL24], roots of unity provide numerically stable look-up table evaluation, which means that it provides a precise evaluation of an arbitrary function $\varphi : \{1, \omega, \ldots, \omega^{t-1}\} \to \mathbb{C}$ where $\omega = e^{2\pi i/t}$ is a primitive t-th root of unity. We denote a homomorphic evaluation of φ with interpolation as LUT_{φ}.

Cleaning. One of the key ingredients of discrete CKKS is *cleaning* [DMPS24], which reduces the error via cleaning polynomials. Note that such a feature does not exist in the original CKKS for approximate data, as one cannot identify the true message hidden behind the RLWE errors. On the other hand, as the message space of discrete CKKS is finite and discrete, the error-free message can easily be identified as the nearest point among the representatives. This allows us to have an interpolation that sends an underlying message closer to the real value, reducing the error. For instance, $h_1 : \mathbb{R} \to \mathbb{R}$ defined as $h_1(x) = 3x^2 - 2x^3$ defined in [DMPS24] has a cleaning functionality with respect to the points $\{0, 1\}$, as it satisfies (1) $h_1(0) = 0$, (2) $h_1(1) = 1$, and (3) $h_1'(0) = h_1'(1) = 0$.

Discrete Bootstrapping In [BCKS24], the authors adapted CKKS bootstrapping to bit encoding $\{0, 1\}$, enabling bootstrapping without a gap between message and base modulus, thereby leading to a reduction in modulus consumption. They modify $\mathsf{ModRaise}$ and $\mathsf{EvalMod}$ to achieve this.

- $\mathsf{ModRaise}$: They set $\Delta = q_0/2$ so that the bits are placed at the most significant bits of the ciphertext $\mathsf{ct} \in \mathcal{R}_{q_0}^2$.

- EvalMod: They replace the approximate modular reduction function to a trigonometric function $f(x) = \frac{1}{2}(1 - \cos(2\pi x))$ that gives $f(b/2 + I) = b$ for $b \in \{0, 1\}$ and $I \in \mathbb{Z}$.

They call this new bootstrapping BinBoot, and it turns out that this not only bootstraps bits properly but also has some cleaning functionality [BCKS24, Theorem 1].

In [BKSS24], the authors proposed an extension of [BCKS24] to small integers. The general framework is to combine the CGGI-to-CKKS conversion in [BGGJ20] and the look-up table evaluation in [CKKL24]. To elaborate, [BGGJ20] homomorphically computes a complex exponential $x \mapsto e^{2\pi i x}$ to remove the small I introduced during ModRaise by mapping real line into a unit circle. Next, the roots-of-unity look-up table evaluation in [CKKL24] ensures that we can stretch the unit circle back to a real line in a numerically stable manner. It can be described as follows:

$$m \xrightarrow{\mathsf{ModRaise}} m + q_0 I \xrightarrow{x \mapsto e^{2\pi i x}} e^{2\pi i m} \xrightarrow{\mathsf{LUT}} m.$$

The authors of [BKSS24] do not use [BGGJ20] and [CKKL24] as black boxes but introduce further optimizations to improve efficiency. In our work, we just use the fact that there is an efficient integer bootstrapping instantiation that does not require a gap between message and modulus. It could be any instantiation that satisfies the property.

3 Bootstrapping the Most Significant Bits

As noted in Section 2.1, conventional bootstrapping requires a gap between the message and base modulus, for approximating modular reduction. In this section, we revisit the existing (discrete) bootstrapping methods that do not require such a gap and introduce an iterative bootstrapping algorithm that efficiently increases the precision. Furthermore, we propose a bootstrapping that bootstraps real numbers in the most significant bits.

3.1 Revisiting [BKSS24]

We revisit the bootstrapping framework in [BKSS24], which integrated the integer MSB bootstrapping for the first time. For simplicity, we describe the very primitive type of [BKSS24] which only concatenates [BGGJ20] and [CKKL24]. Note that the instantiation of integer MSB bootstrapping can further be optimized by the techniques introduced in [BKSS24]. We refer to [BKSS24] for more details. As we use integer MSB bootstrapping as a black box, any optimized instantiation of it can be used as a component of our circuit. The outline of (primitive) integer MSB bootstrapping can be described as follows.

0. Parameter Setting: The input and output ciphertexts both encrypt integer-encoded messages in slots-encoded state. Each slot of the message belongs

to $[0, t)$ to avoid overflow. We set the bottom-level scaling factor $\Delta_0 = q_0/t$ so that the message is encoded in the most significant bits. Let $\psi : \mathbb{Z}_t = [0, t) \to \mathbb{C}^{N/2}$ be defined as $\psi(x) = e^{2\pi ix/t}$, which is exactly the map that is associated to the roots of unity encoding in [CKKL24].

1. **Slots-to-Coefficients**: We put slots into coefficients.
2. **Modulus-Raising**: We increase the modulus as in the conventional CKKS bootstrapping.
3. **Coefficients-to-Slots**: We put coefficients back to slots.
4. **Homomorphic Exponential**: We evaluate the complex exponential $x \mapsto e^{2\pi ix}$ homomorphically to send integers to roots of unity as in [BGGJ20]. We denote this step as $\mathsf{EvalExp}$.
5. **Homomorphic** ψ^{-1}: We homomorphically evaluate the inverse of ψ in order to send roots of unity back to integers. This step can be denoted as $\mathsf{LUT}_{\psi^{-1}}$.

We denote this bootstrapping as $\mathsf{IntBoot}_t$ and detail the algorithm in Algorithm 1. The correctness of this bootstrapping is directly checked by its components $\mathsf{EvalExp}$ and $\mathsf{LUT}_{\varphi^{-1}}$ which were checked independently in [BGGJ20] and [CKKL24], respectively. As an analogue of $\mathsf{HalfBoot}$ in [CHK+21], we denote $\mathsf{IntBoot}_t$ without the first step StC as $\mathsf{HalfIntBoot}_t$.

```
Algorithm 1: $\mathsf{IntBoot}_t$
  Setting: $\Delta_0 = q_0/t$. $\psi = x \mapsto e^{2\pi ix/t} : \mathbb{Z}_t \to \mathbb{C}$
  Input  : $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}} \circ \mathsf{Ecd}(m) \in \mathcal{R}_{q_0}^2$ a slots-encoded ciphertext where $m$ is a
           vector of integers in $[0, t)$ with small error..
  Output: $\mathsf{ct}_{\mathsf{out}} \in \mathcal{R}_Q^2$.
  $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{LUT}_{\psi^{-1}} \circ \mathsf{EvalExp} \circ \mathsf{CtS} \circ \mathsf{ModRaise} \circ \mathsf{StC}(\mathsf{ct})$;
  return $\mathsf{ct}_{\mathsf{out}}$
```

3.2 Iterative bootstrapping

We construct an iterative bootstrapping based upon $\mathsf{IntBoot}_t$. Note that we are not the first ones to use bootstrapping as a black box and construct higher precision bootstrapping. Similar constructions can be found in [BCC+22, BZP+23].

