# Efficient Homomorphic Integer Computer from CKKS

Jaehyung Kim
Stanford University
Stanford, United States
jaehk@stanford.edu

## Abstract

As Fully Homomorphic Encryption (FHE) enables computation over encrypted data, it is a natural question of how efficiently it handles standard integer computations like 64-bit arithmetic. It has long been believed that the CGGI/DM family or the BGV/BFV family are the best options, depending on the size of the parallelism. The Cheon–Kim–Kim–Song (CKKS) scheme, although being widely used in many applications like machine learning, was not considered a good option as it is more focused on computing real numbers rather than integers.

Recently, Drucker *et al.* [J. Cryptol.] suggested to use CKKS for discrete computations, by separating the error/noise from the discrete message. Since then, there have been several breakthroughs in the discrete variant of CKKS, including the CKKS-style functional bootstrapping by Bae *et al.* [Asiacrypt'24]. Notably, the CKKS-style functional bootstrapping can be regarded as a parallelization of CGGI/DM functional bootstrapping, and it is several orders of magnitude faster in terms of throughput. Based on the CKKS-style functional bootstrapping, Kim and Noh [ePrint, 2024/1638] designed an efficient homomorphic modular reduction for CKKS, leading to modulo small integer arithmetic.

Although it is known that CKKS is efficient for handling small integers like 4 or 8 bits, it is still unclear whether its efficiency extends to larger integers like 32 or 64 bits. In this paper, we propose a novel method for homomorphic unsigned integer computations. We represent a large integer (e.g. 64-bit) as a vector of smaller chunks (e.g. 4-bit) and construct arithmetic operations relying on the CKKS-style functional bootstrapping. The proposed scheme supports many of the operations supported in TFHE-rs while outperforming it in terms of amortized running time. Notably, our homomorphic 64-bit multiplication takes 17.9ms per slot, which is more than three orders of magnitude faster than TFHE-rs.

## CCS Concepts

• **Security and privacy → Cryptography**.

## Keywords

Fully Homomorphic Encryption, Integer Arithmetic, Discrete CKKS

## 1 Introduction

Fully Homomorphic Encryption (FHE) is a branch of cryptography that allows computation in an encrypted state. Since Gentry's first instantiation [Gen09], the efficiency of FHE has been improved dramatically. The major FHE schemes can be categorized into LWE-based or RLWE-based depending on whether the default ciphertext format is LWE or RLWE, respectively. LWE-based schemes such as the CGGI/DM family [CGGI16, DM15] are known to be fast (in terms of latency) and flexible, while RLWE-based schemes such as the BGV/BFV family [Bra12, FV12, BGV12] and CKKS [CKKS17] have better throughput. One of the key technical differences is that computations on LWE-based schemes mostly rely on programmable/functional bootstrapping [CJP21, KS22] whereas computations on RLWE-based schemes mainly use homomorphic polynomial evaluations (i.e. addition and multiplication).

It had long been believed that different families are good at different functionalities. For instance, the CGGI/DM family was used to handle small or non-parallelizable computations, the BGV/BFV family was used to handle parallelizable large integer computations, and CKKS was used to handle real number arithmetic. However, recent improvements in the discrete variant of CKKS [DMPS24, CKKL24, BCKS24, BKSS24, AKP24, KN24] suggest that this may not be true. In particular, recent works [BCKS24, BKSS24, AKP24] show that CKKS handles functional/programmable bootstrapping faster than CGGI/DM by several orders of magnitude. As functional/programmable bootstrapping is a core component of CGGI/DM, this means that it can be preferable to use CKKS if we have more than hundreds of parallelism.

Since we know that CKKS handles homomorphic look-up tables efficiently, a natural question is whether CKKS is good at integer computations in general. For computations modulo NTT-friendly primes, it seems that the BGV/BFV family is the right choice, as they provide a fast and efficient solution for both latency and throughput, in the light of recent improvements in BGV/BFV bootstrapping [MHWW24, KSS24]. However, when it comes to standard modulo power-of-two computations like unsigned 64-bit arithmetic, the landscape is relatively unclear. The most straightforward option, to use the plaintext modulus to be a power-of-two has several problems. One problem is that the plaintext modulus is too large to be efficiently supported. For instance, 64-bit multiplications in BGV/BFV enlarge the noise by $> 64$ bits, and choosing large parameters like large ring degree (e.g. $\log(N) \geq 17$) is unavoidable. Recall that large parameters lead to inefficiency in terms of latency and memory footprint, which is not desirable. Another problem is that the use of power-of-two modulus prevents one from using large parallelism. In BGV/BFV, the number of slots is determined by how the cyclotomic polynomial $\Phi_M(X)$ splits in the plaintext space $\mathbb{Z}_t$. As $t$ is a power-of-two, we do not have as many slots as in the case of NTT-friendly primes, leading to lower throughput for handling modulo $t$ computations in parallel. In addition, directly using computations over $\mathbb{Z}_t$ makes it difficult to handle important components of integer arithmetic such as bit shift and comparison.

Instead of directly supporting $\mathbb{Z}_t$ arithmetic for large $t$, the existing approaches often decompose $t$ into several chunks to improve efficiency. For instance, several works [CKK16, XCWF16, QZL+19, ZQH+21, HZY+22] considered radix-2 arithmetic operations for handling integer computations. Similarly, [TLW+21, IZ21] decompose large modulus into smaller finite field elements, leading to

efficient homomorphic comparison. By decomposing the desired plaintext modulus $t$, one can keep the noise growth small while supporting efficient homomorphic operations like comparison and bit shifts. However, as modulo 2 (or $2^\ell$ for small $\ell$) plaintext space still has a limited number of slots, the SIMD capability or throughput is not as great as the case of NTT-friendly plaintext modulus.

An alternative option is to use the other families such as CGGI/DM or CKKS. Using CGGI/DM to handle integer computations is relatively well explored, and the major libraries like TFHE-rs [Zam22] allow standard integer computations of up to 256 bits by using lower precision arithmetic as building blocks. On the other hand, using CKKS to handle integer computations is new (motivated in [DMPS24] and related to [BCKS24, KSS24, AKP24, KN24]) and has not yet been fully explored. For small precision like 4 or 8 bits, one can regard integers as real numbers and rely on CKKS operations, look-up tables [CKKL24], and modular reduction [KN24]. For large precision, a straightforward approach is to use high-precision CKKS and directly supporting $\mathbb{Z}_t$ for large power-of-two integer $t$, but this has a similar problem as in BGV/BFV such as large parameters and lack of efficient comparison/bit shift.

In this regard, it can be tempting to use a decomposition-based approach as in BGV/BFV. Indeed, [ZYZ+24] suggests to use decomposition to efficiently handle both homomorphic multiplication and comparison. However, their arithmetic operations do not modular reduce, and individual digits exceed the base and continuously grow. As a result, their approach cannot support bootstrapping and can evaluate only circuits of predetermined size. To support bootstrapping, we need to take care of the carries every time we perform homomorphic arithmetic operations. Since CKKS does not naturally support modular reduction, we need a different strategy than the ones in the BGV/BFV family. In this paper, we suggest using the modular reduction in [KN24] in a clever way to efficiently instantiate fully homomorphic encryption over large unsigned integers (i.e. arithmetic over $\mathbb{Z}_{2^k}$ for large $k$).

## 1.1 Technical Overview

We provide a simplified overview of our method, focusing on how the underlying message behaves through homomorphic operations. At a high level, we decompose a large integer into digits and rely on the grade-school addition/multiplication that computes iteratively from the lowest digit. Let $d$ be the base for our digit decomposition, and let $t$ be the target modulus. We aim to enable $\mathbb{Z}_t$ arithmetic using digit decomposition of base $d$.

