Faster Homomorphic Integer Computer

Jaehyung Kim

Stanford University
jaehk@stanford.edu

Abstract. We design a fast and efficient fully homomorphic encryption for radix power modulus (i.e. b^k for a small integer b). We mainly rely on the CKKS modular reduction by Kim and Noh [CiC'25] and the CinS encoding from NeuJeans [Ju *et al.*;CCS'24]. Our construction is a direct improvement of the homomorphic integer computer by Kim [TCHES'25]: The asymptotic latency of homomorphic multiplication reduces from $O(k \log k)$ to $O(\log k)$ for a given plaintext modulus b^k for a fixed radix base b, while keeping the throughput. Our experiments show that the latency of our 64 bit multiplication is ≈ 6 times faster than Kim and slightly faster than TFHE-rs, while being three orders of magnitude better in terms of throughput than TFHE-rs. The performance gap widens for larger precision. For instance, our 512 bit multiplication is ≈ 49 times faster than TFHE-rs in terms of latency and four orders of magnitude better in terms of throughput. Our work also concretely outperforms the work by Boneh and Kim [Crypto'25], by a factor of 4.70 better latency and 75.3 times better throughput for 256 bit multiplication.

Keywords: Fully Homomorphic Encryption · Radix Power Modulus · Integer Computation

1 Introduction

Fully homomorphic encryption (FHE) [Gen09] is a powerful cryptographic primitive that handles computations over encrypted messages. Since integer arithmetic is one of the most fundamental forms of computation, it is an important research question to find an efficient instantiation in the encrypted world. We discuss fully homomorphic encryption schemes that have the following two characteristics:

- **Radix Power Modulus**: Modern computers often support (unsigned) integer computations with different power-of-two moduli such as 2^{32} and 2^{64}. In addition, they provide an extensive list of operations: not only arithmetic operations like addition and multiplication, but also logical operations like comparison and bit-wise operations. We require the fully homomorphic encryption scheme to be compatible with this framework. The most straightforward choice for integer arithmetic, directly using the standard BGV/BFV [BGV12,Bra12,FV12] with respect to a radix power modulus, does not satisfy the requirement.

- **Inherent Bootstrapping**: In all the existing fully homomorphic encryption schemes, the noise accumulates through computations, and we need to bootstrap in order to reduce the noise and allow an unbounded number of computations. We consider schemes that bootstrap after every single operation[1], which is standard in the CGGI/DM family [CGGI16,DM15] but not in the BGV [BGV12] like schemes. We make such an assumption because (1) our construction can be understood as a parallelization of the CGGI/DM bootstrapping and we mainly compare with them, and (2) inherent bootstrapping allows users to construct homomorphic computations without worrying about bootstrapping scheduling, significantly improving usability.

The existing solutions can be classified into two categories. One category is to use CGGI/DM as in TFHE-rs [Zam22], and the other is to use a SIMD-FHE[2] (BGV, BFV, CKKS) as in [Kim25b]. The former works independently on individual data, having a smaller latency. The latter is based on (large-dimension) vectorized operations, achieving better throughput (i.e. amortized latency). A natural question is whether we can have both:

> *Can we design a fully homomorphic encryption scheme for integer computation that is both fast (in terms of latency) and efficient (in terms of throughput)?*

The recent work by Boneh and Kim [BK25] gives an interesting direction by building a nested residue system upon the homomorphic modular reduction from [KN25]. The Chinese Remainder Theorem (CRT) structure allows them to sacrifice the number of slots to get better latency, thereby achieving faster computations than the existing SIMD-FHE solutions. However, it is difficult to use their construction in our scenario. First of all, radix power modulus is not a smooth modulus (i.e. product of coprime small integers), and their method needs to use the general homomorphic modular reduction which is more costly. Second, the homomorphic modular reduction is approximate, and there is no efficient way to obtain an exact modular reduction. This is a big problem when one wants to support a wide range of operations like comparison or lookup tables.

Instead of using CRT as in [BK25], we investigate how to efficiently handle digit-based integer arithmetic within a single RLWE ciphertext. Our starting point is a SIMD homomorphic integer computer in [Kim25b], which naturally supports radix power modulus. Since they allocate one RLWE ciphertext per digit, the latency is proportional to the number of digits and takes hundreds of seconds in a single-threaded CPU even for a moderate precision like 64 bit. To solve this problem, we make two modifications. One is to rely on the CinS encoding in [JPK$^+$24], an intermediate CKKS encoding lying between coefficients and slots encoding, and the other is to adopt an approximate (lazy) modular re-

[1] The scheme might allow some laziness and bootstrap once in every few operations, but our main focus is a strategy to bootstrap every time.

[2] SIMD stands for Single Instruction Multiple Data.

duction as in [BK25]. These modifications allow us to put the entire convolution-style textbook multiplication inside a single RLWE, significantly reducing the asymptotic and concrete latency.

1.1 Technical Overview

Let N be a power-of-two integer and $Q \in \mathbb{Z}_{>0}$. Let $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ and $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R}$. The CinS DFT encoding (introduced in [JPK+24]) can be identified as an isomorphism

$$\mathbb{R}[X]/(X^N + 1) \cong (\mathbb{R}[X]/(X^n + 1))^{N/n}$$

for a power-of-two n. That is, instead of having $N/2$ complex slots [CKKS17], one can use N/n many $\mathbb{R}[X]/(X^n + 1)$ slots, supporting SIMD computations.

Ingredients. We consider a homomorphic arithmetic system over

$$\mathcal{S}_n = (\mathbb{Z}[X]/(X^n + 1))^{N/n} \subseteq (\mathbb{R}[X]/(X^n + 1))^{N/n}.$$

Homomorphic addition and multiplication over $\mathcal{S}_n$ are directly inherited from CKKS as in the discrete CKKS framework [DMPS24]. The modular reduction framework of [KN25] can easily be adapted to $\mathcal{S}_n$, allowing coefficient-wise modular reduction by a small integer b. That is, we have

$$\sum_{i=0}^{n} a_i \cdot X^i \quad \mapsto \quad \sum_{i=0}^{n} [a_i]_b \cdot X^i$$

for each of the N/n slots of $\mathcal{S}_n$. In addition, we consider a constant multiplication during the [KN25] modular reductions, providing a coefficient-wise constant multiplication:

$$\sum_{i=0}^{n} a_i \cdot X^i \quad \mapsto \quad \sum_{i=0}^{n} c_i a_i \cdot X^i$$

for each slot and $(c_i)_{0 \le i < n} \in \mathbb{Z}^n$. In what follows, we assume a black-box homomorphic computer with the following operations: addition, multiplication, modular reduction, and coefficient-wise constant multiplication.

Encoding. Let $k > 0$ be an integer and $m = b^k$ for a small radix b. We encode a $\mathbb{Z}_m$ vector of dimension $\frac{N}{2k}$ into $\mathcal{S}_{2k}$ by representing an integer into base b digits. That is, the encoding map $\mathsf{Encode} : \mathbb{Z}_m^{\frac{N}{2k}} \to \mathcal{S}_{2k}$ is written as a vectorization of

$$\alpha = \overline{\alpha_{k-1} \cdots \alpha_1 \alpha_0}\ (b) \quad \mapsto \quad \sum_{i=0}^{k-1} \alpha_i \cdot X^i \in \mathbb{Z}[X]/(X^{2k} + 1).$$

This encoding is the same as [Kim25b] except that it is placed in the coefficients instead of slots. The decoding map $\mathsf{Decode} : \mathcal{S}_{2k} \to \mathbb{Z}_m^{\frac{N}{2k}}$ is defined as evaluating a polynomial at point b:

$$\sum_{i=0}^{2k-1} \alpha_i \cdot X^i \quad \mapsto \quad \sum_{i=0}^{2k-1} \alpha_i \cdot b^i \mod b^k.$$

Observe that addition and multiplication are native: Given two vectors $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathbb{Z}_m^{\frac{N}{2k}}$, we have

$$\mathsf{Decode}(\mathsf{Encode}(\boldsymbol{\alpha}) + \mathsf{Encode}(\boldsymbol{\beta})) = \boldsymbol{\alpha} + \boldsymbol{\beta}$$

and

$$\mathsf{Decode}(\mathsf{Encode}(\boldsymbol{\alpha}) \cdot \mathsf{Encode}(\boldsymbol{\beta})) = \boldsymbol{\alpha} \cdot \boldsymbol{\beta}.$$

However, these operations are not repeatable because (1) the polynomial degree can exceed $2k$ and overflow, and (2) the coefficient continuously grows. Hence, we need to somehow refresh both the degree and coefficient size of the polynomials, while preserving the underlying message.

Multiplication. Since multiplication is more complicated, we discuss how to construct a repeatable multiplication. Addition follows naturally. Given two integers $\alpha = \overline{\alpha_{k-1} \cdots \alpha_1 \alpha_0}\ (d)$ and $\beta = \overline{\beta_{k-1} \cdots \beta_1 \beta_0}\ (d)$, we encode them as

$$\alpha(X) = \sum_{i=0}^{k-1} \alpha_i \cdot X^i \text{ and } \beta(X) = \sum_{i=0}^{k-1} \beta_i \cdot X^i.$$

As discussed earlier, the product

$$\alpha(X) \cdot \beta(X) = \sum_{i=0}^{2k-2} \gamma_i \cdot X^i$$

already decodes to $\alpha \cdot \beta$ as desired. To reduce the degree back to $k-1$, we multiply a masking vector $(1, 1, \ldots, 1, 0, 0, \ldots, 0) \in \mathbb{Z}^{2k}$ with k 1's and k 0's.