To briefly describe the outline of the iterative bootstrapping, we extract the least significant digit using $\mathsf{IntBoot}_t$ and repeat this ℓ times to get a bootstrapping for plaintext space of size t^ℓ. The whole bootstrapping is called $\mathsf{IntBoot}_t^\ell$ and the detailed algorithm is given in Algorithm 2.

To be precise, we first start with sufficiently precise StC to put the message in $[0, t^\ell)$ from slots to coefficients. The key observation is that multiplying by $t^{\ell-1}$ extracts the least significant digit. This allows us to use $\mathsf{HalfIntBoot}_t$ as a subroutine, and output a slots-encoded ciphertext that encrypts the least significant digit. We then perform StC and subtract it from the coefficients-encoded original ciphertext, resulting in a ciphertext with one less digit ($t^\ell \to t^{\ell-1}$). This

```
Algorithm 2: $\mathsf{IntBoot}_t^\ell$
Setting: $\Delta_0 = q_0/t^\ell$.
Input  : $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}} \circ \mathsf{Ecd}(m) \in \mathcal{R}_{q_0}^2$ a slots-encoded ciphertext where $m$ is a
         vector of integers in $[0, t^\ell)$.
Output: $\mathsf{ct}_{\mathsf{out}} \in \mathcal{R}_Q^2$.
$\mathsf{ct}' \leftarrow \mathsf{StC}(\mathsf{ct})$;
for $i \leftarrow 0$ to $\ell - 1$ do
    $\mathsf{ct}_i \leftarrow \mathsf{HalfIntBoot}_t(t^{\ell-1-i} \cdot \mathsf{ct}')$;
    if $i \neq \ell - 1$ then
        $\mathsf{ct}' \leftarrow \mathsf{ct}' - t^i \cdot \mathsf{StC}(\mathsf{ct}_i)$;
    end if
end for
$\mathsf{ct}_{\mathsf{out}} \leftarrow \sum_{i=0}^{\ell-1} t^i \cdot \mathsf{ct}_i$;
return $\mathsf{ct}_{\mathsf{out}}$
```

allows us to repeat this until we extract all the digits and have them in the higher modulus. Finally, we can recover the whole message by combining all the digits with a linear combination. Note that $\mathsf{IntBoot}_t^\ell$ requires ℓ many $\mathsf{HalfIntBoot}_t$ and ℓ many StC so it is almost as costly as ℓ many $\mathsf{IntBoot}_t$.

3.3 Bootstrapping reals

Now we generalize $\mathsf{IntBoot}$ to construct bootstrapping for real numbers. The key idea is that if the CKKS ciphertexts with the real (or complex) messages are in a coefficients-encoded state, we can discretize the message by performing modulus switching to t^ℓ for each polynomial in the ciphertexts. Note that the modulus switching operation $\mathsf{ModSwitch}_q^{q'} : \mathcal{R}_q \to \mathcal{R}_{q'}$ can be extended to $\mathcal{R}_q^k \to \mathcal{R}_{q'}^k$ for $k \geq 1$.

```
Algorithm 3: $\mathsf{Boot}_t^\ell$
Setting: $\Delta_0 = q_0$
Input  : $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}} \circ \mathsf{Ecd}(m) \in \mathcal{R}_{q_0}^2$ a slots-encoded ciphertext where $m$ is a
         vector of real numbers in $[0, 1)$.
Output: $\mathsf{ct}_{\mathsf{out}} \in \mathcal{R}_Q^2$.
$\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{HalfIntBoot}_t^\ell \circ \mathsf{ModSwitch}_{t^\ell}^{q_0} \circ \mathsf{ModSwitch}_{q_0}^{t^\ell} \circ \mathsf{StC}(\mathsf{ct})$;
return $\mathsf{ct}_{\mathsf{out}}$
```

Let $\mathsf{ct} \in \mathcal{R}_{q_0}^2$ be a slots-encoded CKKS ciphertext encrypting a plaintext m whose slots are in the interval $[0, 1)$. After StC, we set the bottom level scaling factor $\Delta_0 = q_0$ so that m is now in the coefficients in the most significant bits. Define $\mathsf{discretize}_q^{t^\ell} = \mathsf{ModSwitch}_{t^\ell}^q \circ \mathsf{ModSwitch}_q^{t^\ell} : \mathcal{R}_q \to \mathcal{R}_q$, which discretizes each coefficient of the polynomial as follows:

$$\sum_{i=0}^{N-1} c_i \cdot X^i \mapsto \sum_{i=0}^{N-1} \lfloor \frac{q}{t^\ell} \lfloor \frac{t^\ell}{q} \cdot c_i \rceil \rceil \cdot X^i.$$

For properly chosen $t, \ell \geq 1$, we perform $\mathsf{discretize}_q^{t^\ell}$ to each polynomial in the coefficients-encoded ciphertext[5] which results in discretizing the scaled message with some additional error. More precisely, the output of the discretization is a ciphertext that encrypts $\lfloor t^\ell \cdot \boldsymbol{z} \rceil$ in its coefficients, thereby making the ciphertext deliberately compatible with $\mathsf{HalfIntBoot}_t^\ell$. Specifically, we have

$$[\langle \mathsf{discretize}_q^{t^\ell}(\mathsf{ct}), \mathsf{sk} \rangle]_q = [\mathsf{discretize}_q^{t^\ell}(\langle \mathsf{ct}, \mathsf{sk} \rangle)]_q + e_{dis}$$

where e_{dis} is the discretization error satisfying $\|e_{dis}\|_\infty = O(\sqrt{h})$, with h denotes the hamming weight of the secret key. The error in the ciphertext polynomials after discretization is essentially a rounding error and is therefore $O(1)$. However, when decrypting with a secret key of hamming weight h, the error in the coefficients of the decrypted plaintext can be attributed to the summation of up to h independent rounding errors, each of order $O(1)$. Thus in practice, we can ensure that the discretization error is bounded by $O(\sqrt{h})$. Note that the cost of discretization is negligible, as it is performed at the base level, which contains only one prime.