*Ingredients.* Recall that discrete CKKS allows us to encrypt integers rather than real numbers. If we rely on the inclusion $\mathbb{Z} \hookrightarrow \mathbb{C}$ as in [DMPS24], we have addition and multiplication over integers directly inherited from the corresponding CKKS operations. Although it can afford arbitrary precision in theory, we assume that it only supports bounded precision arithmetic (e.g. at most $d^2$) for efficiency. In other words, we can add and multiply small integers.

The next ingredient is the modular reduction from [KN24]. The input is a discrete CKKS ciphertext encrypting integers, and it takes modulo $d$ via the discrete bootstrapping from [BKSS24]. Taking into account that bootstrapping is roughly as costly as hundreds of homomorphic multiplications, we can assume that we have an expensive homomorphic modular reduction.

To summarize, we have addition and multiplication over $\mathbb{Z}$, and a modular reduction $[\cdot]_d$ which is very costly.

*Our Goal.* We construct a homomorphic computer that computes unsigned modulo $t$ arithmetic based on addition, multiplication, and modulo $d$. We aim to support common integer operations such as arithmetic, comparison, and shift operations.

The straightforward approach is to imitate what modern computers do. However, it turns out that it is not a good idea because our homomorphic computer has a different characteristic than the usual computer. That is, modular reduction is way more costly than addition and multiplication, which means that the computational complexity is determined by the number of modular reductions,[1] not the number of multiplications. As a result, we need to build algorithms that minimize the number of modular reductions.

*Homomorphic Multiplication.* We describe our multiplication algorithm that relies on grade-school multiplication. Let

$$a = \sum_{i=0}^{u-1} a_i \cdot d^i \text{ and } b = \sum_{i=0}^{u-1} b_i \cdot d^i$$

be digit decompositions of two integers $a, b \in \mathbb{Z}_t$, where $u = \log_d(t)$. As illustrated in Figure 1, we iteratively compute the lowest digit as follows.

- Step 0. We compute $a_0 b_0$, then extract the low part $c_0$ and the high part $c_0$'.
- Step $i$. We compute $c'_{i-1} + \sum_{j=0}^{i} a_j b_{i-j}$, then extract the low part $c_i$ and the high part $c'_i$.
- We iterate until $i = u - 1$, and output $(c_0, c_1, \ldots, c_{u-1})$.

Here we observe that one homomorphic modular reduction per step is sufficient to evaluate both low and high parts: the high part can be computed by subtracting the low part from the original one and dividing by $d$.[2] Therefore, it leads to $u$ modular reductions in total. Although there are $O(u^2)$ homomorphic multiplications in this algorithm, its cost is negligible compared to the cost of modular reduction.

*Other Operations.* Addition and subtraction work similarly to multiplication. Homomorphic comparisons such as $a \geq b$ can be computed as regarding $a - b$ as a $u + 1$ digit integer and extracting the highest digit. Since we already have a reduced form after subtraction, comparison is just as costly as subtraction. To be specific, we may follow (the analogue of) the iterative algorithm for subtraction until the last step, and output $c'_{u-1} + 1$ instead of $(c_0, c_1, \ldots, c_{u-1})$, as it corresponds to the $u$-th digit. For homomorphic shift operation by a scalar, we may appropriately shift the digits and apply modular reductions. As the simplest case, if the right shift amount $y$ is less than $\log_2(d)$, then the $i$th digit can be written as

$$lo(a_i << y) + hi(a_{i-1} << y)$$

where $(a_0, a_1, \ldots, a_{u-1})$ is the input.

[1] This explains why we rely on the naive grade-school addition/multiplication instead of more sophisticated algorithms like Karatsuba or FFT.

[2] Note that dividing by $d$ consumes modulus, leading to continuous decrease in ciphertext modulus. We illustrate how we overcome this problem in Section 3.2.

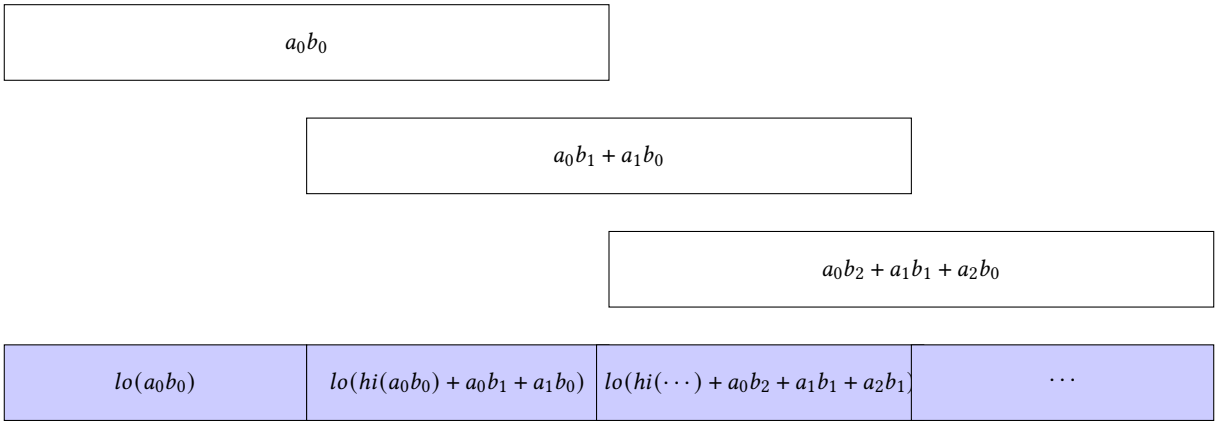


**Figure 1: Grade-school multiplication between $\sum_i a_i \cdot d^i$ and $\sum_i b_i \cdot d^i$.**

## 1.2 Contribution

We propose an efficient SIMD[3] FHE over integers that supports a wide range of operations. In particular, our scheme provides three different types of operations, namely arithmetic operations, comparison, and bootstrapping. Note that these operations are the most important primitives of FHE and are necessary for most applications.

*Concrete Efficiency.* Among the schemes that support diverse operations, our scheme provides the most efficient performance in terms of throughput. As we rely on CKKS, we can enjoy almost the maximum parallelism possible (i.e. $N/2$ slots), leading to outperforming the approaches based on other FHE schemes. We compare the concrete performance on homomorphic multiplication and comparison in Table 1 and 2. Compared to the widely used TFHE-rs [Zam22], our multiplication and comparison are $\approx$ three and two orders of magnitude faster, respectively. For BGV/BFV, we compare with the state-of-the-art decomposition-based methods, achieving $\approx 2$ orders of magnitude acceleration in multiplication and $\approx 6$ times faster in comparison.

| | $k$ | $\lambda$ | amortized time (ms) |
|---|---|---|---|
| [HZY+22] | 32 | 80 | 1020 |
| [Zam22] | 32 | 128 | 7830 |
| | 64 | | 30900 |
| Ours | 32 | $\approx 128$ | 8.79 |
| | 64 | | 17.9 |

**Table 1: Comparison on multiplication.**

Compared to the possible approaches that directly encode large integers without decomposition, our method achieves much smaller FHE parameters. As arithmetic over $\mathbb{Z}_t$ for large $t$ like 64-bit consumes a huge amount of modulus per multiplication, it is unavoidable for the direct encoding approaches to use large ring dimension such as $\log N = 17, 18$. On the other hand, we use the FHE parameter for $\mathbb{Z}_d$ to instantiate modulo at most $d^d$ arithmetic, having significantly smaller parameters to achieve the same precision. Indeed, we were able to instantiate 64-bit precision arithmetic using $\log N = 15$ parameters with only $\approx 800$ bits of modulus budget.