The remaining issue is to modular reduce the representation to the correct digit representation. That is, given a redundantly represented polynomial representation

$$a(X) = \sum_{i=0}^{k-1} a_i \cdot X^i,$$

we need to reduce it to a correct radix b representation

$$\tilde{a}(X) = \sum_{i=0}^{k-1} \tilde{a}_i \cdot X^i$$

such that $a(b) \equiv_m \tilde{a}(b)$ and $0 \leq \tilde{a}_i < b$ for each $0 \leq i < k$. The straightforward approach is to iteratively reduce it from the lowest degree term as in [Kim25b].

However, this solution needs as many modular reductions as in [Kim25b], and the latency remains very high (due to $O(k)$ modular reductions).

To solve the issue, we rely on a similar philosophy as in [BK25], allowing some laziness (i.e. approximate reduction). Instead of requiring $\tilde{a}_i$ to satisfy $0 \leq \tilde{a}_i < b$, we allow representations with $0 \leq \tilde{a}_i < c \cdot b$ for some small constant $c \geq 1$, thereby significantly reducing the number of modular reductions we need. For instance, we consider the case where $b = 2^4$, $k = 16$, and $m = 2^{64}$. If we choose $c = 4$, then the inputs $\alpha(X)$ and $\beta(X)$ satisfy $0 \leq \alpha_i, \beta_i < 4b$. Then we have

$$\gamma_t = \sum_{i=0}^{t} \alpha_i \cdot \beta_{t-i} \leq (4b-1)^2 \cdot (t+1) \leq (4b-1)^2 \cdot k < b^4$$

for each $0 \leq t < k$. The iterative modular reduction from [KN25] allows us to decompose it into four polynomials

$$\Gamma_i(X) = \sum_{t=0}^{k-1} \gamma_{t,i} \cdot X^t \in \mathbb{R}[X]/(X^{2k}+1)$$

for $i = 0, 1, 2, 3$ where $c_t = \sum_{i=0}^{3} c_{t,i} \cdot b^i$ is a digit-d decomposition for each t. Finally, we observe that

$$\Gamma(X) = \Gamma_0(X) + X \cdot \Gamma_1(X) + X^2 \cdot \Gamma_2(X) + X^3 \cdot \Gamma_3(X)$$

gives the correct representation of the product $[\alpha\beta]_m$ where each digit is in $[0, 4b)$. This confirms that the homomorphic multiplication is repeatable.[3]

1.2 Contributions

We propose a novel homomorphic integer computer that improves the state-of-the-art [Kim25b,BK25]. We maintain the advantages of [Kim25b] like versatility and radix power modulus support while achieving better latency than both.

Faster SIMD Integer Computer. The use of the CinS encoding [JPK+24] allows us to support large radix power modulus within a single RLWE ciphertext. In particular, the polynomial ring structure efficiently implements the textbook multiplication written in terms of convolutions. To reduce the number of modular reductions, we adopt the laziness idea of [BK25] and significantly improve the latency. Our approach can be understood as an acceleration of the previous integer computer [Kim25b] by putting all the digits into a single ciphertext. In particular, when achieving modulo b^k computation with fixed [KN25] modulus b, the number of bootstrappings reduces from $O(k \log k)$ to $O(\log k)$. The number of multiplications, which can be a bottleneck for [Kim25b] for a large k, drops from $O(k^2)$ to $O(1)$ since convolution in the intermediate (CinS) space is inherent.

[3] Note that we again need to remove the coefficients of degree higher than m, but we omit here for simplicity.

Better Compatibility. For achieving fast homomorphic multiplication, we may compare our method with [BK25]. They leverage a nested CRT system to support larger precision inside a single RLWE ciphertext, whereas we rely on the CinS encoding and polynomial multiplication. The biggest benefit of our approach over theirs is that our scheme is directly compatible with the usual radix-based integer arithmetic, which [Kim25b] utilized to support various operations (e.g. comparison, bit shift). Recall that there is no efficient way for [BK25] to achieve exact modular reduction or other operations like comparison. In this regard, our scheme can be used as an accelerator for [Kim25b] and enjoy their versatility whenever needed. We describe how to use their framework with respect to our encoding structure in Section 3.2.

Concrete Efficiency We provide some proof-of-concept experimental results that show substantial improvements over the previous works. At a high level, our method is almost as fast as TFHE-rs [Zam22] in terms of latency while being as efficient as [Kim25b] in terms of throughput. The intersection point between ours and [Zam22] is 64 bit multiplication, which means that it is preferable to use ours starting from the standard 64 bit unsigned multiplication. Compared to [BK25] on power-of-two modulus multiplication, our method is significantly better both in terms of latency and throughput, as they need to use the homomorphic modular reduction by a general modulus. In this regard, our work is a new state-of-the-art for standard integer arithmetic.

	prec	latency	throughput
[Kim25b]	64	145 sec	8.85 ms
[BK25]	256	150 sec	4.69 sec
	2048	190 sec	47.5 sec
[Zam22]	64	25.4 sec	25.4 sec
	128	101 sec	101 sec
	256	403 sec	403 sec
Ours	64	24.9 sec	12.2 ms
	128	30.9 sec	30.2 ms
	256	31.9 sec	62.3 ms
	2048	37.7 sec	589 ms

Table 1. Comparison with prior works on homomorphic multiplication. Here, prec denotes the plaintext precision in bits, latency denotes the multiplication execution time, and throughput denotes the amortized latency (i.e. latency divided by the number of slots).

1.3 Related Works

The problem of computing over integers has been extensively studied throughout the literature. The entire BGV [BGV12] and BFV [Bra12,FV12] family is dedicated to supporting integer arithmetic and has been considered as a standard in

general. Given the recent improvements in bootstrapping [GIKV23,MHWW24,KSS24] and generalization [GV25,CHM+25], the BGV/BFV family remains to be the first option to be considered.

When it comes to computations like standard 64 bit unsigned integer arithmetic, the landscape is relatively unclear. There are mainly three aspects that make the BGV/BFV family less preferable. First, the power-of-two modulus cannot provide as much parallelism as NTT-friendly primes. Given an odd integer M, the cyclotomic polynomial $\Phi_M(X)$ splits into $M/\operatorname{ord}_M(2)$ irreducible factors modulo power-of-two [GV23, Section 2.3], which is at most $M/\log_2(M)$. Second, 64 bit is a large modulus that introduces large errors during bootstrapping, which necessitates large parameters like a large RLWE dimension. The recent generalized BFV [CHM+25,GV25] does not seem to help because power-of-two is generally not a cyclotomic modulus. Lastly, the BGV/BFV family is less compatible to digit-level operations like digit decomposition, comparison, and bit shifts, as the primary operations are addition and multiplication.

Due to the aforementioned reasons, the CGGI [CGGI16] and DM [DM15] family has been considered as a good alternative, especially if one prioritizes latency. Based on the powerful functional/programming framework [CJP21,KS22], they represent large integer into smaller digits and enable fast multiplication, as in the open source TFHE-rs [Zam22].

The CKKS scheme [CKKS17] is another alternative that does not natively support integer arithmetic. The discrete variant of CKKS [DMPS24] simulates the small integer computations inside the complex plane $\mathbb{C}$. Notably, the CKKS-style functional bootstrapping [BCKS24,BKSS24,AKP25] gives a SIMD version of the CGGI/DM functional bootstrapping, greatly improving the throughput. Such a bootstrapping, combined with the inherent RLWE modular reduction, gives a modular reduction over $\mathbb{Z}$ as suggested in [KN25]. Given small integer arithmetic and modular reduction, one can build a high-precision integer arithmetic. [Kim25b] explores the digit-style textbook multiplication, whereas [BK25] explores how to embed a CRT system inside a single RLWE ciphertext to provide large integer computations.

One of our key ingredients is the CinS encoding from [JPK+24]. In [JPK+24], they mainly discuss how to use such an encoding to efficiently compute the convolution layer in the context of machine learning. In addition, they propose a conversion from the CinS encoding to other encodings (slots and coefficients encoding) by integrating the CKKS bootstrapping inside the framework. A similar encoding was used in [ZYZ+24] to support large integer arithmetic as in our case. They interpreted the base b digit-style multiplication into polynomial multiplication (i.e. convolution) to handle the large modulus consumption from the naive CKKS-based solution. Nevertheless, their scheme is not bootstrappable and can support only a bounded number of multiplications. In particular, they cannot (1) reduce the noise, (2) recover the modulus, and (3) reduce the digit length. Our scheme can be regarded as a *bootstrappable* version of theirs, bootstrapping all three aspects.

As a final remark, it seems that our construction demonstrates some connection between CKKS and GBFV, as in [Kim25a]. The GBFV-to-CKKS conversion in [Kim25a] implies that the GBFV multiplication can be understood as a convolution between digit-decomposed integers, which looks very similar to our construction. The biggest difference is that we embed such a convolution inside $\mathbb{Z}[X]/(X^{2k}+1)$ whereas GBFV introduces a non-integer plaintext space inside $\mathbb{Z}[X]/\Phi_M(X)$. It is an interesting research question to further investigate the connections.