We then apply $\mathsf{HalfIntBoot}_t^\ell$ to finish our bootstrapping[6]. Since modulus switching is approximately an identity function, this results in a bootstrapping for ct, with precision approximately t^ℓ. We denote this bootstrapping as Boot_t^ℓ, and the detailed algorithm is given in Algorithm 3.

Note that Boot_t^ℓ works up to modulo 1. That is, 0 is identified with 1 and the message near 0 can be bootstrapped to near 1 and vice versa. When constructing homomorphic modular reduction and bit extraction in Section 4, it is okay to have bootstrapping modulo 1. If one needs exact bootstrapping that guarantees correctness without the “up to modulo 1" condition, one can handle this by recovering the error via subtraction and bootstrapping.

3.4 Bit extraction

We revisit Algorithm 2 and 3 to find different versions of $\mathsf{IntBoot}_t^\ell$ and Boot_t^ℓ. In short, we add an extra bit extraction step for each ct_i for $0 \leq i < \ell$.

We first discuss how to extract bits from the roots of unity. Let

$$\iota_j : \{1, \omega, \ldots, \omega^{t-1}\} \to \{0, 1\}$$

[5] A similar idea, to modulus switch and allow some contamination is proposed in [LW24].

[6] Here, $\mathsf{HalfIntBoot}_t^\ell$ needs to be normalized, as the original definition inputs and outputs integers instead of real numbers in $[0, 1)$.

be defined as extracting the jth least significant bit in the exponent, where $0 \leq j < t$. Evaluating LUT_{ι_j} for each j gives all the bits of the roots of unity. Let LUT_{ι} be defined as executing LUT_{ι_j} for all j and outputting t ciphertexts each encrypting bits of the initial ciphertext. We replace $\mathsf{LUT}_{\psi^{-1}}$ of Algorithm 1 with this LUT_{ι} and get the bit extraction version of $\mathsf{IntBoot}_t$. This naturally extends to $\mathsf{IntBoot}_t^{\ell}$ and Boot_t^{ℓ}.

Bit extraction of a ciphertext could be very useful when computing discrete functions, especially when we need to construct a complex discrete function that involves simpler discrete functions as a subroutine. We discuss this in detail in Section 4.

Note that a bit extraction of CKKS ciphertexts has already been proposed in [DMPS24, Algorithm 3]. If we fix t and regard our parameter ℓ as an analogue of N in their algorithm, their algorithm can be described as iterating $O(\ell)$ bit precision homomorphic sign function $O(\ell)$ times. According to [CKK20, Theorem 1], the $O(\ell)$ bit precision sign function of optimal depth consumes $O(\ell)$ multiplicative depths. On the other hand, the depth consumption of $\mathsf{IntBoot}_t$ is independent of ℓ, so our algorithm can be described as evaluating a depth $O(1)$ circuit ℓ times, which is asymptotically better than that of [DMPS24].

4 Modular Reduction

As illustrated in Section 1, modular reduction for RLWE ciphertexts is naturally inherited to coefficients-encoded CKKS ciphertexts. That is, taking modular reduction on ciphertexts gives modular reduction for underlying plaintexts. Once we apply modular reduction, the output no longer has enough space for additional multiplications. Hence, we use the MSB bootstrapping methods from Section 3 to recover the modulus. From now on we will denote q_0 as the base modulus, Q as the bootstrapping output modulus, and q as the arbitrary CKKS modulus such that $q_0 \mid q$ and $q \mid Q$.

4.1 Main Algorithm

For simplicity, we propose an (unsigned) modulo 1 function for CKKS. Modular reduction by arbitrary modulus can be easily constructed with proper constant multiplications.

There are two key observations that make our new homomorphic modular reduction work. The first observation is that it is easy to perform $[\cdot]_{q_0}$ to the coefficients-encoded ciphertext of an arbitrary CKKS modulus. More precise, given a coefficients-encoded ciphertext $\mathsf{ct} \in \mathcal{R}_q^2$ encrypting a message $m \in \mathcal{R}_q$, $[\mathsf{ct}]_{q_0} \in \mathcal{R}_{q_0}^2$ encrypts the message $[m]_{q_0} \in \mathcal{R}_{q_0}^2$. Secondly, we can compute the modulo 1 function $[\cdot]_1$ using $[\cdot]_{q_0}$ with a proper scaling factor. In more detail, for $z \in \mathbb{R}$, $[z]_1 = \frac{1}{q_0} \cdot [q_0 \cdot z]_{q_0} + \epsilon$, where $|\epsilon| \leq \frac{1}{2q_0}$. The second observation can be extended to arbitrary modulus, which allows us to propose modular reduction by an arbitrary positive real number using $[\cdot]_{q_0}$ by adjusting the scaling factor.

```
Algorithm 4: $\mathsf{Mod}_{t^\ell}$ (resp. $\mathsf{Mod'}_{t^\ell}$)
    Setting: $\Delta_{\mathsf{StC}} = q_0$ where $\Delta_{\mathsf{StC}}$ denotes the scaling factor after StC.
    Input  : $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}} \circ \mathsf{Ecd}(\boldsymbol{z}) \in \mathcal{R}_q^2$.
    Output: $\mathsf{ct}_{\mathsf{out}} \in \mathcal{R}_Q^2$ (resp. $(\mathcal{R}_Q^2)^k$).
    $\mathsf{ct}' \leftarrow [\mathsf{StC}(\mathsf{ct})]_{q_0}$;
    $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{HalfBoot}_t^\ell(\mathsf{ct}')$ (resp. $\mathsf{HalfBoot}_t^{\ell'}(\mathsf{ct}')$);
    return $\mathsf{ct}_{\mathsf{out}}$
```

Let $\mathsf{ct} \in \mathcal{R}_q^2$ be a slots-encoded CKKS ciphertext at modulus q, encrypting $\boldsymbol{z} \in \mathbb{C}^{N/2}$. Δ_{StC}, the scaling factor after StC, is set as q_0. We first perform StC, outputting a ciphertext that encrypts (the reordering of) $q_0 \cdot \boldsymbol{z}$ in its coefficients. Next, we take modulo q_0 to ct', outputting a ciphertext encrypting $[\boldsymbol{z}]_1$ in its MSBs.[7] Finally, we use $\mathsf{HalfBoot}_t^\ell$ to raise the modulus from q_0 to Q, while preserving the most significant bits of the message. You may find the details in Algorithm 4.