[3]Refers to Single Instruction Multiple Data.

| | $\lambda$ | # slots | latency (sec) | amortized time (ms) |
|---|---|---|---|---|
| [TLW+21] | $> 80$ | 256 | 20.2 | 78.8 |
| | | 128 | 20.5 | 160 |
| | | 16 | 4.75 | 297 |
| [Zam22] | 128 | 1 | 0.852 | 852 |
| Ours | $\approx 128$ | 16384 | 197 | 12.0 |

**Table 2: Homomorphic Comparison over $\mathbb{Z}_{2^{64}}$.**

*Versatility.* Our scheme not only provides efficient operations but also offers a variety of new operations. Traditionally, SIMD schemes were only capable of computing addition and multiplication, leading to inefficiency in computing other types of functions. On the contrary, our scheme supports arithmetic, comparison, and shift operation, leading to a more complete set of instructions to emulate the (unencrypted) computer. Furthermore, our scheme is compatible with the multi-precision arbitrary function evaluation in [AKP24], supporting any function in a reasonable time complexity.

In addition, our method gives CKKS the ability to efficiently compute over large integers, which means that CKKS is now capable of computing both real numbers and integers. As our encoding is directly compatible with the CKKS encoding, one can efficiently convert one format to the other to enjoy maximal efficiency. This extends the quantization framework in [KN24], allowing both accurate and approximate computations.

*Enhanced Security.* As the original CKKS cannot distinguish the error from the message, it was difficult for CKKS to achieve advanced security notions such as IND-CPA$^D$ security [LM21] or Threshold FHE security [AJLA+12, BGG+18]. Since the discrete CKKS framework allows us to distinguish the error from the ciphertext by discretizing the message space, it can enjoy the advantages of the exact schemes in terms of security. As we provide an efficient integer computer, discrete CKKS can now handle any operation without going through the original CKKS, leading to better security.

For instance, one may use our homomorphic fixed point arithmetic instead of the usual CKKS operations to avoid large noise flooding needed to achieve advanced security.

## 2 Preliminaries

Let $N > 1$ be a power of two integers and $Q > 1$ be an integer. Let $\mathcal{R} = \mathbb{Z}[X]/(X^N+1)$ and $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R}$. Let $\mathsf{DFT} : \mathbb{R}[X]/(X^N+1) \to \mathbb{C}^{N/2}$ be a discrete Fourier transform (DFT) defined as

$$m(X) \quad \mapsto \quad \left(m(\zeta^{5^i})\right)_{0 \le i < N/2}$$

where $\zeta$ is a complex primitive $2N$th root of unity. Let $\mathsf{iDFT} : \mathbb{C}^{N/2} \to \mathbb{R}[X]/(X^N+1)$ be its inverse.

### 2.1 CKKS Basics

In CKKS, there are two kinds of encoding called slots-encoding and coeffs-encoding. The **slots-encoding** $\mathsf{Ecd} : \mathbb{C}^{N/2} \to \mathcal{R}$ is defined as

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

where $\Delta > 0$ is a scaling factor. The slots-decoding $\mathsf{Dcd} : \mathcal{R} \to \mathbb{C}^{N/2}$ is defined as

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

The slots-encoding supports single instruction multiple data (SIMD) computations and used as a default encoding for CKKS. The **coeffs-encoding** $\mathsf{CoeffEcd} : \mathbb{R}^N \to \mathcal{R}$ is defined as

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

where $\vec{z} = (z_0, z_1, \ldots, z_{N-1})$, simply scaling up the vector and round. The coeffs-decoding $\mathsf{CoeffEcd} : \mathcal{R} \to \mathbb{R}^N$ is its approximate inverse defined as

$$\mathsf{CoeffDcd}\left(\sum_{i=0}^{N-1} m_i X^i\right) = \frac{1}{\Delta}(m_0, m_1, \ldots, m_{N-1}).$$

As coeffs-encoding is compatible with coefficient-wise operations, we may use it for bootstrapping (to raise modulus), conversions to/from other schemes [BGGJ20], or modular reduction [KN24].

In CKKS, each homomorphic multiplication increases the scaling factor from $\Delta$ to $\Delta^2$, and we rescale the ciphertext to keep the scaling factor to be $\simeq \Delta$. As rescaling reduces the ciphertext modulus, the ciphertext modulus gradually decreases and cannot allow further multiplications at some point. The **CKKS bootstrapping** [CHK+18] increases the modulus, recovering the multiplicative capability.

Definition 1 (CKKS Bootstrapping). *Let $q, Q > 1$ be integers such that $Q > q$. Let $\mathsf{ct} \in \mathcal{R}_q^2$ be a CKKS ciphertext encrypting a vector $\vec{z} \in \mathbb{C}^{N/2}$ via the slots-encoding, where both real and imaginary parts of each entry are in $[-1, 1]$. The CKKS bootstrapping $\mathsf{BTS} : \mathcal{R}_q^2 \to \mathcal{R}_Q^2$ raises the modulus while approximately preserving the underlying message. That is,*

$$\mathsf{BTS}(\mathsf{ct}) = \mathsf{ct}' \in \mathcal{R}_Q^2$$

*where $\mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}) \simeq \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}')$. Here $\mathsf{Dec}$ denotes the CKKS decryption.*

The standard CKKS bootstrapping [CHK+18] can be simplified as a combination of modulus raising (denoted as ModRaise) and modular reduction (denoted as EvalMod). As the natural modulus raising adds a small error on the most significant bits ($m \mapsto m + q_0 \cdot I$), we need to remove this error by homomorphically evaluating a modular reduction ($m + q_0 \cdot I \to m$).

An additional issue is that ModRaise requires coeffs-encoding whereas EvalMod requires slots-encoding. Hence, we need conversions between them, which we can instantiate with homomorphic evaluation of DFT/iDFT. Conversions from slots-encoding to coeffs-encoding and vice versa are denoted as StC and CtS, respectively. As a result, the CKKS bootstrapping can be instantiated within the order of StC-ModRaise-CtS-EvalMod, as illustrated in Algorithm 1.

**Algorithm 1:** Slot Bootstrapping [BCC+22]

**Setting :** $\Delta \ll q_0$.
**Input :** $\mathsf{ct} = \mathsf{Enc} \circ \mathsf{Ecd}(\vec{z}) \in \mathcal{R}_q^2$ with $\vec{z} \in [-1, 1]^{N/2}$.
**Output :** $\mathsf{ct}_{\mathsf{out}} \in \mathcal{R}_Q^2 = \mathsf{Enc} \circ \mathsf{Ecd}(\vec{w})$, where $\vec{w} \simeq \vec{z}$.

$\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{EvalMod} \circ \mathsf{CtS} \circ \mathsf{ModRaise} \circ \mathsf{StC}(\mathsf{ct})$;
**return** $\mathsf{ct}_{\mathsf{out}}$.

### 2.2 Discrete CKKS

The recent improvements in the discrete variant of CKKS (first formalized in [DMPS24]) add additional encoding structure to CKKS for handling discrete data. Instead of using the whole continuous space such as $\mathbb{C}$ or $\mathbb{R}$, we focus on its discrete subset. For instance, one may use the inclusion $\mathbb{Z} \hookrightarrow \mathbb{C}$ to deal with integer computations using CKKS (as suggested in [DMPS24]). The observation is that this inclusion is a ring homomorphism and we can use CKKS operations to handle integer addition and multiplication. Importantly, this framework allows us to distinguish the underlying error from the ciphertext which was not possible in the original CKKS.

Definition 2 (Discrete CKKS Ciphertext). *Let $X \subseteq \mathbb{C}$ be a finite set. Let $X \hookrightarrow \mathbb{C}$ be an additional discrete encoding. A discrete CKKS ciphertext encrypting a vector $\vec{z} \in X^{N/2}$ is a CKKS ciphertext $\mathsf{ct}$ that encrypts a vector $\vec{z} + \vec{e} \in \mathbb{C}^{N/2}$ where $\vec{e}$ is small. In this case, we call $\vec{e}$ as the underlying error of the discrete CKKS ciphertext $\mathsf{ct}$.*

As the error is separated from the message, we may reduce the error by evaluating a cleaning polynomial. For instance, [DMPS24] describes a cleaning function $h_1(x) = 3x^2 - 2x^3$ for the encoding $\{0, 1\} \hookrightarrow \mathbb{C}$ that reduces the error. Note that the cleaning in the context of discrete CKKS is very similar to the notion of bootstrapping in the other schemes, and can be regarded as another type of bootstrapping.