Concurrent and Subsequent Works. A concurrent work [BFG+25] discusses a GBFV-style scheme for a non-cyclotomic polynomial, supporting a power-of-two modulus. They use a polynomial of the form $X^n - X + 2$ and take the quotient by $X - 2$ so that the plaintext space supports modulo $2^n - 2 + 2 = 2^n$ arithmetic. During bootstrapping, they extract each coefficient, which corresponds to the bit decomposition of an integer, providing logical operations at a cost of bootstrapping. Their bootstrapping bootstraps each bit via the CGGI/DM bootstrapping, and they need k bootstrappings for 2^k modulus. They estimate their homomorphic 64 bit multiplication to have similar performance as TFHE, but they do not provide actual experimental results. We expect that the latency of our homomorphic multiplication would be comparable to theirs, and the throughput will be significantly better.

A subsequent work [CPL25] uses discrete CKKS for integer computations, trying to achieve better latency and functionality than the prior works. Instead of using CinS encoding, they use FFT and slot rotation to encode power-of-two integers, which is mathematically almost equivalent to ours. Their key improvement is an exact modular reduction at an asymptotically the same cost as our approximate modular reduction, which provides significantly better performance for logical operations like comparison and bit-wise operations. Arithmetic operation cost remains almost the same.

2 Preliminaries and Ingredients

In this section, we mainly review the CinS encoding from [JPK+24] and the discrete CKKS framework, and discuss how we combine them to get a CinS version of discrete CKKS. We summarize the main ingredients we would use later in the constructions. Let $N > 0$ be a power-of-two integer and $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ be a power-of-two cyclotomic ring. For $q \in \mathbb{Z}_{>0}$, we let $\mathcal{R}_q = \mathcal{R}/q\mathcal{R} \simeq \mathbb{Z}_q[X]/(X^N + 1)$.

(Partial) Discrete Fourier Transform. We rephrase [JPK+24, Section 2.2] to explain the discrete Fourier transform (DFT) and its decomposition. 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 primitive $2N$-th root of unity $\zeta = e^{\pi i/N} \in \mathbb{C}$. It is identified with a matrix

$$U = \begin{bmatrix} 1 & \zeta_0 & \cdots & \zeta_0^{N-1} \\ 1 & \zeta_1 & \cdots & \zeta_1^{N-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \zeta_{N/2-1} & \cdots & \zeta_{N/2-1}^{N-1} \end{bmatrix} \in \mathbb{C}^{(N/2)\times N}.$$

In particular, the matirx U can be written as $U = [A \mid Ai]$ where

$$A = \begin{bmatrix} 1 & \zeta_0 & \cdots & \zeta_0^{N/2-1} \\ 1 & \zeta_1 & \cdots & \zeta_1^{N/2-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \zeta_{N/2-1} & \cdots & \zeta_{N/2-1}^{N/2-1} \end{bmatrix} \in \mathbb{C}^{(N/2)\times(N/2)}.$$

The Fast Fourier transform (FFT) algorithm states that A is decomposed into a product of butterfly matrices B_k's and a permutation matrix P as

$$A = P \cdot B_{N/2} \cdot B_{N/4} \cdots B_1,$$

where $P \in \mathbb{C}^{(N/2)\times(N/2)}$ is a bit-reverse matrix that satisfies $P^2 = P$,

$$B_k = I_{\frac{N}{4k}} \otimes \begin{bmatrix} I_k & W_k \\ I_k & -W_k \end{bmatrix},$$

and $W_k = \mathsf{diag}(\omega_{8k}^{5^i})_{0\le i<k}$ where ω_m is a primitive m-th root of unity. Let $T_{n/2} := B_{n/4} \cdot B_{n/8} \cdots B_1$ for each power-of-two n. Then we see that the partial (permuted) DFT $T_{n/2}$ is identified with an isomorphism between $\mathbb{C}^{N/2}$ and $(\mathbb{C}[X]/(X^{n/2} - i))^{N/n} \cong (\mathbb{R}[X]/(X^n + 1))^{N/n}$.

CinS Encoding. We modify the definitions and properties in [JPK+24] to better understand the CinS encoding as simulating $\mathbb{R}[X]/(X^n + 1)$ arithmetic in parallel. As mentioned in the previous paragraph, the matrix $T_{n/2}$ can be understood as an isomorphism between $\mathbb{C}^{N/2}$ and $(\mathbb{R}[X]/(X^n+1))^{N/n}$. Combined with the inverse full transform $\mathsf{iDFT} : \mathbb{C}^{N/2} \to \mathbb{R}[X]/(X^N + 1)$, we get our encoding $\mathsf{Ecd}_n : (\mathbb{R}[X]/(X^n + 1))^{N/n} \to \mathcal{R}$ defined as

$$\mathsf{Ecd}_n(\boldsymbol{p}) = \lfloor \Delta \cdot \mathsf{iDFT}(T_{n/2} \cdot \boldsymbol{p}) \rceil$$

where $\Delta > 0$ is a scaling factor. The case $n = N$ is referred to as **coefficients encoding** and the case $n = 2$ is referred to as the **slots encoding**. In addition, we define decoding $\mathsf{Dcd}_n : \mathcal{R} \to (\mathbb{R}[X]/(X^n + 1))^{N/n}$ as its inverse,

$$\mathsf{Dcd}_n(m(X)) = \frac{1}{\Delta} \cdot T_{n/2}^{-1} \cdot \mathsf{DFT}(m(X)).$$

We now look at homomorphic operations with respect to this encoding. Since $\mathsf{iDFT} \circ (\times T_{n/2})$ is a ring isomorphism between $(\mathbb{R}[X]/(X^n + 1))^{N/n}$ and

$\mathbb{R}[X]/(X^N+1)$, we have approximate (scaled) homomorphic addition and multiplication as

$$\mathsf{Ecd}_n(\boldsymbol{p}+\boldsymbol{p}') \approx \mathsf{Ecd}_n(\boldsymbol{p}) + \mathsf{Ecd}_n(\boldsymbol{p}')$$

and

$$\mathsf{Ecd}_n(\boldsymbol{p} \odot \boldsymbol{p}') \approx \frac{1}{\Delta} \cdot \mathsf{Ecd}_n(\boldsymbol{p}) \cdot \mathsf{Ecd}_n(\boldsymbol{p}')$$

where $\odot$ denotes the element-wise polynomial multiplication. These homomorphic operations immediately translate to ciphertext operations, exactly the same as in the original CKKS [CKKS17].

The original CKKS bootstrapping [CHK+18] involves conversions between slots and coefficients encoding, defined as homomorphic evaluations of discrete Fourier transforms. In particular, the homomorphic evaluations of DFT and iDFT give conversion from slots encoding to coefficients encoding and vice versa, respectively. [JPK+24] generalizes these conversions to the CinS encoding. That is, homomorphically evaluating $\times T_{n/2}$ can be identified as a conversion from the CinS encoding to the coefficients encoding. In particular, the original slots-to-coefficients (StC) transformation can be decomposed into two parts StC_1 and StC_2 so that $\mathsf{StC} = \mathsf{StC}_2 \circ \mathsf{StC}_1$ and StC_2 is a homomorphic evaluation of $\times T_{n/2}$. Given this, one can start from the slots encoding, apply StC_1 to reach the CinS encoding, and apply StC_2 to reach the coefficients encoding.

Discrete CKKS. In [DMPS24], they suggested using CKKS for discrete computations. The main idea is to choose a discrete (or finite) subset of $\mathbb{C}$ and choose messages from the subspace. This allows us to distinguish the underlying error from the message, providing an exact FHE scheme, as in the FHE schemes other than CKKS. Since then, many works [CKKL24,BCKS24,BKSS24,AKP25,KN25] significantly improved the framework. One of the most important components is discrete bootstrapping [BKSS24,AKP25], which not only raises the modulus but also evaluates an arbitrary function as in the functional/programmable bootstrapping framework in CGGI/DM [CJP21,KS22]. [KN25] observed that leveraging the RLWE modular reduction and the identity discrete bootstrapping can give homomorphic modular reduction for discrete CKKS. Upon the modular reduction framework, [Kim25b,KN25] has built efficient homomorphic integer computation systems.

We mainly rely on the homomorphic modular reduction from [KN25], which is explained in Algorithm 1. Note that IntMod_d naturally extends to modular-reducing over the space $(\mathbb{Z}[X]/(X^n+1))^{N/n}$. Such a variant can be instantiated by replacing

$$\mathsf{LUT}_{\mathsf{id}} \circ \mathsf{EvalExp} \circ \mathsf{CtS} \circ \mathsf{ModRaise} \circ [\mathsf{StC}(\mathsf{ct})]_{q_0}$$

with

$$\mathsf{StC}_1 \circ \mathsf{LUT}_{\mathsf{id}} \circ \mathsf{EvalExp} \circ \mathsf{CtS} \circ \mathsf{ModRaise} \circ [\mathsf{StC}_2(\mathsf{ct})]_{q_0}.$$

We denote it as IntMod_d^*.

Given this observation, we consider an adapted version of the discrete CKKS framework to the message space $(\mathbb{Z}[X]/(X^n+1))^{N/n}$. It supports the following list of operations:

Algorithm 1: IntMod_d [KN25]

Setting: LUT a univariate interpolation from $(e^{2\pi ik/d})_{0\le k<d}$ to $(k)_{0\le k<d}$, $\mathsf{EvalExp}$ homomorphic evaluation(s) of a complex exponential $x \mapsto e^{2\pi ix}$, StC and CtS homomorphic evaluation of DFT and iDFT, respectively, $\mathsf{ModRaise}$ modulus raising from modulus q_0 to a large modulus, and the base level scaling factor $\Delta_0 = q_0/d$.