Note that homomorphic computation of modular reduction has already been extensively studied throughout the literature [CHK$^+$18,LLL$^+$21,JM22,LLK$^+$22] as $\mathsf{EvalMod}$ is a key subroutine of the conventional CKKS bootstrapping. However, such approaches suffer from small and restricted input range, as they rely on polynomial approximations. Indeed, approximating modulo 1 function over an interval $[0, K)$ requires at least $O(K)$ degree polynomial (or equivalent[8]), leading to at least $O(K)$ asymptotic complexity in terms of running time. Furthermore, since modular reduction is a discontinuous function it is extremely expensive to approximate it over the whole interval $[0, K)$. This is one of the reasons why most $\mathsf{EvalMod}$ instantiations approximate modular reduction only near the integer points.

On the other hand, our modular reduction solves both problems. First, the efficiency of our modular reduction is sublinear to the input range, as we utilize the inherent modular reduction functionality of RLWE. Note that increasing the input range only leads to an increase in StC precision, leading to larger modulus consumption during StC. As the running time of StC is proportional to the input modulus size (in bits), the input interval of $[0, K)$ leads to $O(\log K)$ time complexity. Recall that even a few bit target precision (e.g. 2-3 bits) on messages already requires a lot of modulus consumption for StC (e.g. 20 bits per level). In this regard, even if we deal with a large input range like $[0, 2^{20})$ it should not be much more expensive than evaluating over $[0, 1)$. In addition, as we first discretize the data via modulus switching, our method does not suffer from discontinuity issues - we use interpolation rather than approximation.

[7] To be more precise, the entries of $[Re(\boldsymbol{z})]_1$ and $[Im(\boldsymbol{z})]_1$ are properly ordered, creating a vector in $\mathbb{R}^N$. Here we denoted as $\boldsymbol{z}$ for simplicity.

[8] Alternative method in [HMWW24] called $\mathsf{SgnToStep}$ does not directly approximate the target function. Instead, they write step function as a linear combination of sign functions. However, this still has a linear asymptotic complexity $O(K)$.

As a result, the modular reduction should output integers with negligible failure property. Taking into account that the previous polynomial approximation based approaches often output some intermediate results between two consecutive integers (e.g. 3.6 instead of 4), our approach reduces one type of failure probability (defined as the probability that modular reduction outputs non-integer) greatly.

```
Algorithm 5: $\mathsf{IntMod}_{t^\ell}$ (resp. $\mathsf{IntMod}'_{t^\ell}$)
  Setting: $\Delta_{\mathsf{StC}} = q_0/t^\ell$ where $\Delta_{\mathsf{StC}}$ denotes the scaling factor after $\mathsf{StC}$.
  Input  : $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}} \circ \mathsf{Ecd}(\boldsymbol{z}) \in \mathcal{R}_q^2$ where each $z_i$ is a Gaussian integer
  Output: $\mathsf{ct}_{\mathsf{out}} \in \mathcal{R}_Q^2$ (resp. $(\mathcal{R}_Q^2)^k$).
  $\mathsf{ct}' \leftarrow [\mathsf{StC}(\mathsf{ct})]_{q_0}$;
  $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{HalfIntBoot}_t^\ell(\mathsf{ct}')$ (resp. $\mathsf{HalfIntBoot}_t^{\ell\prime}(\mathsf{ct}')$);
  return $\mathsf{ct}_{\mathsf{out}}$
```

As an integer adaptation of Mod_{t^ℓ}, we may also define IntMod_{t^ℓ} that takes modulo t^ℓ for Gaussian integers using $\mathsf{HalfIntBoot}_t^\ell$. See Algorithm 5 for details. By slightly modifying $\mathsf{HalfIntBoot}$, one could generalize it to taking modulo $t_1 \cdots t_\ell$ as well. Instead of performing $\mathsf{HalfIntBoot}_t$ ℓ times, we could perform $\mathsf{HalfIntBoot}_{t_i}$ for $i = 1, ..., \ell$, and define $\mathsf{IntMod}_{\{t_i\}_{i=1,\ldots,\ell}}$ which takes modulo $t_1 ... t_\ell$ for Gaussian integers. This can be useful when computing integer modular reduction for modulus, which has many prime factors. Also note that Mod_{t^ℓ} and IntMod_{t^ℓ} have bit-decomposition versions $\mathsf{Mod'}_{t^\ell}$ and $\mathsf{IntMod}'_{t^\ell}$, respectively.

4.2 Arbitrary Precision

In Section 3, we discussed MSB bootstrapping methods for integers and reals. As long as the ciphertext precision does not exceed the base modulus, these bootstrappings can handle any message with a few iterations. In this section, we extend our framework to arbitrary precision by relying on our modular reduction framework. In particular, we bootstrap arbitrarily large ciphertexts on a fixed MSB bootstrapping parameter set (consisting of Mod_{t^ℓ} and IntMod_{t^ℓ}). As MSB bootstrapping suggested in Section 3 used as a building block of modular reduction in Section 4, the arbitrary precision bootstrapping leads to an arbitrary precision modular reduction.

The extended bootstrapping algorithm is described as follows. The key idea is to first modular reduce the decimal part(s) with Mod_{t^ℓ}, and iteratively bootstrap[9] the remaining part using IntMod_{t^ℓ}, as it can be regarded as a large integer. This bootstrapping is denoted as $\mathsf{Boot}_{t^\ell}^k$ and the detailed algorithm can be found in Algorithm 6. Although we only covered a CKKS ciphertext with $\ell \log t$ bits of decimal part and $(k-1)\ell \log t$ bits of integer part, one can normalize properly to adjust the size of decimal and integer parts.

[9] Again, this follows the philosophy of [BCC$^+$22].

Algorithm 6: $\mathsf{Boot}_{t^\ell}^k$

```
Input  : $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}} \circ \mathsf{Ecd}(\boldsymbol{z}) \in \mathcal{R}_q^2$.
Output: $\mathsf{ct}_{\mathsf{out}} \in \mathcal{R}_Q^2$.
$\mathsf{ct}_0 \leftarrow \mathsf{Mod}_{t^\ell}(\mathsf{ct})$;
$\mathsf{ct}' \leftarrow \mathsf{ct} - \mathsf{ct}_0$;
for $i \leftarrow 1$ to $k-1$ do
    $\mathsf{ct}_i \leftarrow \mathsf{IntMod}_{t^\ell}(\mathsf{ct}')$;
    if $i \neq \ell - 1$ then
        $\mathsf{ct}' \leftarrow \frac{\mathsf{ct}' - \mathsf{ct}_i}{t^\ell}$
    end if
end for
$\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{ct}_0 + \sum_{i=1}^{k-1} t^{(i-1)\ell} \cdot \mathsf{ct}_i$;
return $\mathsf{ct}_{\mathsf{out}}$
```

In addition, we also have a bit-decomposition analogue of $\mathsf{Boot}_{t^\ell}^k$ that outputs all $k\ell \log t$ bits instead of the whole ciphertext. This kind of output format could be useful in evaluating discrete functions, as using bits is often more efficient than using approximations for general real numbers. Note that $\mathsf{Boot}_{t^\ell}^k$ requires one $\mathsf{Mod}_1^{t^l}$ and $k-1$ many IntMod_{t^l}, and each algorithm both require one StC and one $\mathsf{HalfIntBoot}_t^\ell$. Therefore, $\mathsf{Boot}_{t^\ell}^k$ requires k many $\mathsf{HalfIntBoot}_t^l$ and k many StC, which is almost as costly as k many $\mathsf{IntBoot}_t^\ell$ or $k\ell$ many $\mathsf{IntBoot}_t$.