Definition 3 (Cleaning). *Let $X \subseteq \mathbb{C}$ be a finite set. Let $X \hookrightarrow \mathbb{C}$ be an additional discrete encoding. Let $\mathsf{ct} \in \mathcal{R}_q^2$ be a discrete CKKS ciphertext encrypting a vector $\vec{z} \in X^{N/2}$ whose underlying error is $\vec{e} \in \mathbb{C}^{N/2}$. The cleaning function maps $\mathsf{ct}$ to $\mathsf{ct}'$, so that the underlying error of $\mathsf{ct}'$, denoted as $\vec{f}$ is much smaller than $\vec{e}$.*

Homomorphic operations on discrete CKKS ciphertexts can be classified into two types, i.e. arithmetic operation and interpolation. Arithmetic operation consists of addition and multiplication

inherited from the original CKKS whereas interpolation refers to polynomial interpolation over a finite number of points. For interpolation, [CKKL24] suggests to use complex roots-of-unity for polynomial interpolations, as it provides more numerically stable interpolation than the typical equispaced points on the real line (e.g. $\mathbb{Z} \subseteq \mathbb{C}$). The power of interpolation is that we can evaluate an arbitrary function. Note that the original CKKS needs to rely on polynomial approximations and thus could not compute discontinuous functions efficiently.

Definition 4 (Interpolation over roots-of-unity). *Let $X = \{1, \omega, \ldots, \omega^{t-1}\} \subseteq \mathbb{C}$ where $\omega$ is a primitive $t$-th root of unity. Let $f : \mathbb{Z}_t \to \mathbb{C}$ be an arbitrary function. The homomorphic look-up table $\mathsf{LUT}_f : \mathcal{R}_Q^2 \to \mathcal{R}_Q^2$ homomorphically evaluates an interpolation that corresponds to $f$. That is, $\omega^\alpha \in X$ is mapped to $f(\alpha)$ for each slot.*

Other lines of works [BCKS24, BKSS24, AKP24] build an analogue of functional/programmable bootstrapping in CGGI/DM [CJP21, KS22] for discrete CKKS. That is, they design new bootstrapping circuits dedicated to handling discrete data while evaluating an arbitrary function. The major observation is that one can use interpolation rather than approximation for evaluating homomorphic modular reduction, which means that we do not necessarily have a gap between the message and the base modulus. Notably, the CKKS-style functional bootstrapping (i.e. discrete bootstrapping) can be regarded as a parallelization of CGGI/DM bootstrapping which is several orders of magnitude faster than the state-of-the-art such as [Zam22]. We provide a (simplified) instantiation of discrete bootstrapping from [BKSS24] in Algorithm 2. Here EvalExp denotes a homomorphic evaluation of the complex exponential function $x \mapsto e^{2\pi i x}$.

**Algorithm 2:** Discrete Bootstrapping [BKSS24]

**Setting :** We rely on the encoding $X = \{0, 1, \ldots, t-1\} \hookrightarrow \mathbb{C}$.
**Input :** $\mathsf{ct} \in \mathcal{R}_Q^2$ a discrete CKKS ciphertext encrypting $\vec{z} \in X^{N/2}$. $f : \mathbb{Z}_t \to \mathbb{C}$ an arbitrary function.
**Output :** $\mathsf{ct}_{\mathsf{out}}$ encrypting $f(\vec{z})$.

1 $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{LUT}_f \circ \mathsf{EvalExp} \circ \mathsf{CtS} \circ \mathsf{ModRaise} \circ \mathsf{StC}(\mathsf{ct})$;
2 **return** $\mathsf{ct}_{\mathsf{out}}$.

More recently, [KN24] suggested using discrete bootstrapping to instantiate homomorphic modular reduction. Recall that the original CKKS does not have an inherent modular reduction, unlike other schemes like BGV/BFV/CGGI/DM. The main reason is that CKKS encodes messages in the least significant bits via DFT which is not compatible with modular reduction. However, [KN24] observed that we can indeed use the native modular reduction at the bottom modulus, by incorporating discrete bootstrapping as a subroutine.

As a summary of the previous works, we provide a simplified overview of discrete CKKS in Figure 2. In discrete CKKS, there are three separated ciphertext formats providing different operations. Here separated means that one can only perform one type of operation in a single format and needs to evaluate some (expensive) transformations to convert it to the other formats. To be specific, three formats can be denoted as arithmetic, coefficient, and interpolation formats. First, the arithmetic format refers to the usual CKKS

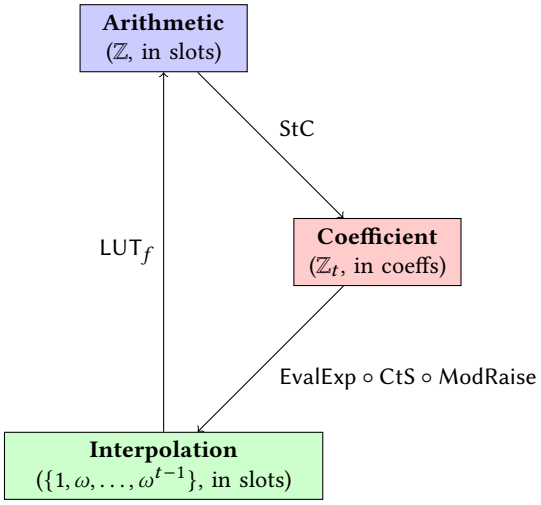


**Figure 2: Overview of Discrete CKKS**

slots-encoding which supports addition and multiplication inherited from the original CKKS. In terms of discrete computations, one can use this type for integer additions and multiplications. Second, the coefficients format refers to the coeffs-encoded CKKS ciphertext at the bottom modulus. This type of ciphertexts are used to perform RLWE modular reduction as suggested in [KN24] or conversions between RLWE/MLWE/LWE (one may refer to [BCK+23]). Third, interpolation formats refer to discrete CKKS ciphertexts encoded via roots-of-unity encoding as in [CKKL24], suitable for handling polynomial interpolations. It would usually be integrated with bootstrapping as in [BKSS24, AKP24], but one can specifically use this kind of encoding to handle arbitrary functions more efficiently. The arith-to-coeff, coeff-to-interpolate, interpolate-to-arith conversions can be instantiated with StC, EvalExp ∘ CtS ∘ ModRaise, and $\mathsf{LUT}_f$, respectively.

## 3 Proposed Method

In this section, we describe our strategy on how to efficiently enable integer computations using discrete CKKS. Let $t = 2^k$ be the target modulus for which we want to evaluate. As a high-level overview, we decompose $t$ via a digit decomposition by a smaller modulus $d = 2^\ell$ and handle modulo $t$ computation using several (extended) modulo $d$ computations. For instance, we may choose $k = 64$ and $\ell = 4$, leading to evaluating 64-bit computation using 16 digits of modulo 16 computations.