Input : $\mathsf{ct} \in \mathcal{R}_Q^2$ a discrete CKKS ciphertext encrypting $\boldsymbol{z} \in \mathbb{Z}^N$. We consider using all the slots by using Gaussian integers.

Output: $\mathsf{ct}_{\mathsf{out}}$ encrypting $[\boldsymbol{z}]_d$.

1 $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{LUT}_{\mathsf{id}} \circ \mathsf{EvalExp} \circ \mathsf{CtS} \circ \mathsf{ModRaise} \circ [\mathsf{StC}(\mathsf{ct})]_{q_0}$;

2 **return** $\mathsf{ct}_{\mathsf{out}}$.

1. **Arithmetic Operations**: As mentioned in the CinS preliminary, the ring isomorphism between $\mathbb{R}[X]/(X^N+1)$ and $(\mathbb{R}[X]/(X^n+1))^{N/n}$ gives homomorphic addition and multiplication. We then restrict the message space to
$$\mathcal{S}_n = (\mathbb{Z}[X]/(X^n+1))^{N/n} \subseteq (\mathbb{R}[X]/(X^n+1))^{N/n},$$
just as in the discrete CKKS framework. This gives elementwise addition and multiplication over $\mathcal{S}_n$.
2. **Modular Reduction**: As discussed in the previous paragraph, we adapt the homomorphic modular reduction from [KN25] by a small integer $d > 0$ to $(\mathbb{Z}[X]/(X^n+1))^{N/n}$. This operation, denoted as IntMod_d^*, provides an element-wise coefficient-wise modular reduction. That is, the underlying message is mapped as
$$\left(\sum_{j=0}^{n-1} a_{ij}\cdot X^j\right)_{0\le i<N/n} \mapsto \left(\sum_{j=0}^{n-1} [a_{ij}]_d \cdot X^j\right)_{0\le i<N/n},$$
where $[\cdot]_d : \mathbb{Z} \to \mathbb{Z}_d$ is a modular reduction by d according to a canonical representative (e.g. $\{0,1,\dots,d-1\} \subseteq \mathbb{Z}$).
3. **Coefficient-wise Multiplication:** The multiplication over $\mathcal{S}_n$ gives a polynomial multiplication over $\mathbb{Z}[X]/(X^n+1)$. Instead of using the discrete bootstrapping framework as a black box, we observe that we can use the original CKKS slots encoding. That is, IntMod_d^* without the last StC_1 can be identified with converting a CinS encoding ciphertext into a CKKS encoded ciphertext, bridging $\mathcal{S}_n = (\mathbb{Z}[X]/(X^n+1))^{N/n}$ with $\mathbb{Z}^N$. In this case, every coefficient of each polynomial in slots of $\mathcal{S}_n$ becomes each slot of $\mathbb{Z}^N$. If we perform multiplication in the CKKS side, it translates to coefficient-wise multiplication. That is, we have
$$\left(\sum_{j=0}^{n} a_{ij}\cdot X^j\right), \left(\sum_{j=0}^{n} a'_{ij}\cdot X^j\right) \mapsto \left(\sum_{j=0}^{n} a_{ij}a'_{ij}\cdot X^j\right).$$

In the later sections, we assume that we have a black box homomorphic computer over $\mathcal{S}_n$ with the mentioned properties.

3 Our Construction

In this section, we design a fully homomorphic encryption scheme that efficiently supports radix power modulus. The arithmetic of our scheme depends on the standard radix-based integer arithmetic as in [Kim25b]. We introduce two modifications to accelerate [Kim25b] in terms of latency. One modification is to use the CinS encoding as in [JPK+24], and the other one is to use a redundant radix encoding that allows digits to be possibly greater than the base b.

3.1 Scheme Description

Let $b \in \mathbb{Z}_{>0}$ be the radix base and $k \in \mathbb{Z}_{>0}$ be a power-of-two integer so that b^k is the plaintext modulus. The message space of our scheme is $\mathbb{Z}_{b^k}^{\frac{N}{2k}}$. At a high level, we embed $\mathbb{Z}_{b^k}^{\frac{N}{2k}}$ inside $(\mathbb{R}[X]/(X^{2k}+1))^{\frac{N}{2k}}$, use polynomial multiplication to simulate integer multiplication, and remove the carry digits via masking.

Encoding. Let $\boldsymbol{z} = (z_0, z_1, \ldots, z_{\frac{N}{2k}-1}) \in \mathbb{Z}_{b^k}^{\frac{N}{2k}}$ be a vector of integers. Let z_i be decomposed as

$$z_i = \sum_{j=0}^{k-1} z_{ij} \cdot b^j$$

via the based-b digit decomposition, so that $0 \le z_{ij} < b$ for each $0 \le j < k$. We first encode the vector into an integer polynomial vector

$$\left(\sum_{j=0}^{k-1} z_{ij} \cdot X^j\right)_{0 \le i < \frac{N}{2k}} \in \left(\mathbb{R}[X]/(X^{2k}+1)\right)^{\frac{N}{2k}},$$

scale up by a scaling factor $\Delta > 0$, and round to the nearest integer polynomials. That is, the encoding of $\boldsymbol{z}$ is defined as

$$\mathsf{RxEcd}(\boldsymbol{z}) = \mathsf{Ecd}_{2k}\left(\left(\sum_{j=0}^{k-1} z_{ij} \cdot X^j\right)_{0 \le i < \frac{N}{2k}}\right) \in \mathcal{R}.$$

Decoding. Decoding is just the reverse order of encoding. Given a polynomial $m(X) \in \mathcal{R}$, the decoding of $m(X)$ is defined as

$$\mathsf{RxDcd}(m(X)) = \lfloor \mathsf{Dcd}_{2k}(m(X)) \rceil (b)$$

which is to evaluate on b after performing CinS decoding and rounding. Here, rounding $\lfloor \cdot \rceil$ means rounding each coefficient to the nearest integer. Note that we allow plaintexts (i.e. decrypted ciphertexts in $\mathcal{R}$) to have redundant digit representation and degree higher than $k-1$. That is, After applying Dcd_{2k}, the resulting polynomial vector

$$\left(\sum_{j=0}^{2k-1} z_{ij} \cdot X^j \right)_{0 \le i < \frac{N}{2k}}$$

satisfies $0 \le z_{ij} < \ell$ for some $\ell \in \mathbb{Z}_{>0}$, possibly having $\ell > d$. Even in such a case, our decoding is well defined because we simply evaluate polynomials at the point b. The terms of degree higher than $k-1$ are evaluated as multiples of b^k, which are identified as 0.

Theorem 1 (Encoding Correctness). *Let $\boldsymbol{z} \in \mathbb{Z}_{b^k}^{\frac{N}{2k}}$. Then we have*

$$\mathsf{RxDcd}(\mathsf{RxEcd}(\boldsymbol{z})) = \boldsymbol{z},$$

assuming that the scaling factor $\Delta \gg 1$.

Proof. By the definitions of encoding and decoding, we have

$$\begin{aligned}
\mathsf{RxDcd}(\mathsf{RxEcd}(\boldsymbol{z})) &= \left\lfloor \mathsf{Dcd}_{2k} \circ \mathsf{Ecd}_{2k} \left(\sum_{j=0}^{k-1} z_{ij} \cdot X^j \right)_{0 \le i < \frac{N}{2k}} \right\rceil (b) \\
&= \left\lfloor \frac{1}{\Delta} \cdot T_k^{-1} \cdot \mathsf{DFT} \left(\left\lfloor \Delta \cdot \mathsf{iDFT} \left(T_k \cdot \left(\sum_{i=0}^{k-1} z_{ij} \cdot X^j \right)_{0 \le i < \frac{N}{2k}} \right) \right\rceil \right) \right\rceil (b) \\
&= \left\lfloor \left(\sum_{i=0}^{k-1} z_{ij} \cdot X^j \right)_{0 \le i < \frac{N}{2k}} + \frac{1}{\Delta} \cdot T_k^{-1} \cdot \mathsf{DFT}(e) \right\rceil (b)
\end{aligned}$$

where $e \in \mathbb{R}[X]/(X^N+1)$ is a rounding error such that $\|e\|_\infty < 1/2$. Since the size of $T_k^{-1} \cdot \mathsf{DFT}(e)$ does not depend on Δ, we can set Δ to be sufficiently large so that the error terms disappear at rounding. In this case,

$$\begin{aligned}
\mathsf{RxDcd}(\mathsf{RxEcd}(\boldsymbol{z})) &= \left(\sum_{i=0}^{k-1} z_{ij} \cdot X^j \right)_{0 \le i < \frac{N}{2k}} (b) \\
&= \left(\sum_{i=0}^{k-1} z_{ij} \cdot b^j \right)_{0 \le i < \frac{N}{2k}} = (z_i)_{0 \le i < \frac{N}{2k}} = \boldsymbol{z}.
\end{aligned}$$

This finishes the proof. □

Encryption and Decryption. Given an encoded plaintext in $\mathcal{R}$, we use the usual RLWE encryption and decryption. That is, given a secret key $s \in \mathcal{R}$ and a modulus $q \in \mathbb{Z}_{>0}$, the secret key encryption of a plaintext $m(X)$ is defined as

$$\mathsf{Enc}_s(m(X)) = (-as + m + e, a) \in \mathcal{R}_q^2$$

where $a \in \mathcal{R}_q$ is sampled uniformly and $e \in \mathcal{R}_q$ is sampled from a small distribution (e.g. discrete Gaussian). The decryption is an inner product with the secret key:

$$\mathsf{Dec}_s(b, a) = (b, a) \cdot (1, s) = b + as \in \mathcal{R}.$$

Given a ciphertext ct, its plaintext $m(X)$ which is an element of $\mathcal{R}$ is identified with a polynomial vector $\mathbf{p}$ in $(\mathbb{Z}[X]/(X^{2k}+1))^{\frac{N}{2k}}$ via Dcd_{2k} and round. We call $\mathbf{p}$ an underlying polynomial vector or plaintext polynomial vector of ct, and say that ct is an encryption of $\mathbf{p}$. The original element in $\mathbb{Z}_{b^k}^{\frac{N}{2k}}$ is called the (underlying) message.