5 Applications

We suggest two applications based on our framework. One application is efficient encrypted computation over modulo integers, which aims to outperform BGV/BFV in certain circumstances. Another application is quantization, which imitates quantization in the context of machine learning, but in a slightly different manner.

5.1 $\mathbb{Z}_n$ arithmetic with CKKS

With efficient homomorphic modular reduction Mod_{t^ℓ}, we can naturally construct a modulo n arithmetic for $n \in \mathbb{Z}_{>0}$ in CKKS. Computation over $\mathbb{Z}_n$ in CKKS has been constructed in [DMPS24, Section 8.1], but it relies on costly bit decomposition suggested in [DMPS24, Section 6], making it extremely inefficient. However, our method utilizes the natural modular reduction of RLWE ciphertexts and costs as much as several iterations of integer bootstrapping in [BKSS24]. Another advantage of our instantiation over the previous works is that our method allows lazy modular reduction. Since the cost of modular reduction depends on the input range in the prior works, they cannot afford lazy modular reduction. However, as our modular reduction has almost the same cost regardless of the input range, we may modular reduce lazily after several multiplications, leading to a huge improvement in terms of efficiency.

It is often considered that the exact schemes such as BGV/BFV are the best option for arithmetic over integers. One may compare our approach with the conventional exact schemes. Long story short, our method can be preferable for the case where n is sufficiently small for an arithmetic over $\mathbb{Z}_n$. When n is sufficiently small, the maximum number of slots for the exact schemes (= the number of distinct factors of a cyclotomic polynomial $\Phi_M(X)$ modulo n) is much smaller than the ring dimension $N = \phi(M)$. On the other hand, our method relies on the CKKS encoding which has full $N/2$ slots (or N slots for coefficients-encoding). Hence, our method should be more efficient in terms of throughput when dealing with relatively small n. Furthermore, our modular reduction does not have to be very precise for a small n, which means that it could be as costly as a few CKKS bootstrappings. Taking into account that BGV/BFV bootstrapping is usually more costly than CKKS bootstrapping, our integer arithmetic would be even more efficient. For the extreme case where $n = 2$, one may refer to [BCKS24] on bits.

5.2 Quantization in CKKS

In the machine learning literature, a technique called quantization [NFA$^+$21] improves efficiency by sacrificing the data size and precision. For instance, instead of using 16 bit floating point arithmetic, one may consider using 4 or 8 bit integer arithmetic, greatly reducing memory footprint and computation cost. As an analogue in FHE, we propose a *quantization* in CKKS.

Recall that our framework (modulus switching trick in particular) provides an efficient way to discretize a message. That is, given a CKKS ciphertext encrypting a real number, we output a ciphertext encrypting an integer or a bit. Computing over discrete space could sometimes be more efficient than computing over approximate space, as we may use interpolation instead of approximation. For instance, evaluation of discontinuous functions is often extremely difficult, because there is no effective way to handle them with low cost. On the other hand, interpolation enables look-up table evaluation (as in [CKKL24]), and its cost is independent of the target function. Hence, we may rely on discretization and interpolation to efficiently tackle the issue of discontinuity. Another advantage of our framework is that the result of the discrete computation is always discrete, while the conventional CKKS scheme suffers from non-discrete results. For instance, the result of homomorphic comparison $a \geq b$ is expected to be 0 or 1, but there are often some intermediate results if a and b are sufficiently close. This stems from the nature of polynomial approximations, and it is somewhat unavoidable. However, since our method starts with discretization, the result should also be discrete as long as we properly handle the deviation through the cleaning framework in [DMPS24].

Note that the notion of quantization in CKKS is slightly different from quantization in the context of machine learning. In machine learning, the main advantage is coming from reducing the data type. For instance, using 4 bit data type instead of 16 could lead to saving 4 times a memory footprint which could lead to a dramatic performance improvement. However, although we discretize a real ciphertext into a discrete ciphertext and reduce precision, the ciphertext size does not change a lot. This is mainly because there is a huge initial cost in FHE. Even if one needs only p bits of precision for small p, scaling factors in CKKS should be much larger than p (e.g. $p + 20$ bits), as the error enlarges through several steps such as canonical embedding or decryption. Instead, the gain is coming from utilizing smaller precision interpolation rather than a large precision approximation.

6 Experiments

We provide some proof-of-concept implementations for our new methods. Our implementation is developed upon the C++ HEaaN library [Cry22]. All the experiments are conducted on an Intel Xeon Gold 6242 at 2.8GHz with 512 GB of RAM running Ubuntu 20.04.5 LTS with a single-threaded CPU. For experiments, we constructed a new CKKS parameter (see Table 4 for details) with 128 bits of security according to the lattice estimator [APS15]. In terms of precision, we used mean (i.e. $-\log_2 \mathbb{E}(\|\boldsymbol{e}\|_1)$) and worst (i.e. $-\log_2 \max\|\boldsymbol{e}\|_\infty$) precision, where $\boldsymbol{e} \in \mathbb{C}^{N/2}$ is the difference between the decryptions (i.e. underlying messages) of input and output ciphertexts. In terms of running time, we experimented > 100 iterations and took an average. In what follows, T_{avg}, P_m, and P_w denote average running time, mean precision, and worst precision, respectively.

We specify the notations used for parameter descriptions (Table 4 and 9) as follows. N denotes the degree of RLWE ciphertexts, h and $\tilde{h}$ denote the hamming weights of dense and sparse secret keys [BTPH22][10], $\log_2(QP)$ denotes the bit length of the maximum RLWE modulus, *dnum* denotes the gadget rank of switching keys, and *depth* denotes the remaining multiplication capacity (i.e. the number of available multiplications) after bootstrapping. In addition, $\log_2(q)$ and $\log_2(p)$ denote the bit length and number of RNS primes used for ciphertext modulus and auxiliary modulus, respectively. For $\log_2(q)$, we indicate the role of primes as Base, StC, Mult, $\mathsf{EvalMod}$, and CtS. A prime list written as $X \times Y$ refers to Y many X bit primes[11].