### 3.1 Scheme Description

We first illustrate how we put $k$-bit data into discrete CKKS ciphertexts. The message in our encoding is a vector of $k$-bit integers $\vec{z} = (z_0, z_1, \ldots, z_{N/2-1}) \in \mathbb{Z}_t^{N/2}$. To encrypt this vector, we use $u = k - \ell$ ciphertexts $\mathsf{ct}_0, \mathsf{ct}_1, \ldots, \mathsf{ct}_{u-1}$ where $\mathsf{ct}_i$ encrypts the $i$th digit of the base-$d$ representation of $\vec{z}$. That is,

$$\mathsf{ct}_i = \mathsf{Enc} \circ \mathsf{Ecd}(z_{0i}, z_{1i}, \ldots, z_{(N/2-1)i})$$

where $z_j = \overline{z_{j(u-1)} z_{j(u-2)} \cdots z_{j0}}\ (d)$ for each $0 \le j < N/2$. Here each ciphertext $\mathsf{ct}_i$ for $0 \le i < u$ is a valid discrete CKKS ciphertext

for $\mathbb{Z}_d$ computations. Therefore, we may use any operations in the prior works of discrete CKKS, such as arithmetic operations, table look-ups, and modular reduction. In particular, we mainly rely on addition, multiplication, and modular reduction, to handle ring operations over $\mathbb{Z}_d$ as well as the carry operation. To keep the discrete bootstrapping for $\mathbb{Z}_d$ efficient, we choose $d$ to be sufficiently small (e.g. $d \leq 2^8$). In terms of cleaning, we may use digit-wise cleaning for $\mathbb{Z}_d$ (e.g. cleaning for $d$th roots of unity as in [CKKL24]) which should guarantee the decryption precision. We elaborate on the encryption scheme as follows.

- **Encryption**: Let $\vec{z} = (z_0, z_1, \ldots, z_{N/2-1}) \in \mathbb{Z}_t^{N/2}$ be a vector of $k$-bit unsigned integers. The encryption $\mathsf{IntEnc} : \mathbb{Z}_t^{N/2} \to (\mathcal{R}_Q^2)^u$ is defined as
$$\mathsf{IntEnc}(\vec{z}) = \left(\mathsf{Enc} \circ \mathsf{Ecd}(z_{0i}, z_{1i}, \ldots, z_{(N/2-1)i})\right)_{0 \leq i < u}$$
where $z_j = \overline{z_{j(u-1)} z_{j(u-2)} \cdots z_{j0}}\ (d)$ is a base $d$ representation of $z_j$ for each $0 \leq j < N/2$.
- **Decryption**: Let $\mathsf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1, \ldots, \mathsf{ct}_{u-1})$ be a vector of discrete CKKS ciphertexts. The decryption $\mathsf{IntDec} : (\mathcal{R}_Q^2)^u \to \mathbb{Z}_t^{N/2}$ is defined as
$$\mathsf{IntDec}(\mathsf{ct}) = \sum_{i=0}^{u-1} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}_i) \rceil \cdot d^i$$
where we regard each $\mathbb{Z}_d$ integers as an element of $\mathbb{Z}$ and perform element-wise multiplications and additions.
- **Bootstrapping**: Given a ciphertext $\mathsf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1, \ldots, \mathsf{ct}_{u-1})$ encrypting a vector $\vec{z} \in \mathbb{Z}_t^{N/2}$, the (discrete) bootstrapping $\mathsf{Boot} : (\mathcal{R}_q^2)^u \to (\mathcal{R}_Q^2)^u$ is defined as
$$\mathsf{Boot}(\mathsf{ct}) = (\mathsf{IntBoot}_d(\mathsf{ct}_i))_{0 \leq i < u}$$
where $\mathsf{IntBoot} : \mathcal{R}_q^2 \to \mathcal{R}_Q^2$ refers to an identity discrete bootstrapping (i.e. functional bootstrapping evaluating an identity function) for $\mathbb{Z}_d$. For simplicity, we assume that this bootstrapping not only raises the modulus but also reduces the noise. See [BKSS24, AKP24] for details.

Theorem 3.1 (Encryption Correctness). *Let $\vec{z} = (z_0, z_1, \ldots, z_{N/2-1} \in \mathbb{Z}_t^u$ be a vector of $k$-bit unsigned integers. Then we have*
$$\mathsf{IntDec} \circ \mathsf{IntEnc}(\vec{z}) = \vec{z}.$$

Proof. Observe that
$$z_j = \sum_{i=0}^{u-1} z_{ji} \cdot d^i$$
from the base-$d$ representation of $z_j$. By the correctness of CKKS encryption and decryption, we have
$$\begin{aligned}\mathsf{IntDec} \circ \mathsf{IntEnc}(\vec{z}) &= \sum_{i=0}^{u-1} (z_{0i}, z_{1i}, \ldots, z_{(N/2-1)i}) \cdot d^i \\ &= \left(\sum_{i=0}^{u-1} z_{ji} \cdot d^i\right)_{0 \leq j < N/2} \\ &= (z_0, z_1, \ldots, z_{N/2-1}) = \vec{z}.\end{aligned}$$
□

Theorem 3.2 (Bootstrapping Correctness). *Let $\mathsf{ct} \in (\mathcal{R}_q)^u$ be a ciphertext encrypting a vector $\vec{z} \in \mathbb{Z}_t^{N/2}$ according to the above encryption. Then we have*
$$\mathsf{IntDec} \circ \mathsf{Boot}(\mathsf{ct}) = \vec{z}.$$

Proof. Let $\mathsf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1, \ldots, \mathsf{ct}_{u-1})$ and $\vec{z} = (z_0, z_1, \ldots, z_{N/2-1})$. Let $z_j = \overline{z_{j(u-1)} z_{j(u-2)} \cdots z_{j0}}\ (d)$ be the base-$d$ representation of $z_j$ for each $0 \leq j < N/2$. Let $\vec{w}_i = (z_{0i}, z_{1i}, \ldots, z_{(N/2-1)i}) \in \mathbb{Z}_d^{N/2}$ be a vector of $\ell$-bit unsigned integers for each $0 \leq i < u$. Then by the definition of the encryption, we have that $\mathsf{ct}_i$ is a valid discrete CKKS encryption of $\vec{w}_i$ for each $i$. Recall that the bootstrapping for ct is defined as an element-wise integer bootstrapping, which means that the components of $\mathsf{Boot}(\mathsf{ct})$ are the bootstrappings of $\mathsf{ct}_i$. Therefore, we have
$$\begin{aligned}\mathsf{IntDec} \circ \mathsf{Boot}(\mathsf{ct}) &= \sum_{i=0}^{u-1} \lfloor \mathsf{Dcd} \circ \mathsf{Dec} \circ \mathsf{Boot}(\mathsf{ct}_i) \rceil \cdot d^i \\ &= \sum_{i=0}^{u-1} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}_i) \rceil \cdot d^i = \vec{z}.\end{aligned}$$
The second equality follows from the correctness of the discrete bootstrapping over $\mathbb{Z}_d$. □

## 3.2 Arithmetic Operations

Based on the encryption scheme defined in the previous subsection, we describe how we define homomorphic arithmetic operations over $\mathbb{Z}_t$. We not only provide simple addition and multiplication but also some popular operations (e.g. comparison, bit shift) that are defined over unsigned integers. Let $\mathsf{IntMod}_d$ be a homomorphic modular reduction by $\mathbb{Z}_d$ as defined in [KN24] and $\mathsf{Carry}_d = (id - \mathsf{IntMod}_d)/d$.[4]