(Approximate) Reduction. As in [Kim25b], we introduce a modular reduction that allows one to reduce the size of each coefficient of the plaintext polynomial vector in $(\mathbb{Z}[X]/(X^{2k}+1))^{\frac{N}{2k}}$. Unlike [Kim25b], we lazily reduce, which means that we allow the resulting coefficient to be greater than b. Such a laziness gives us better latency than the full reduction in [Kim25b]. Given a ciphertext ct, the reduction operation with t iterations is defined as in Algorithm 2.

Algorithm 2: Reduce

Setting: Iteration number $0 < t < k$.
Input : $\mathsf{ct} \in \mathcal{R}_Q^2$ a ciphertext encrypting a message $\boldsymbol{z}$ whose polynomial vector is $\mathbf{p} = (p_i(X))_{0 \le i < \frac{N}{2k}} \in (\mathbb{Z}[X]/(X^{2k}+1))^{\frac{N}{2k}}$.
Output: $\mathsf{ct}_{\mathsf{out}}$ encrypting the same message $\boldsymbol{z} \in \mathbb{Z}_{b^k}^{\frac{N}{2k}}$ but with a smaller plaintext polynomial vector.

$\mathsf{ct}' \leftarrow \mathsf{ct}$;
$\mathsf{ct}_{\mathsf{out}} \leftarrow 0$;
for $j \leftarrow 0$ **to** $t-1$ **do**
  $\mathsf{ct}_j \leftarrow \mathsf{IntMod}_b^*(\mathsf{ct}')$;
  $\mathsf{ct}' \leftarrow (\mathsf{ct}' - \mathsf{ct}_j)/b$;
  $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{ct}_{\mathsf{out}} + (X^j)_{0 \le i < \frac{N}{2k}} \cdot \mathsf{ct}_j$;
end for
return $\mathsf{Reduce}(\mathsf{ct}) := \mathsf{ct}_{\mathsf{out}}$.

The key idea of the algorithm is that the monomial X in the digit decomposed representation $\mathbb{Z}[X]/(X^{2k}+1)$ corresponds to the base b. Hence, we iteratively extract the jth digit of ct and store it in ct_j. To get the reduced result, we multiply ct_j with the monomial X^j (for every coordinate) and sum them all, getting an approximate modular reduction.

Theorem 2 (Reduction). *Let* ct *be a ciphertext encrypting a message* $\boldsymbol{z} \in \mathbb{Z}_{b^k}^{\frac{N}{2k}}$ *whose underlying polynomial vector* $\mathbf{p} \in (\mathbb{Z}[X]/(X^{2k}+1))^{\frac{N}{2k}}$ *has degree* $\leq k-1$ *and coefficients in* $[0,\ell)$*, where* $\ell \leq b^t$*. Then the reduction algorithm in Algorithm 2 with* t *iterations gives a ciphertext encrypting the same message* $\boldsymbol{z}$ *whose underlying plaintext polynomial vector has coefficients in* $[0, tb)$*.*

Proof. We first prove that the resulting ciphertext encrypts the same message $\boldsymbol{z}$. We observe that ct_j iteratively extracts the lowest digit, which means that ct_j encrypts a polynomial vector whose coefficients are the jth digits of the initial polynomial vector $\mathbf{p}$. Let $\boldsymbol{z}_j$ be the underlying message of ct_j. Then we have

$$\boldsymbol{z} = \sum_{j=0}^{t-1} b^j \cdot \boldsymbol{z}_j.$$

To check the underlying message that $\mathsf{ct}_{\mathsf{out}}$ encrypts, we use the additive homomorphic property of our encoding (and encryption). We observe that $(X^j)_{0 \leq i < \frac{N}{2k}}$ is a polynomial vector that decodes to a message b^j, which means that $\mathsf{ct}_{\mathsf{out}}$ is encrypting a message

$$\sum_{j=0}^{t-1} b^j \cdot \boldsymbol{z}_j = \boldsymbol{z}$$

as desired.

Next, we prove the upper bound for coefficients of the polynomial vector that $\mathsf{ct}_{\mathsf{out}}$ encrypts. For each $0 \leq j < t$, we check that the coefficients of the underlying polynomial vectors of $(X^j)_{0 \leq i < \frac{N}{2k}} \cdot \mathsf{ct}_j$ belong to $[0, b)$. Since $\mathsf{ct}_{\mathsf{out}}$ sums all of them, the coefficients of the underlying polynomial vector belong to $[0, tb)$. This finishes the proof. □

Multiplication. Homomorphic multiplication can be defined using the polynomial multiplication over $\mathbb{Z}[X]/(X^{2k}+1)$, masking over CKKS slots, and the homomorphic reduction. Given two k-digit integers encoded as polynomials in $\mathbb{Z}[X]/(X^{2k}+1)$, they have degree $\leq k-1$ by definition. Once we multiply these two polynomials, we get a polynomial of degree $\leq 2k-2$ without introducing any overflow. To modular reduce by b^k, we zero out all the coefficients with degree greater than or equal to k. This can be instantiated by muliplying the mask vector filled with 0 or 1 in the CKKS side. In the meantime, we need to reduce the coefficients so that we can repeat multiplication. A straightforward approach is to first zero out and then reduce afterwards, but we fuse them to reduce the number of bootstrappings. The detailed algorithm is described in Algorithm 3.

Theorem 3 (Multiplication). *Let* $\mathsf{ct}, \mathsf{ct}' \in \mathcal{R}_Q^2$ *be ciphertexts encrypting messages* $\boldsymbol{z}, \boldsymbol{z}' \in \mathbb{Z}_{b^k}^{\frac{N}{2k}}$ *such that the underlying polynomial vectors* $\mathbf{p}, \mathbf{p}' \in (\mathbb{Z}[X]/(X^{2k}+1))^{\frac{N}{2k}}$ *have coefficients in* $[0,\ell)$*. Then the product* $\mathsf{Mult}(\mathsf{ct}, \mathsf{ct}')$ *as defined in Algorithm 3 (with iteration number* t *such that* $k \cdot \ell^2 \leq b^t$*) encrypts a message* $\boldsymbol{z} \odot \boldsymbol{z}' \in \mathbb{Z}_{b^k}^{\frac{N}{2k}}$ *and the underlying polynomial vector* $\mathbf{p}_{\mathsf{out}}$ *has coefficients in* $[0, tb)$*.*

Algorithm 3: Mult

Setting: Homomorphic modular reduction IntMod_b^* parameters as defined in Algorithm 1. A homomorphic masking Mask_j for each $0 \le j < t$ that zeros out the degree $\ge k$ and $< j$ terms in the space $(\mathbb{Z}[X]/(X^{2k}+1))^{\frac{N}{2k}}$ by multiplying appropriate masking vectors in the CKKS side. Iteration number $0 < t < k$.

Input : $\mathsf{ct}, \mathsf{ct}' \in \mathcal{R}_Q^2$ ciphertexts encrypting messages $\boldsymbol{z}, \boldsymbol{z}' \in \mathbb{Z}_{b^k}^{\frac{N}{2k}}$, respectively. In addition, the underlying polynomial vectors $\mathbf{p}$ and $\mathbf{p}'$ have coefficients in $[0, \ell)$.

Output: $\mathsf{ct}_{\mathsf{out}}$ encrypting the message $\boldsymbol{z} \odot \boldsymbol{z}' \in \mathbb{Z}_{b^k}^{\frac{N}{2k}}$.

1 $\mathsf{tmp} \leftarrow \mathsf{ct} \cdot \mathsf{ct}'$;
2 $\mathsf{ct}_{\mathsf{out}} \leftarrow 0$;
3 **for** $j \leftarrow 0$ **to** $t-1$ **do**
4 $\quad \mathsf{ct}_j \leftarrow (X^j)_{0 \le i < \frac{N}{2k}} \cdot \mathsf{tmp}$;
5 $\quad \mathsf{ct}_j \leftarrow \mathsf{StC}_1 \circ \mathsf{Mask}_j \circ \mathsf{LUT}_{\mathsf{id}} \circ \mathsf{EvalExp} \circ \mathsf{CtS} \circ \mathsf{ModRaise} \circ [\mathsf{StC}_2(\mathsf{ct}_j)]_{q_0}$;
6 $\quad \mathsf{tmp} \leftarrow (\mathsf{tmp} - (X^{-j})_{0 \le i < \frac{N}{2k}} \cdot \mathsf{ct}_j)/b$;
7 $\quad \mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{ct}_{\mathsf{out}} + \mathsf{ct}_j$;
8 **end for**
9 **return** $\mathsf{Mult}(\mathsf{ct}, \mathsf{ct}') := \mathsf{ct}_{\mathsf{out}}$.