6.1 Iterative MSB bootstrapping

We implemented our iterative MSB bootstrapping (i.e. $\mathsf{IntBoot}_t$ and Boot_t^ℓ) proposed in Section 3.2 and 3.3 for $t = 32$. We varied the number of iterations (denoted as ℓ) from 1 to 3. For the integer bootstrapping (i.e. $\mathsf{IntBoot}_t$), we bootstrapped normalized integers (in $[0, 1)$) rather than integers (in $\mathbb{Z}$). To be

[10] The sparse secret encapsulation [BTPH22] allows one to use relatively small Hamming weight during $\mathsf{ModRaise}$, reducing the complexity of $\mathsf{EvalMod}$ with negligible overhead. One uses the dense key most of the time, while temporarily using the sparse key to enjoy a small Hamming weight.

[11] Here X bit refers to the integers close to 2^X, instead of the bit size in the usual sense.

Table 4. Description of our parameter for the experiments. See the beginning of Section 6 for the meaning of each notation

	N	$(h, \tilde{h})$	$\log_2(QP)$	$dnum$	$depth$
Param I	2^{16}	$(192, 32)$	1260	5	10

$\log_2(q)$					$\log_2(p)$
Base	StC	Mult	EvalMod	CtS	
42	42×3	42×10	42×8	42×3	42×5

precise, $\mathsf{IntBoot}_t^\ell$ bootstraps a ciphertext encrypting a vector

$$z \in \left\{ \frac{n}{t^\ell} \mid n \in [0, t^\ell) \cap \mathbb{Z} \right\}^{N/2}.$$

Thus, precision refers to the number of bits preserved and counted from the most significant bits of the input message. On the other hand, the real bootstrapping (i.e. Boot_t^ℓ) bootstraps a ciphertext encrypting a vector

$$z \in [0, 1)^{N/2}.$$

Since Boot_t^ℓ works up to modulo 1, we measure errors after taking modulo 1 in this case. The detailed results of the experiment are illustrated in Table 5.

Table 5. Bootstrapping time and precision for $\mathsf{IntBoot}_t^\ell$ and Boot_t^ℓ. Here $t = 32$ and ℓ varies from 1 to 3.

	$\mathsf{IntBoot}_t^\ell$			Boot_t^ℓ		
ℓ	T_{avg} (sec)	P_m	P_w	T_{avg} (sec)	P_m	P_w
1	14.7	30.0	25.7	14.8	3.32	1.00
2	28.9	30.0	25.4	28.7	8.32	5.74
3	42.6	28.5	22.1	43.7	13.3	10.8

The precision of $\mathsf{IntBoot}_t^\ell$ does not depend on ℓ and it is sufficiently high in practice. This precision is dominated by the StC precision. On the other hand, the precision of Boot_t^ℓ is approximately proportional to ℓ increases. Since we discretize the input ciphertext via modulus switching, the precision cannot exceed $\log t^\ell$. In addition, the discretization error reduces the precision, which occurs when we discretize the ciphertext with modulus switching, having a scale of $O(\sqrt{h})$. In these regards, the precision of Boot_t^ℓ is slightly lower than $\log t^\ell$ (about 1 to 2 bits in practice). The running time of both $\mathsf{IntBoot}_t^\ell$ and Boot_t^ℓ

are expected to be (an observed as) ℓ times the $\mathsf{IntBoot}_t$ and Boot_t, respectively, since we perform $\mathsf{halfIntBoot}_t$ ℓ times during the algorithm.

6.2 Modular Reduction

In Section 4, we proposed a new modular reduction that utilizes the inherent modular reduction of RLWE. We experimented Mod_{t^ℓ} and IntMod_{t^ℓ} with the same t as in Section 6.1 using proper variants of $\mathsf{IntBoot}_t^\ell$ and Boot_t^ℓ as subroutines. As the precision depends on the size of the integer part of an input vector $\boldsymbol{z}$, we denote its bit length as $\log_2\lfloor \boldsymbol{z} \rfloor$. The experimental results of IntMod_{t^ℓ}, Mod_{t^ℓ} are illustrated in Table 6, 7 respectively.

The precision of IntMod_{t^ℓ} is limited by the precision of the StC operation. We also observe that the precision drops sharply when $\log_2\lfloor \boldsymbol{z} \rfloor$ varies from 10 to 20. This is because, as $\log_2\lfloor \boldsymbol{z} \rfloor$ increases, the number of the preserved bits below the decimal point decreases, which results in the input incompatible with the $\mathsf{halfIntBoot}$. Since $\mathsf{halfIntBoot}$ utilizes polynomial interpolation instead of polynomial approximation for the $\mathsf{halfIntBoot}$, it outputs completely unexpected value if the inputs are not compatible with the polynomial interpolation it utilizes. Even if this issue occurs in the lower bits of the ciphertext, it can still contaminate the higher bits of the message, leading to a sudden decrease in precision. This also explains why the precision decreases when ℓ increases from 2 to 3, in $\log_2\lfloor \boldsymbol{z} \rfloor = 20$.

Mod_{t^ℓ} has almost same precision with $\mathsf{IntBoot}$, which is linear in ℓ but slightly lower than $\log t^\ell$ due to the modulus switching error. However, the precision is stable with respect to $\log_2\lfloor \boldsymbol{z} \rfloor$, in contrast to IntMod_{t^ℓ}, and doesn't drop sharply when $\log_2\lfloor \boldsymbol{z} \rfloor = 20$. This is based on the fact that the output after discretization is always compatible with the $\mathsf{HalfIntBoot}$, even if it is not correctly discretized. That is, the message of the ciphertext after discretization is compatible with the polynomial interpolation, which results in the message after $\mathsf{HalfIntBoot}$ remains bounded. As a result, even if we lose precision in the lower bits before performing $\mathsf{halfIntBoot}$ due to the precision of the StC operation, it doesn't contaminate the higher bits, in contrast with the IntMod. This is also an important role of discretization: to ensure the input ciphertext compatible with the polynomial interpolation, leading output to be bounded even if it is not correct.

The running time of both IntMod_{t^ℓ} and Mod_{t^ℓ} are observed to be as ℓ times the $\mathsf{HalfIntBoot}_t$, respectively, as we expected in the section 4.1.