- **Reduction**: Let $\mathsf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1, \ldots, \mathsf{ct}_{u-1}) \in (\mathcal{R}_Q^2)^u$ be a vector of discrete CKKS ciphertexts encrypting vectors in $\mathbb{Z}^{N/2}$. The modular reduction of ct is sequentially defined as
$$\mathsf{Reduce}(\mathsf{ct}) = (\mathsf{ct}'_0, \mathsf{ct}'_1, \ldots, \mathsf{ct}'_{u-1})$$
where $\mathsf{ct}'_0 = \mathsf{IntMod}_d(\mathsf{ct}_0)$, $\mathsf{ct}''_0 = \mathsf{Carry}_d(\mathsf{ct}_0)$, and
$$\begin{aligned}\mathsf{ct}'_i &= \mathsf{IntMod}_d(\mathsf{ct}''_{i-1} + \mathsf{ct}_i) \\ \mathsf{ct}''_i &= \mathsf{Carry}_d(\mathsf{ct}''_{i-1} + \mathsf{ct}_i)\end{aligned}$$
for each $1 \leq i < u$. The result of Reduce encrypts a message
$$\sum_{i=0}^{u-1} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}_i) \rceil \cdot d^i$$
modulo $t$, i.e., reducing the ciphertext ct to the correct base-$d$ representation. Let $\mathsf{Carry}(\mathsf{ct})$ be defined as $\mathsf{ct}''_{u-1}$.
- **Addition**: Let $\mathsf{ct} = (\mathsf{ct}_i)_{0 \leq i < u}, \mathsf{ct}' = (\mathsf{ct}'_i)_{0 \leq i < u} \in (\mathcal{R}_Q^2)^u$ be vectors of ciphertexts each encrypting a $k$-bit unsigned integer according to the encryption scheme in Section 3.1. The addition of ct and ct′ is defined as
$$\mathsf{Add}(\mathsf{ct}, \mathsf{ct}') = \mathsf{Reduce}((\mathsf{ct}_i + \mathsf{ct}'_i)_{0 \leq i < u})$$

[4] Note that $\mathsf{Carry}_d(\mathsf{ct})$ may be at a lower level than ct. This may lead to modulus consumption and bootstrapping, but we consider the efficiency aspect in the later subsection.

where the addition on the right-hand side is the usual CKKS addition.

- **Subtraction**: Let $\mathsf{ct} = (\mathsf{ct}_i)_{0\le i<u}, \mathsf{ct}' = (\mathsf{ct}'_i)_{0\le i<u} \in (\mathcal{R}_Q^2)^u$ be vectors of ciphertexts each encrypting a $k$-bit unsigned integer according to the encryption scheme in Section 3.1. The subtraction of ct and ct′ is defined as
$$\mathsf{Sub}(\mathsf{ct}, \mathsf{ct}') = \mathsf{Reduce}((\mathsf{ct}_i - \mathsf{ct}'_i)_{0\le i<u})$$
where subtraction on the right-hand side is the usual CKKS subtraction.
- **Multiplication**: Let $\mathsf{ct} = (\mathsf{ct}_i)_{0\le i<u}, \mathsf{ct}' = (\mathsf{ct}'_i)_{0\le i<u} \in (\mathcal{R}_Q^2)^u$ be vectors of ciphertexts each encrypting a $k$-bit unsigned integer according to the encryption scheme in Section 3.1. The multiplication of ct and ct′ is defined as
$$\mathsf{Mult}(\mathsf{ct}, \mathsf{ct}') = \mathsf{Reduce}\left(\left(\sum_{j=0}^{i} \mathsf{Mult}(\mathsf{ct}_j, \mathsf{ct}'_{i-j})\right)_{0\le i<u}\right)$$
where Mult on the right-hand side refers to the usual CKKS multiplication.
- **Comparison**: Let $\mathsf{ct} = (\mathsf{ct}_i)_{0\le i<u}, \mathsf{ct}' = (\mathsf{ct}'_i)_{0\le i<u} \in (\mathcal{R}_Q^2)^u$ be vectors of ciphertexts each encrypting a $k$-bit unsigned integer according to the encryption scheme in Section 3.1. The comparison $\mathsf{ct} \ge \mathsf{ct}'$ is defined as
$$\mathsf{ct} \ge \mathsf{ct}' = \mathsf{Carry}(\mathsf{ct} - \mathsf{ct}') + 1$$
which outputs 1 for true and 0 for false.
- **Right Shift**: Let $\mathsf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1, \ldots, \mathsf{ct}_{u-1}) \in (\mathcal{R}_Q^2)^u$ be a vector of ciphertexts encrypting a $k$-bit unsigned integer according to the encryption scheme in Section 3.1. Let $0 \le m < k$ be an integer. The right shift of ct by $m$ denoted as $\mathsf{ct} << m$ is defined as
$$\mathsf{ct} << m = \left(\mathsf{IntMod}_d(\mathsf{ct}_{i-x} << y) + \mathsf{Carry}_d(\mathsf{ct}_{i-x-1} << y)\right)_{0\le i<u}$$
where $x = \lfloor m/\ell \rfloor$, $y = [m]_\ell$, and $\mathsf{ct}_i = 0$ for $i < 0$.
- **Left Shift**: Let $\mathsf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1, \ldots, \mathsf{ct}_{u-1}) \in (\mathcal{R}_Q^2)^u$ be a vector of ciphertexts encrypting a $k$-bit unsigned integer according to the encryption scheme in Section 3.1. Let $0 \le m < k$ be an integer. The left shift of ct by $m$ denoted as $\mathsf{ct} >> m$ is defined as
$$\mathsf{ct} >> m = \left(\mathsf{Carry}_d(\mathsf{ct}_{i+x} << (\ell - y)) + \mathsf{IntMod}_d(\mathsf{ct}_{i+x+1} << (\ell - y))\right)_{0\le i<u}$$
where $x = \lfloor m/\ell \rfloor$, $y = [m]_\ell$, and $\mathsf{ct}_i = 0$ for $i \ge u$.

Theorem 3.3 (Arithmetic Correctness). *Reduction, Addition, Subtraction, Multiplication, Comparison, Right Shift, and Left Shift defined above are correct.*

Proof. We give proof per each operation.

- Reduction: It suffices to show that
$$\sum_{i=0}^{u-1} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}'_i) \rceil \cdot d^i \equiv_t \sum_{i=0}^{u-1} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}_i) \rceil \cdot d^i.$$
By mathematical induction, we first check that
$$\lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}''_j) \rceil = \left\lfloor \frac{\sum_{i=0}^{j} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}_i) \rceil \cdot d^i}{d^j} \right\rfloor$$
holds for each $j$. Again, we use mathematical induction and get
$$\sum_{i=0}^{j} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}'_i) \rceil \cdot d^i = \left\lfloor \sum_{i=0}^{j} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}_i) \rceil \cdot d^i \right\rfloor_{d^{j+1}}$$
for each $j$. By plugging in $j = u - 1$, we finish the proof.
- Addition: We first check that
$$\sum_{i=0}^{u-1} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}_i + \mathsf{ct}'_i) \rceil \cdot d^i =$$
$$\sum_{i=0}^{u-1} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}_i) \rceil \cdot d^i + \sum_{i=0}^{u-1} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}'_i) \rceil \cdot d^i$$
from the decryption correctness of $(\mathsf{ct}_i)_i$ and $(\mathsf{ct}'_i)_i$. Then by the correctness of reduction, we have the correctness of addition.
- Subtraction: We first check that
$$\sum_{i=0}^{u-1} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}_i - \mathsf{ct}'_i) \rceil \cdot d^i =$$
$$\sum_{i=0}^{u-1} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}_i) \rceil \cdot d^i - \sum_{i=0}^{u-1} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}'_i) \rceil \cdot d^i.$$
- Multiplication: We first check that
$$\sum_{i=0}^{u-1} \left\lfloor \mathsf{Dcd} \circ \mathsf{Dec}\left(\sum_{j=0}^{i} \mathsf{Mult}(\mathsf{ct}_j, \mathsf{ct}'_{i-j})\right)\right\rceil \cdot d^i$$
$$= \sum_{i=0}^{u-1} \left(\sum_{j=0}^{i} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}_j) \rceil \odot \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}_{i-j}) \rceil \cdot d^i\right)$$
$$\equiv_t \left(\sum_{i=0}^{u-1} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}_i) \rceil \cdot d^i\right) \odot \left(\sum_{i=0}^{u-1} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}'_i) \rceil \cdot d^i\right)$$
by the decryption correctness. Then by the correctness of reduction, we have the correctness of multiplication.
- Comparison: We observe that given an integer $x$, the corresponding carry operation computes $(x - [x]_t)/t$. Let $\mathsf{ct}' - \mathsf{ct}$ encrypts an integer vector $\vec{z} \in \mathbb{Z}^{N/2}$ according to the decryption function in Section 3.1. Since both ct and ct′ encrypt unsigned $k$-bit integers, we have $-t < z_i < t$ for each entry $z_i$ for $\vec{z}$. In particular, the $i$th entry of ct is greater than or equal to the $i$th entry of ct′ if and only if $z_i \ge 0$. Note that the carry operation outputs 0 when $0 \le z_i < t$ and outputs 1 when $-t < z_i < 0$. Therefore, we have that $\mathsf{Carry}(\mathsf{ct} - \mathsf{ct}') + 1$ encrypts the right result.
- Right shift: It suffices to check the base-$d$ representation of the right shift of the original message. Let $\alpha = \overline{\alpha_{u-1}\alpha_{u-2}\cdots\alpha_0}$ be the base-$d$ representation of an unsigned $k$-bit integer $x$. Let $\beta = \overline{\beta_{u-1}\beta_{u-2}\cdots\beta_0}$ be the base-$d$ representation of $\alpha << m$. For each $i$,
$$\beta_i = [\alpha_{i-x} << y]_d + \left\lfloor \frac{\alpha_{i-x-1} << y}{d} \right\rfloor.$$
This proves the correctness.