Proof. We can divide the algorithm into two parts: a usual RLWE homomorphic multiplication and a reduction. First, we observe that $\mathsf{ct} \cdot \mathsf{ct}'$ gives a polynomial representation of the integer vector $\boldsymbol{z} \odot \boldsymbol{z}' \in \mathbb{Z}^{\frac{N}{2k}}$ as a vector over $\mathbb{Z}$ (instead of $\mathbb{Z}_{b^k}$). In other words, the decryption of $\mathsf{ct} \cdot \mathsf{ct}'$ can be identified with a polynomial vector

$$\mathbf{p}'' = \left(\sum_{j=0}^{2k-2} z''_{ij} \cdot X^j \right)_{0 \le i < \frac{N}{2k}}$$

such that $\mathbf{p}''(b) = \boldsymbol{z} \odot \boldsymbol{z}'$. This comes from the fact that replacing X with b after polynomial multiplication simulates digit-style multiplication, and that there is no overflow as $(k-1)+(k-1)$ (i.e. the maximum degree) is less than $2k$.

Next, we check that the remaining part reduces correctly. That is, we need to check that (1) the result equals $\boldsymbol{z} \odot \boldsymbol{z}'$ modulo b^k and (2) the coefficients of the output polynomial vectors belong to $[0, tb)$. At step 0, Line 4,5 of Algorithm 3 gets the remainder mod b for all the coefficients. In Line 6, tmp stores the quotient of the division by b. At step $j \ge 1$, Line 4,5 shifts the j-th digit by j and removes all the coefficients that need to be removed. That is, all the degree $\ge k$ coefficients and the coefficients that wrapped around (i.e. degree $< j$) are zeroed out so that it stores the correct modulo b^k multiplication. The iteration repeatedly extracts the lowest digits so that ct_j can properly store the j-th digit. As a result, $\mathsf{ct}_{\mathsf{out}}$, which is a summation of all the ct_j's, has the correct result as desired. □

We add a minor remark on Mask_j function in Algorithm 3. The degree $\ge k$ terms of the space $\mathbb{R}[X]/(X^{2k}+1)$ are identified with imaginary parts during

CKKS bootstrapping, and removing them does not need masking and is almost free: We can just omit the $\mathsf{EvalExp}$ for the imaginary part. In terms of efficiency, this allows us to reduce the number of $\mathsf{EvalExp}$ from 2 (= real and imaginary) to 1 (= real).

Example 1 (64 bit multiplication). We may choose $b = 16$, $k = 16$, $\ell = 64$, $t = 4$ which satisfy the condition in Theorem 3. Such a choice allows repeated multiplication, as the input range $[0, \ell)$ and the output range $[0, tb)$ are equal.

Addition. Homomorphic addition is much simpler than multiplication. One may simply add ciphertexts and reduce afterwards if needed.

Theorem 4 (Addition). *Let* $\mathsf{ct}, \mathsf{ct}' \in \mathcal{R}_Q^2$ *be ciphertexts encrypting messages* $\boldsymbol{z}, \boldsymbol{z}' \in \mathbb{Z}_{b^k}^{\frac{N}{2k}}$ *such that the underlying polynomial vectors* $\mathbf{p}, \mathbf{p}' \in (\mathbb{Z}[X]/(X^{2k}+1))^{\frac{N}{2k}}$ *have coefficients in* $[0, \ell)$*, where* $2\ell \leq b^t$*. The addition of* ct *and* ct'*, defined as*

$$\mathsf{Add}(\mathsf{ct}, \mathsf{ct}') := \mathsf{Reduce}(\mathsf{ct} + \mathsf{ct}')$$

with reduction iteration number t *encrypts a message* $\boldsymbol{z} + \boldsymbol{z}' \in \mathbb{Z}_{b^k}^{\frac{N}{2k}}$ *and the underlying polynomial vector has coefficients in* $[0, tb)$*.*

Proof. The fact that $\mathsf{Add}(\mathsf{ct}, \mathsf{ct}')$ encrypts the message $\boldsymbol{z}+\boldsymbol{z}'$ comes directly from the encoding structure and the correctness of reduction. It remains to check the coefficient bound. After adding the two polynomial vectors, each coefficient lives in $[0, 2\ell)$. According to the reduction algorithm in Algorithm 2, we can reduce the ciphertext with t iterations, leading to coefficients in $[0, tb)$. This finishes the proof. □

Note that one would often use lazy addition (as in [Kim25b, Theorem 8]) to reduce the number of reductions. That is, addition of u ciphertexts $\mathsf{ct}_0, \mathsf{ct}_1, \ldots, \mathsf{ct}_{u-1}$ can be instantiated with

$$\mathsf{Add}(\mathsf{ct}_0, \mathsf{ct}_1, \ldots, \mathsf{ct}_{u-1}) = \mathsf{Reduce}\left(\sum_{i=0}^{u-1} \mathsf{ct}_i\right).$$

Since the coefficient bound $[0, \ell)$ for addition grows much slower than multiplication (additive v.s. multiplicative), this significantly reduces the cost of additions.[4]

Example 2 (64 bit addition). We choose $b = 16$, $k = 16$, $\ell = 64$ as we did in Example 1. According to Theorem 9, we can choose $t = 2$ and satisfy $2\ell \leq b^t$. If we use lazy addition and choose $t = 3$, the requirement becomes $2u\ell \leq b^t$, and it can afford at most $u = b^t/(2\ell) = 32$ additions with $t = 3$ bootstrappings.

[4] This further extends to a combination of additions and multiplications, such as a homomorphic inner product.

3.2 Analysis

We analyze our construction in terms of functionality and efficiency. First, we check the compatibility between our scheme and [Kim25b] and discuss how we can benefit from such a connection. For efficiency, we discuss how our construction gives better efficiency than the previous approaches like [Kim25b,BK25].

Exact Reduction. Instead of reducing lazily as in the previous section, we may exactly reduce as in [Kim25b], at a higher cost. At a high level, we exactly emulate the reduction algorithm in [Kim25b, Section 3.2], and extract the digits one by one from the lowest digit. The detailed algorithm is presented in Algorithm 4.

Algorithm 4: ExactReduce

Setting: Homomorphic masking Mask_j for each $0 \le j < k$ that extracts the degree j coefficients by multiplying an approriate $0,1$ vector in the CKKS side. Bootstrapping plus masking $\mathsf{MaskBoot}_j$ defined as

$$\mathsf{MaskBoot}_j = \mathsf{StC}_1 \circ \mathsf{Mask}_j \circ \mathsf{LUT}_{\mathsf{id}} \circ \mathsf{EvalExp} \circ \mathsf{CtS} \circ \mathsf{ModRaise} \circ [\cdot]_{q_0} \circ \mathsf{StC}_2$$

for each $0 \le j < k$. We assume that we have sufficient modulus $\gg b^t$.

Input : $\mathsf{ct} \in \mathcal{R}_Q^2$ a ciphertext encrypting a polynomial vector $\mathbf{p} = (p_i(X))_{0 \le i < \frac{N}{2k}} \in (\mathbb{Z}[X]/(X^{2k}+1))^{\frac{N}{2k}}$. In other words,

$$\left\lfloor \frac{1}{\Delta} \cdot \mathsf{Dcd}_{2k} \circ \mathsf{Dec}_s(\mathsf{ct}) \right\rceil = \mathbf{p}.$$

In addition, all the coefficients of $\mathbf{p}$ belongs to $[0, b^t - b^{t-1})$.

Output: $\mathsf{ct}_{\mathsf{out}}$ encrypting the same message $\boldsymbol{z} \in \mathbb{Z}_{b^k}^{\frac{N}{2k}}$ but with a plaintext polynomial vector representing the correct digit decomposition.

$\mathsf{ct}' \leftarrow \mathsf{ct}$;
$\mathsf{ct}_{\mathsf{out}} \leftarrow 0$;
for $j \leftarrow 0$ **to** $k-1$ **do**
  $\mathsf{ct}_j \leftarrow \mathsf{MaskBoot}_j(\mathsf{ct}')$;
  $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{ct}_{\mathsf{out}} + \mathsf{ct}_j$;
  if $j \neq k-1$ **then**
    $\mathsf{ct}'' \leftarrow (\mathsf{ct}' - \mathsf{ct}_j)/b$;
    $\mathsf{sum} \leftarrow 0$;
    for $u \leftarrow 1$ **to** $t-1$ **do**
      $\mathsf{tmp} \leftarrow \mathsf{MaskBoot}_j(\mathsf{ct}'')$;
      $\mathsf{ct}'' \leftarrow (\mathsf{ct}'' - \mathsf{tmp})/b$;
      $\mathsf{sum} = \mathsf{sum} + b^{u-1} \cdot \mathsf{tmp}$;
    end for
    $\mathsf{ct}' \leftarrow \mathsf{ct}' + (X)_{0 \le i < \frac{N}{2k}} \cdot \mathsf{sum}$;
  end if
end for
return $\mathsf{ExactReduce}(\mathsf{ct}) := \mathsf{ct}_{\mathsf{out}}$.