6.3 Arbitrary precision

We implemented the arbitrary precision bootstrapping (i.e. $\mathsf{Boot}_{t^\ell}^k$) that we have proposed in section 4.2, using the same t as in section 6.1 and fixing $k = 2$. That is, we first bootstrap the $\log t^\ell$ bits in decimal with some modulus switching error, utilizing Mod_{t^ℓ} as a subroutine, and then bootstrap the $\log t^\ell$ bits of the integer parts with sufficiently small error using IntMod_{t^ℓ}. Therefore, we achieved approximately $k\ell \log t$ bit precision for $\ell \geq 2$. Notably, for $\ell = 3$, we achieved a real bootstrap with 23.3 bit mean precision.

Table 6. Experimental results of IntMod_{t^ℓ}.

	$\ell = 1$			$\ell = 2$			$\ell = 3$		
$\log_2 \lfloor \boldsymbol{z} \rfloor$	T_{avg} (sec)	P_m	P_w	T_{avg} (sec)	P_m	P_w	T_{avg} (sec)	P_m	P_w
0	14.5	30.0	25.1	28.7	30.0	25.2	42.8	28.5	22.9
10	14.8	30.0	25.1	29.2	29.9	25.6	43.3	28.1	23.0
20	15.3	20.1	15.5	29.2	14.9	9.17	45.3	3.24	1.00

Table 7. Experimental results of Mod_{t^ℓ}.

	$\ell = 1$			$\ell = 2$			$\ell = 3$		
$\log_2 \lfloor \boldsymbol{z} \rfloor$	T_{avg} (sec)	P_m	P_w	T_{avg} (sec)	P_m	P_w	T_{avg} (sec)	P_m	P_w
0	14.4	3.32	1.00	28.5	8.32	5.69	43.2	13.3	10.7
10	14.0	3.32	1.00	27.3	8.32	5.73	41.9	13.3	10.8
20	15.1	3.32	1.00	30.1	8.32	5.55	44.9	13.3	10.6

However, for $\ell = 1$, we obtained a precision lower than expected. This occurs because the Mod_{t^ℓ} operation we use to bootstrap the decimal part is not sufficiently precise when $\ell = 1$. As a result, the message obtained after subtracting the result of the Mod_{t^ℓ} from the original ciphertext is not close enough to an integer. Consequently, the input of IntMod_{t^ℓ} is not compatible enough, leading to a significant error during polynomial interpolation.

As we described in section 4.3, $\mathsf{Boot}^k_{t^\ell}$ is observed to be almost as costly as k many $\mathsf{IntBoot}_{t^\ell}$ or $k\ell$ many $\mathsf{IntBoot}_t$ as we expected.

Table 8. Experimental results of $\mathsf{Boot}^k_{t^\ell}$.

	$\ell = 1$			$\ell = 2$			$\ell = 3$		
k	T_{avg} (sec)	P_m	P_w	T_{avg} (sec)	P_m	P_w	T_{avg} (sec)	P_m	P_w
2	29.9	3.58	0.522	58.4	17.8	12.7	86.7	27.8	23.3

6.4 Bit Extraction

In Section 4, we proposed a novel bit decomposition method using our framework. We compare our method with the state-of-the-art bit decomposition introduced in [DMPS24], which utilizes an approximate sign function as a subroutine.

For k-bit extraction, both methods take a ciphertext ct encrypting a message $\boldsymbol{z} \in [0,1)^{N/2}$ as a input. They then output a vector of ciphertexts, $(\mathsf{ct}_i)_{0 \le i < k}$,

where ct_i encrypts the $\boldsymbol{b}_i$, the vector of i-th bit of the $\boldsymbol{z}$ for each slot.
We compare the two methods using two different types of errors. The first error indicates how close the output message is to either 0 or 1. We define

$$P_w^1 = -\log_2 \max\|\boldsymbol{e_{1,i}}\|_\infty,\ P_m^1 = -\log_2 \mathbb{E}(\|\boldsymbol{e}_{1,i}\|_1)$$

where

$$\boldsymbol{e}_{1,i} = \boldsymbol{b}_i - \lfloor \boldsymbol{b}_i \rceil \ (0 \leq i < k).$$

In addition, we define the failure probability p_{fail} as the probability that elements in vector $\boldsymbol{e}_{1,i}$ exceed 2^{-k}. This definition captures the notion of correctness as an error in the MSB exceeding 2^{-k} affects the extraction of the k-th bit, leading to incorrect extraction. The second error indicates how accurately the bit extraction was performed. We define

$$P_w^2 = -\log_2 \max\|\boldsymbol{e}_2\|_\infty,\ P_m^2 = -\log_2 \mathbb{E}(\|\boldsymbol{e}_2\|_1)$$

where

$$\boldsymbol{e}_2 = \boldsymbol{z} - \sum_{0 \leq i < k} \frac{\lfloor \boldsymbol{b}_i \rceil}{2^{i+1}}.$$

Note that we compute the second error using the rounded results of the output vectors, and hence we can ignore the first type error when computing the second type error.

Table 9. Description of a parameter used to implement the bit decomposition in [DMPS24]. See the beginning of Section 6 for the meaning of each notation.

	N	$(h, \tilde{h})$	$\log_2(QP)$	$dnum$	$depth$
Param II	2^{16}	$(256, 32)$	1508	4	10

$\log_2(q)$					$\log_2(p)$
Base	StC	Mult	EvalMod	CtS	
58	34×3	42×10	58×7	56×3	59×6

We compared the two methods with 10-bit decomposition. For [DMPS24], we used a parameter with the same RLWE dimension and multiplicative depths to ensure a fair comparison. More details of the parameter are provided in Table 9. For the choice of homomorphic sign function, we utilized the homomorphic sign function proposed in [CKK20]. Specifically, we used approximated the sign function by composing a degree 7 polynomial

$$f_3(x) = -\frac{5}{16}x^7 + \frac{21}{16}x^5 - \frac{35}{16}x^3 + \frac{35}{16}x$$

Table 10. Performance Comparison of 10-bit decomposition methods.

	T_{avg} (sec)	P_m^1	P_w^1	P_m^2	P_w^2	p_{fail}
Ours	53.8	31.0	25.6	9.46	11.0	$<2^{-15}$
[DMPS24]	385	10.0	23.6	10.0	11.0	1.00%

six times.
Our method is 7.1× faster than the state-of-the-art method [DMPS24] while maintaining precision. Furthermore, while the state-of-the-art method has a 1.00% failure probability due to the approximation errors of the homomorphic sign function, our method does not fail for any slot, indicating a negligible failure probability.

7 Conclusion

In this paper, we proposed a novel homomorphic modular reduction on CKKS both for real and integer messages. We provided experimental results based on an implementation of our method using the C++ HEaaN library [Cry22]. Our approach is asymptotically faster than the existing approaches based on polynomial approximations [CHK$^+$18, LLL$^+$21, JM22, LLK$^+$22, HMWW24]. An additional advantage of our method is that it is less dependent on the input interval compared to existing methods ($O(k)$ versus $O(\log k)$).