- Left shift: It suffices to check the base-$d$ representation of the left shift of the original message. Let $\alpha = \overline{\alpha_{u-1}\alpha_{u-2}\cdots\alpha_0}$ be the base-$d$ representation of an unsigned $k$-bit integer $x$. Let $\beta = \overline{\beta_{u-1}\beta_{u-2}\cdots\beta_0}$ be the base-$d$ representation of $\alpha >> m$. For each $i$,

$$\beta_i = [\alpha_{i+x+1} << (\ell - y)]_d + \left\lfloor \frac{\alpha_{i+x} << (\ell - y)}{d} \right\rfloor.$$

  This proves the correctness.

□

## 3.3 Efficiency Analysis

We analyze the computational efficiency of each operation by counting the number of discrete bootstrapping. Note that discrete bootstrapping is usually $> 100$ times faster than any other homomorphic operations (e.g. addition, multiplication) so it gives a good approximation of the complexity.

Theorem 3.4 (Reduction Complexity). *Let*

$$\mathsf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1, \ldots, \mathsf{ct}_{u-1}) \in (\mathcal{R}_Q^2)^u$$

*be a vector of ciphertexts encrypting integer vectors in* $[0, d^m - d^{m-1}]$, *where* $Q \gg d^m$. *Then the number of discrete bootstrappings used to instantiate the* Reduce *operation is* $\leq um - m + 1$.

Proof. We provide an algorithm with at most $um - m + 1$ bootstrappings. Recall that Reduce consists of $u$ $\mathsf{IntMod}_d$ and $u - 1$ $\mathsf{Carry}_d$. The main issue is that the definition of $\mathsf{Carry}_d$ includes homomorphic division by $d$, which consumes modulus. To handle this issue, we keep all the $\mathsf{ct}_i$'s, $\mathsf{ct}'_i$'s, $\mathsf{ct}''_i$'s at the bootstrapping (output) level via discrete bootstrapping. To do this, we instantiate $\mathsf{Carry}_d$ with the iterative bootstrapping in [KN24]. The key observation is that $\mathsf{ct}''_{i-1} + \mathsf{ct}_i$ decrypts to an element

$$< d^m$$

which can be proved by mathematical induction on $i$.

- Base case $i = 0$: $\mathsf{ct}_i$ decrypts to $\leq d^m - d^{m-1} < d^m$.
- Assume $i$ and prove $i + 1$: Assume that $\mathsf{ct}''_{i-2} + \mathsf{ct}_i$ decrypts to $< d^m$. Then by the definition of $\mathsf{Carry}_d$, $\mathsf{ct}''_{i-1}$ decrypts to $< d^{m-1}$. Thus, $\mathsf{ct}''_{i-1} + \mathsf{ct}_i$ decrypts to

$$< d^{m-1} + d^m - d^{m-1} = d^m.$$

- By mathematical induction, we proved the desired property.

Therefore, $\mathsf{Carry}_d$ can be instantiated with $\mathsf{IntBoot}_d^{m-1}$ of [KN24, Algorithm 2] which requires $m - 1$ discrete bootstrappings. As a result, we instantiate Reduce with $u \cdot 1 + (u-1)\cdot(m-1) = um - m + 1$ discrete bootstrappings. □

Corollary 3.5 (Addition/Subtraction Complexity). *The number of bootstrappings needed for addition/subtraction is* $\leq 2u - 1$.

Proof. It follows directly from the fact that $[0, d) + [0, d) = [0, 2d-1)$ and $[0, d) - [0, d) = (-d, d)$ whose lengths are $\leq d^2 - d$. □

Corollary 3.6 (Comparison Complexity). *The number of bootstrappings needed for comparison is* $\leq 2u$.

Proof. Recall that the comparison operation does exactly the same as the subtraction except for the last carry step. Hence, we need $2u - 1 + 1 = 2u$ discrete bootstrappings. □

Theorem 3.7 (Multiplication Complexity). *Suppose that* $k \leq \ell \cdot 2^\ell$. *The number of bootstrappings needed for multiplication is* $\leq 3u - 2$.

Proof. Recall that the Mult operation heavily relies on the Reduce operation. Hence, it suffices to check the size of the underlying message of

$$M_i = \sum_{j=0}^{i} \mathsf{Mult}(\mathsf{ct}_j, \mathsf{ct}'_{i-j}).$$

As each $\mathsf{ct}_j$ and $\mathsf{ct}'_j$ decrypts to a $\ell$-bit unsigned integer, we have that each coordinate of the decryption of $M_i$ is less than or equal to

$$(2^\ell - 1)\cdot(2^\ell - 1)\cdot u = (d-1)^2 \cdot u \leq (d-1)^2 \cdot d < d^3 - d^2.$$

Hence, by Theorem 3.4, the number of bootstrapping needed is at most $3d - 2$. □

Theorem 3.8 (Shift Complexity). *The number of bootstrappings needed for the right/left shift operation is* $\leq u - x$.

Proof. We first prove the right shift operation. Observe that we need to compute $\mathsf{IntMod}_d(\mathsf{ct}_i << y)$ for $0 \leq i < u - x$ and $\mathsf{Carry}_d(\mathsf{ct}_i << y)$ for $0 \leq i < u - x - 1$. We can compute $\mathsf{IntMod}_d(\mathsf{ct}_i << y)$ for $0 \leq i < u - x$ using $u - x$ discrete bootstrappings. The remaining $\mathsf{Carry}_d$'s can be computed without introducing additional discrete bootstrappings. Hence $u - x$ in total.

The left shift operation works exactly the same except that the $\mathsf{IntMod}_d$ is needed for index $x \leq i < u$ and $\mathsf{Carry}_d$ is needed for index $x + 1 \leq i < u$. □

We summarize the analyses in Table 3.

| Operation Name | Number of Discrete Bootstrapping |
|---|---|
| Addition | $\leq 2u - 1$ |
| Subtraction | |
| Multiplication | [5] $\leq 3u - 2$ |
| Comparison | $\leq 2u$ |
| Right Shift | $\leq u - x$ |
| Left Shift | |

**Table 3: Number of discrete bootstrapping used in each operation.**

Next, we discuss some optimization techniques that can further reduce the computational complexity. The first optimization is to reduce the number of discrete bootstrapping by lazily bootstrap $\mathsf{Carry}_d$. For instance, one may bootstrap for every $\gamma$ (instead of 1) $\mathsf{Carry}_d$ when evaluating Reduce as in Theorem 3.4. Although it reduces the number of bootstrapping by a factor $\gamma$, it increases the modulus consumption by a factor $\gamma$ so one should carefully examine the efficiency to find a sweet spot.