Theorem 5. *The exact reduction in Algorithm 4 is correct and uses $tk - t + 1$ many discrete bootstrappings.*

Proof. For the number of discrete bootstrappings, we check that $\mathsf{MaskBoot}_j$ contains one discrete bootstrapping, and all j's except $j = k - 1$ use t bootstrappings. The case $j = k - 1$ uses exactly one bootstrapping, resulting in $t(k-1) + 1 = tk - t + 1$.

Next, we check the correctness of the operation. At the start of each for loop, we first store the j-th bit to ct_j. If $j \neq k - 1$, we need to move the carry to the higher bit, so we use a CinS encoding variant of the iterative MSB bootstrapping in [KN25]. Since all the coefficients of $\mathbf{p}$ belong to $[0, b^t - b^{t-1})$, $t-1$ unit bootstrappings are sufficient. The rest of the correctness is checked by the correctness of individual homomorphic operations (e.g. addition, multiplication, discrete bootstrapping). □

Exact reduction can be regarded as an exact digit decomposition, as it converts a modulo b^k integer into its correct digit decomposition form. In addition, exact reduction allows us to connect our encoding to the arithmetic system in [Kim25b]. That is, using the exact reduction as a subroutine, we can redefine all the (exact) homomorphic operations described in [Kim25b] such as addition, subtraction, comparison[5], bit shift, and even arbitrary function evaluation in [AKP25, Section 5.3].

Another remark is that one may combine both Reduce and $\mathsf{ExactReduce}$ to make exact reduction more efficient. That is, the approximate reduction algorithm Reduce first reduces the coefficients to a logarithmic scale: coefficients that belongs to $[0, b^t)$ reduces to $[0, tb)$. One can apply the exact reduction $\mathsf{ExactReduce}$ afterwards and get the exact digit decomposition. Such a combination is significantly more efficient than directly applying $\mathsf{ExactReduce}$. That is, if each coefficient (roughly) belongs to $[0, b^t)$, the direct application of $\mathsf{ExactReduce}$ needs t iterations and $tk-t+1$ bootstrappings, but the combination only requires $t + (\log_b(t) + 1) \cdot (k - 1) + 1$ bootstrappings.

Asymptotic Efficiency. We mainly compare the computational complexity of our scheme and [Kim25b] in terms of latency and throughput. We fix the discrete bootstrapping modulus b and check how many bootstrappings (i.e. modular reduction by b) we need to instantiate repeatable multiplication for the plaintext modulus b^k (so that k varies). In [Kim25b], the convolution term

$$\sum_{i=0}^{j} a_i \cdot b_{j-i}$$

[5] Comparison is a little bit tricky because our message space is $\mathbb{Z}_{b^k}^{\frac{N}{2k}}$ and extended subtraction as in [Kim25b] is not supported directly. We observe that the exact reduction in Algorithm 4 naturally extends to at most degree $2k - 1$ polynomials, allowing for extracting the k-th bit. Note that we count from 0 so that the k-bit is one bit higher than the MSB.

for $0 \le a_i, b_i < b$ can grow up to $(b-1)^2 \cdot k \approx b^2 k$. By [Kim25b, Theorem 4], this leads to

$$(2 + \log_b k) \cdot (k-1) + 1 = O(k \log k)$$

bootstrappings in total. On the other hand, repeatable multiplication in our case can occur when the equalities hold in Theorem 3: $\ell = tb$ and $k\ell^2 = b^t$. We need t bootstrappings in total where t satisfies

$$k \cdot (tb)^2 = b^t \quad \Longleftrightarrow \quad \log_b k + 2 \log_b t = t - 2.$$

In other words, $t = O(\log k)$. Usually, the parameter b is bounded (due to the efficiency of discrete bootstrapping) and can be regarded as a constant. In this regard, our construction reduces the asymptotic latency from $O(k \log k)$ to $O(\log k)$, which is a significant improvement. When it comes to throughput, the number of slots for [Kim25b] does not depend on k, whereas ours is $O(1/k)$. Therefore, both methods have asymptotic complexity $O(\log k)$. In conclusion, our construction asymptotically keeps the throughput while greatly reducing the latency.

Another important asymptotic is the number of multiplications, whose effect is usually negligible for moderately sized k but can be significant for large k. In [Kim25b], to construct all the convolution terms $\sum_i a_i \cdot b_{j-i}$, they need $O(k^2)$ homomorphic multiplications. Since it can be relinearized lazily, it has $O(k)$ many relinearizations and $O(k^2)$ ring multiplications over $\mathcal{R}_Q$. In practice, the ring multiplication cost is negligible compared to the bootstrapping cost (e.g. $1000\times$), and they ignored it in the analysis. However, when we consider extremely large k (e.g. $k = 10000$), the quadratic growth can be problematic. On the other hand, we rely on the CinS encoding, which natively supports the convolutions in one ring multiplication.

Next, we compare with [BK25], which can support power-of-radix moduli efficiently by simulating the modulo arithmetic inside a CRT system. If we assume that there are infinitely many primes with a given size (e.g. b bit) and use a single-layer version of their scheme, the number of bootstrappings for IMod_r in [BK25, Algorithm 3] is 2, which is a constant. However, they have $O(k)$ many homomorphic rotations, which can exceed the bootstrapping cost for large k.[6] To reduce this, one may use a double-layer version of their scheme. The number of bootstrappings increases to 10 and the asymptotic complexity for rotations becomes $O(\sqrt{k})$. Since their scheme naturally generalizes to an arbitrary number of iterations, one could consider using three or more CRT layers. In general, there is a trade-off between the number of bootstrappings per modular reduction and the asymptotic complexity of the number of rotations. The asymptotic comparison does not seem to be well defined, and we rather compare the concrete performance in Section 4.

[6] Here k can be either the number of moduli in the CRT system or the plaintext modulus b^k.

4 Experiments

We provide proof-of-concept implementation results for our scheme. Our implementation is built upon Lattigo [lat24] and all our experiments are run single-threaded on Apple M4 Max with 128GB of RAM running Sequoia 15.6. All of our parameters satisfy 128-bit security according to [BTPH22]. We repeated each experiment at least 20 times to measure execution time and noise size.

4.1 Our implementation

We used a fixed FHE parameter for all the experiments, which is described in Table 2. We built a generic (unoptimized) parameter that can afford all the plaintext precision we experimented with. A dedicated construction for each plaintext precision would give much better performance, especially in terms of latency.

$\log N$	$\log QP$	$(h, \tilde{h})$	$dnum$
16	1537	$(192, 32)$	4

$\log q_i$								$\log p_j$
Base	StC_2	PolyMult	StC_1	CKKSMult	LUT	EvalExp	CtS	
53	53×2	53×2	53×1	53×2	53×4	53×8	53×3	53×6

Table 2. Parameter set for the experiments. Here N denotes the RLWE dimension, $\log QP$ denotes the bit size of the maximum switching key modulus, $(h, \tilde{h})$ denote Hamming weight of the dense and sparse secret keys [BTPH22], respectively, $dnum$ denotes the gadget rank of the switching keys, $\log q_i$ denotes the moduli size of the ciphertext moduli chain, and $\log p_j$ denotes the moduli size of the auxiliary moduli chain. The names under $\log q_i$ denote how many moduli there are for each operation, and $X \times Y$ means that there are Y many X bit primes.

For different plaintext precisions (from 16 bit to 2048 bit), we experimented with our homomorphic multiplication, as detailed in Table 3. As discussed in the earlier section, the number of bootstrappings is proportional to the logarithm of the plaintext precision $\log b^k$ (i.e. $\propto \log \log b^k$). As a result, the latency grows very slowly, and the throughput is almost proportional to the plaintext precision. Since higher plaintext precision involves larger integers during discrete CKKS computations, the noise size grows as we increase plaintext precision.

4.2 Comparison with Prior Works

We first compare our result with [Kim25b], as our method is a direct improvement over their method. As can be observed in Table 4, our method is significantly faster than [Kim25b] while maintaining similar throughput. In particular, our latency for 64 bit multiplication is 5.82 times faster than [Kim25b], and our throughput is about 1.38 times worse. Importantly, the number of bootstrapping

$\log b^k$	# slots	b	k	# BTS	latency	throughput	noise size
16	8192	2^4	4	4	25.3 sec	3.09 ms	$(28.3, 26.0)$
32	4096	2^4	8	4	24.9 sec	6.08 ms	$(28.1, 24.6)$
64	2048	2^4	16	4	24.9 sec	12.2 ms	$(27.2, 23.0)$
128	1024	2^4	32	5	30.9 sec	30.2 ms	$(23.7, 19.3)$
256	512	2^4	64	5	31.9 sec	62.3 ms	$(21.1, 16.5)$
512	256	2^4	128	5	32.4 sec	127 ms	$(18.1, 13.7)$
1024	128	2^4	256	6	38.0 sec	297 ms	$(14.2, 10.0)$
2048	64	2^4	512	6	37.7 sec	589 ms	$(11.0, 6.96)$

Table 3. Homomorphic multiplication for different plaintext precision. Here, BTS means discrete bootstrapping, throughput is latency divided by the number of slots, and the noise size is (mean, worst) error precision, respectively, after all the iterations. Mean noise size is measured as $-\log$ of the average error across all iterations and slots, and worst noise size is measured as $-\log$ of the largest error across all iterations and slots.

of [Kim25b] grows linearly in the plaintext precision, while ours grows logarithmically. Since they did not provide figures for precision greater than 64 bit so we could not directly compare, but we can estimate the difference for larger precision. For larger precision, the latency gap will significantly increase, due to the different number of bootstrapping. For instance, for 256 bit multiplication, they need at least $3 \times 256 - 2 = 766$ multiplications, and the expected latency is at least $145 \text{ sec} \times 766/46 \approx 2410$ sec. It is almost a hundred times slower than our result in Table 3.

	prec	# BTS	latency	throughput
[Kim25b]	16	10	31.2 sec	1.90 ms
	32	22	69.6 sec	4.25 ms
	64	46	145 sec	8.85 ms
Ours	16	4	25.3 sec	3.09 ms
	32	4	24.9 sec	6.08 ms
	64	4	24.9 sec	12.2 ms

Table 4. Comparison with [Kim25b] on homomorphic multiplication. The figures for [Kim25b] are borrowed from [Kim25b, Table 7], and it was experimented in almost the same environment.