While proposing our homomorphic modular reduction method, we introduced iterative MSB bootstrapping for real messages (Boot) and integer messages ($\mathsf{IntBoot}$), thereby completing the MSB bootstrapping framework in CKKS based on other approaches [BCKS24, DMPS24, BKSS24]. The bit extraction version of our MSB bootstrapping method outperforms the state-of-the-art method proposed in [DMPS24] by a factor of 7.1.

We also introduced a discretization method, which discretizes the ciphertext encrypting a real message using modulus switching at the bottom modulus, making general CKKS ciphertext compatible with discrete CKKS. This could allow us to take advantage of discrete CKKS in general situations such as in machine learning, by exploiting features such as look-up tables.

References

[APS15] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 2015. Software available at https://github.com/malb/lattice-estimator.

[BCC$^+$22] Y. Bae, J. H. Cheon, W. Cho, J. Kim, and T. Kim. META-BTS: Bootstrapping precision beyond the limit. In *CCS*, 2022.

[BCKS24] Y. Bae, J. H. Cheon, J. Kim, and D. Stehlé. Bootstrapping bits with CKKS. In *EUROCRYPT*, 2024.

[BGGJ20] C. Boura, N. Gama, M. Georgieva, and D. Jetchev. CHIMERA: Combining ring-LWE-based fully homomorphic encryption schemes. *J. Math. Crypt.*, 2020.

[BGV12] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS*, 2012.

[BKSS24] Y. Bae, J. Kim, D. Stehlé, and E. Suvanto. Bootstrapping small integers with CKKS. In *ASIACRYPT*, 2024.

[Bra12] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *CRYPTO*, 2012.

[BTPH22] J.-P. Bossuat, J. Troncoso-Pastoriza, and J.-P. Hubaux. Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation. In *ACNS*, 2022.

[BZP$^+$23] S. Bian, Z. Zhang, H. Pan, R. Mao, Z. Zhao, Y. Jin, and Z. Guan. HE3DB: An efficient and elastic encrypted database via arithmetic-and-logic fully homomorphic encryption. In *CCS*, 2023.

[CBH$^+$22] T. Chen, H. Bao, S. Huang, L. Dong, B. Jiao, D. Jiang, H. Zhou, J. Li, and F. Wei. The-x: Privacy-preserving transformer inference with homomorphic encryption. arXiv 2206.00216, 2022.

[CCKK24] J. H. Cheon, H. Choe, M. Kang, and J. Kim. Grafting: Complementing RNS in CKKS. Cryptology ePrint Archive, Paper 2024/1014, 2024.

[CCKS23] J. H. Cheon, W. Cho, J. Kim, and D. Stehlé. Homomorphic multiple precision multiplication for CKKS and reduced modulus consumption. In *CCS*, 2023.

[CGGI16] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *ASIACRYPT*, 2016.

[CHK$^+$18] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song. Bootstrapping for approximate homomorphic encryption. In *EUROCRYPT*, 2018.

[CHK$^+$21] J. Cho, J. Ha, S. Kim, B. Lee, J. Lee, J. Lee, S. Moon, and H. Yoon. Transciphering framework for approximate homomorphic encryption. In *ASIACRYPT*, 2021.

[CKK$^+$19] J. H. Cheon, D. Kim, D. Kim, H. H. Lee, and K. Lee. Numerical method for comparison on homomorphically encrypted numbers. In *ASIACRYPT*, 2019.

[CKK20] J. H. Cheon, D. Kim, and D. Kim. Efficient homomorphic comparison methods with optimal complexity. In *ASIACRYPT*, 2020.

[CKKL24] H. Chung, H. Kim, Y.-S. Kim, and Y. Lee. Amortized large look-up table evaluation with multivariate polynomials for homomorphic encryption. IACR eprint 2024/274, 2024.

[CKKS17] J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT*, 2017.

[Cry22] CryptoLab. HEaaN library, 2022. Available at https://heaan.it/.

[DM15] L. Ducas and D. Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In *EUROCRYPT*, 2015.

[DMPS24] N. Drucker, G. Moshkowich, T. Pelleg, and H. Shaul. BLEACH: Cleaning errors in discrete computations over CKKS. *J. Cryptol.*, 2024.

[FV12] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012.

[HMWW24] T. Huang, S. Ma, A. Wang, and X. Wang. Approximate methods for the computation of step functions in homomorphic encryption. Cryptology ePrint Archive, Paper 2024/171, 2024.

[JM22] C. S. Jutla and N. Manohar. Sine series approximation of the mod function for bootstrapping of approximate he. In *EUROCRYPT*, 2022.

[KDE$^+$24] A. Kim, M. Deryabin, J. Eom, R. Choi, Y. Lee, W. Ghang, and D. Yoo. General bootstrapping approach for RLWE-based homomorphic encryption. *IEEE Transactions on Computers*, 2024.

[lat23] Lattigo v5. Online: https://github.com/tuneinsight/lattigo, November 2023. EPFL-LDS, Tune Insight SA.

[LLK$^+$22] J.-W. Lee, Y. Lee, Y.-S. Kim, Y. Kim, J.-S. No, and H. Kang. High-precision bootstrapping for approximate homomorphic encryption by error variance minimization. In *EUROCRYPT*, 2022.

[LLKN22] E. Lee, J.-W. Lee, Y.-S. Kim, and J.-S. No. Optimization of homomorphic comparison algorithm on RNS-CKKS scheme. *IEEE Access*, 2022.

[LLL$^+$21] J.-W. Lee, E. Lee, Y. Lee, Y.-S. Kim, and J.-S. No. High-precision bootstrapping of rns-ckks homomorphic encryption using optimal minimax polynomial approximation and inverse sine function. In *EUROCRYPT*, 2021.

[LLL$^+$22] E. Lee, J.-W. Lee, J. Lee, Y.-S. Kim, Y. Kim, J.-S. No, and W. Choi. Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions. In *ICML*, 2022.

[LLNK22] E. Lee, J.-W. Lee, J.-S. No, and Y.-S. Kim. Minimax approximation of sign function by composite polynomial for homomorphic comparison. *IEEE TDSC*, 2022.

[LW24] Z. Liu and Y. Wang. Relaxed functional bootstrapping: A new perspective on bgv/bfv bootstrapping. Cryptology ePrint Archive, Paper 2024/172, 2024.

[NFA$^+$21] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort. A white paper on neural network quantization. Arxiv 2106.08295, 2021.