Next, we compare our result with [Zam22], as they are the state-of-the-art for *fast* homomorphic multiplication for power-of-two modulus arithmetic. The comparison is detailed in Table 5. Although we greatly improved the latency, our result is slower than [Zam22] for small precision (16, 32 bits). As soon as we reach 64 bit, however, our latency becomes better than [Zam22]. The gap increases to more than an order of magnitude or more when it comes to higher precision like 256 or 512 bit. As we keep good throughput as [Kim25b], the throughput gap

between ours and [Zam22] is still roughly three to four orders of magnitude. In this regard, we mark the crossing point:

> *The RLWE-based scheme performs better than LWE-based scheme(s) for standard (64 bit) integer arithmetic, even in terms of latency.*

	prec	latency	throughput
[Zam22]	16	1.61 sec	1.61 sec
	32	6.44 sec	6.44 sec
	64	25.4 sec	25.4 sec
	128	101 sec	101 sec
	256	403 sec	403 sec
	512	1600 sec	1600 sec
Ours	16	25.3 sec	3.09 ms
	32	24.9 sec	6.08 ms
	64	24.9 sec	12.2 ms
	128	30.9 sec	30.2 ms
	256	31.9 sec	62.3 ms
	512	32.4 sec	127 ms

Table 5. Comparison with [Zam22] on homomorphic multiplication. The timings for [Zam22] are measured in the same machine as our experimental results, running their integer benchmark with commit hash 312952007f044bfc97e557b6369440252fd5acd6.

The recent work [BK25] also achieves good latency by using a nested CRT system. Although their method is good for arbitrary modulus, our approach outperforms them for power-of-two modulus. For example, we have 4.70 times better latency and 75.3 times better throughput for 256 bit computations. We detail the comparison in Table 6.

	prec	# slots	latency	throughput
[BK25]	256	32	150 sec	4.69 sec
	2048	4	190 sec	47.5 sec
Ours	256	512	31.9 sec	62.3 ms
	2048	64	37.7 sec	589 ms

Table 6. Comparison with [BK25] on homomorphic multiplication. The figures for [BK25] are borrowed from [BK25, Table 7], and it was experimented in almost the same environment. Although they experimented with prime modulus, their approach has the same performance for a similar-sized power-of-two modulus.

Alternatively, one may consider comparing with the BGV/BFV family. When directly using power-of-two as the plaintext modulus, the state-of-the-art bootstrapping methods either go through the standard digit extraction [GIKV23,MHWW24]

or the CKKS bootstrapping [KSS24]. The standard approach needs to use a high-precision digit extraction, which is costly due to the large modulus consumption. For instance, the experimental result in [GIKV23, Table 7] reports the modulo 2^{51} bootstrapping with ≈ 1000 bits of modulus consumption and execution time of 64 seconds, having 2016 slots and 80 bits of security. Recall that our bootstrapping bootstraps a modulo 2^{64} integer in 25.3 seconds, having 2048 slots and 128 bits of security. Although [MHWW24] may give better performance on modulus 2^{64}, they did not report such a large modulus, and its performance is expected to be not so much better than [GIKV23]. For [KSS24], to support at least one multiplication, the CKKS bootstrapping precision needs to be at least as large as the plaintext precision. For 64 bit precision, this leads to ≈ 80 bit of precision. Assuming that we use four iterations of META-BTS [BCC$^+$22] on standard 20 bit precision CKKS bootstrapping, its performance is roughly similar or slightly worse than ours, which uses four discrete bootstrappings of 4 bit precision. For higher precision, the latency gap increases as our latency grows much more slowly than their linear growth. In terms of throughput, however, their approach may be preferable as they can use the entire plaintext space with N slots. Note that these approaches are not directly compatible with [Kim25b] and struggle to support a broader set of operations such as comparison and bit shifts.

Lastly, we compare with the GBFV homomorphic modular reduction in [Kim25a]. In [Kim25a, Section 4.2], they discuss converting a GBFV ciphertext into a CKKS ciphertext and computing modular reduction. Since they do not provide implementation, we cannot directly compare our results with theirs. To estimate the cost, suppose that we want to achieve modulo 2^{64} multiplication by using modulo $p = 2^{256} + 1$ arithmetic with $b = 8$ and $N/k = 32$ so that the plaintext space is $\mathbb{Z}_{b^{N/k}+1}[X]/(X^k - b)$. If we choose $N = 2^{16}$, they have $k = 2^{16}/32 = 2048$ slots. They need to use the MSB bootstrapping from [KN25] and achieve bootstrapping precision of ≥ 14 bits[7], requiring at least four iterations of 4 bit discrete bootstrapping. Such a rough estimation shows that our approach is at least as good as [Kim25a].

5 Conclusion

Discrete CKKS is emerging as a promising primitive for general encrypted computations, significantly outperforming CGGI/DM in terms of throughput. In this work, we showed that discrete CKKS can be better at integer arithmetic even in terms of latency, as long as the target precision is sufficiently large. An open question is to extend this result to other circuits, and ultimately to arbitrary complex computations in general.

[7] According to [GV25, Table 1].

Acknowledgement

This work was supported by Stanford Graduate Fellowship, NSF, DARPA, and the Simons Foundation. Opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

AKP25. A. Alexandru, A. Kim, and Y. Polyakov. General functional bootstrapping using CKKS. In *CRYPTO*, 2025.

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.

BFG+25. Z. Brakerski, O. Friedman, D. Golan, A. Gurny, D. Mutzari, and O. Sheinfeld. REFHE: Fully homomorphic ALU. Cryptology ePrint Archive, Paper 2025/1449, 2025.

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

BK25. D. Boneh and J. Kim. Homomorphic encryption for large integers from nested residue number systems. In *CRYPTO*, 2025.

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.

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.

CHM+25. H. Cha, I. Hwang, S. Min, J. Seo, and Y. Song. MatriGear: Accelerating Authenticated Matrix Triple Generation with Scalable Prime Fields via Optimized HE Packing . In *IEEE S&P*, 2025.

CJP21. I. Chillotti, M. Joye, and P. Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In *CSCML*, 2021.

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.

CPL25. G. Cha, D. Park, and J.-W. Lee. Improved radix-based approximate homomorphic encryption for large integers via lightweight bootstrapped digit carry. Cryptology ePrint Archive, Paper 2025/1740, 2025.

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.

Gen09. C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.

GIKV23. R. Geelen, I. Iliashenko, J. Kang, and F. Vercauteren. On polynomial functions modulo p^e and faster bootstrapping for homomorphic encryption. In *EUROCRYPT*, 2023.

GV23. R Geelen and F Vercauteren. Bootstrapping for BGV and BFV revisited. *J. Cryptol.*, 2023.

GV25. R. Geelen and F. Vercauteren. Fully homomorphic encryption for cyclotomic prime moduli. In *EUROCRYPT*, 2025.

JPK+24. J. H. Ju, J. Park, J. Kim, M. Kang, D. Kim, J. H. Cheon, and J. H. Ahn. Neujeans: Private neural network inference with joint optimization of convolution and fhe bootstrapping. In *CCS*, 2024.

Kim25a. J. Kim. Bootstrapping GBFV with CKKS. Cryptology ePrint Archive, Paper 2025/888, 2025.

Kim25b. J. Kim. Efficient homomorphic integer computer from CKKS. *TCHES*, 2025.

KN25. J. Kim and T. Noh. Modular reduction in CKKS. *Comm. in Cryptol.*, 2025.

KS22. K. Kluczniak and L. Schild. FDFB: Full domain functional bootstrapping towards practical fully homomorphic encryption. *TCHES*, 2022.

KSS24. J. Kim, J. Seo, and Y. Song. Simpler and faster BFV bootstrapping for arbitrary plaintext modulus from CKKS. In *CCS*, 2024.

lat24. Lattigo v6. Online: `https://github.com/tuneinsight/lattigo`, August 2024. EPFL-LDS, Tune Insight SA.

MHWW24. S. Ma, T. Huang, A. Wang, and X. Wang. Accelerating BGV bootstrapping for large p using null polynomials over $\mathbb{Z}_{p^e}$. In *EUROCRYPT*, 2024.

Zam22. Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. `https://github.com/zama-ai/tfhe-rs`.

ZYZ+24. F. Zhang, C. Yang, R. Zong, X. Zheng, J. Wang, and Y. Meng. An efficient and scalable FHE-based PDQ scheme: Utilizing FFT to design a low multiplication depth large-integer comparison algorithm. *TIFS*, 2024.