[5] Under the assumption that $k \leq \ell \cdot 2^\ell$.

Theorem 3.9 (Lazy Bootstrap). *Let* $\mathsf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1, \ldots, \mathsf{ct}_{u-1}) \in (\mathcal{R}_Q^2)^u$ *be a vector of ciphertexts encrypting integer vectors in* $[0, d^m - d^{m-1}]$*, where* $Q \gg d^{\gamma m}$*. Then there is an instantiation of* Reduce *that uses at most* $m \cdot \lfloor (u-1)/\gamma \rfloor + u$ *bootstrappings and* $d^{m+\gamma-1}$ *of modulus.*

Proof. We may instantiate exactly as in the proof of Theorem 3.4, except that we lazily bootstrap for $\mathsf{Carry}_d$. To be explicit, we first obtain $\mathsf{ct}_i''$ (via $\mathsf{Carry}_d$ operation with lazy bootstrapping) sequentially and compute $\mathsf{ct}_i'$ at the end (via $u$ discrete bootstrappings). Here we may choose the ciphertext modulus of $\mathsf{ct}_i''$ to be a factor $d$ smaller than that of $\mathsf{ct}_{i-1}''$ for $\gamma \nmid i$ and bootstrap at $\gamma \mid i$. As a result, the ciphertext modulus for $\mathsf{ct}_i''$ becomes $Q/d^{\lfloor i \rfloor_\gamma}$. Recall that the modulus consumption for the last bootstrapping (i.e. the iterative bootstrapping in [KN24]) is $d^m$ and the number of bootstrapping is $m$. Hence the total modulus consumption is $d^m \cdot d^{\gamma-1} = d^{m+\gamma-1}$, and the total number of bootstrapping is

$$m \cdot (\#\text{ of bootstrap for } \mathsf{Carry}_d) + u \cdot (\#\text{ of bootstrap for } \mathsf{IntMod}_d)$$
$$= m \cdot \lfloor (u-1)/\gamma \rfloor + u.$$

□

Another observation is that we may reduce lazily, especially in addition/subtraction. For instance, if we add three ciphertext vectors $\mathsf{ct}, \mathsf{ct}', \mathsf{ct}'' \in (\mathcal{R}_Q^2)^u$, we may define the addition of them as

$$\mathsf{Add}(\mathsf{ct}, \mathsf{ct}', \mathsf{ct}'') = \mathsf{Reduce}((\mathsf{ct}_i + \mathsf{ct}_i' + \mathsf{ct}_i'')_{0 \le i < u})$$

hence saving one Reduce operation. We formalize this observation in the following theorem.

Theorem 3.10 (Lazy Addition). *Let* $\mathsf{ct}^0, \mathsf{ct}^1, \ldots, \mathsf{ct}^{v-1} \in (\mathcal{R}_Q^2)^u$ *be vectors of ciphertexts encrypting unsigned* $k$*-bit integer vectors according to the encryption scheme in Section 3.1, where* $v \le d$*. The addition of* $\mathsf{ct}^0, \mathsf{ct}^1, \ldots, \mathsf{ct}^{v-1}$ *can be instantiated as*

$$\mathsf{Reduce}\left(\left(\sum_{j=0}^{v-1} \mathsf{ct}_i^j\right)_{0 \le i < u}\right)$$

*with* $\le 2u - 1$ *discrete bootstrappings.*

Proof. By Theorem 3.4, we only need to check that for each $i$, $\sum_{j=0}^{v-1} \mathsf{ct}_i^j$ decrypts to an integer vector whose entries are in $[0, d^2 - d]$. To check this, we use the fact that $\mathsf{ct}_i^j$ decrypts to an integer vector whose entries are in $[0, d-1]$. Then, by summing up at most $d$ such ciphertexts, we get the desired property. □

Thus, we may lazily reduce after adding $\le d$ ciphertext vectors.

## 3.4 Implications

We discuss some direct applications of our method, namely fixed point arithmetic and arbitrary function evaluation over unsigned integers.

*3.4.1 Fixed Point Arithmetic.* As our scheme supports both multiplication and shift operations, it immediately supports fixed point multiplication which can be written as a combination of multiplication of two $m$ bit integers and right shift by $m$. However, we may do much better than this because the fixed point multiplication can be regarded as computing the high part of the multiplication.

- **Extended Reduction**: Let $\mathsf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1, \ldots, \mathsf{ct}_{2u-1}) \in (\mathcal{R}_Q^2)^{2u}$ be a vector of discrete CKKS ciphertexts encrypting vectors in $\mathbb{Z}^{N/2}$. The (extended) modular reduction of ct is sequentially defined as
$$\mathsf{ExtReduce}(\mathsf{ct}) = (\mathsf{ct}_0', \mathsf{ct}_1', \ldots, \mathsf{ct}_{2u-1}')$$
where $\mathsf{ct}_0' = \mathsf{IntMod}_d(\mathsf{ct}_0)$, $\mathsf{ct}_0'' = \mathsf{Carry}_d(\mathsf{ct}_0)$, and
$$\mathsf{ct}_i' = \mathsf{IntMod}_d(\mathsf{ct}_{i-1}'' + \mathsf{ct}_i)$$
$$\mathsf{ct}_i'' = \mathsf{Carry}_d(\mathsf{ct}_{i-1}'' + \mathsf{ct}_i)$$
for each $1 \le i < 2u$. The result of Reduce encrypts a message
$$\sum_{i=0}^{2u-1} \lfloor \mathsf{Dcd} \circ \mathsf{Dec}(\mathsf{ct}_i) \rceil \cdot d^i$$
modulo $t^2$, i.e., reducing the ciphertext ct to the correct base-$d$ representation.
- **Extended Multiplication**: Let $\mathsf{ct} = (\mathsf{ct}_i)_{0 \le i < u}, \mathsf{ct}' = (\mathsf{ct}_i')_{0 \le i < u} \in (\mathcal{R}_Q^2)^u$ be vectors of ciphertexts each encrypting a $k$-bit unsigned integer according to the encryption scheme in Section 3.1. The extended multiplication of ct and ct′ is defined as
$$\mathsf{ExtMult}(\mathsf{ct}, \mathsf{ct}') = \mathsf{ExtReduce}\left(\left(\sum_{j=\max(0,i-u+1)}^{\min(i,u-1)} \mathsf{Mult}(\mathsf{ct}_j, \mathsf{ct}_{i-j}')\right)_{0 \le i < 2u}\right)$$
where Mult on the right-hand side refers to the usual CKKS multiplication. The output vector contains the element-wise product of two input integer vectors, without modular reduction by $t$.

One may simply take the last $u$ entries of the result of the extended multiplication to instantiate a $k$-bit precision fixed point multiplication. Recall that CKKS often struggles to satisfy the security notions like IND-CPA$^D$ [LM21] or Threshold FHE [AJLA+12, BGG+18] as they cannot separate noise from the message. Although our fixed point arithmetic is much slower than the usual CKKS operations (as it involves bootstrapping), we may avoid huge flooding which increases the parameters greatly.

*3.4.2 Arbitrary Function Evaluation.* As our encoding stores a large integer after decomposing it into small pieces, we may directly benefit from the multi-precision arbitrary function evaluation in [AKP24, Section 5.3]. Since we maintain the digit-decomposed format, we do not need to perform digit extraction in [AKP24]. We restate (and slightly modify) the arbitrary function evaluation framework as follows.

Let $f : \mathbb{Z}_t \to \mathbb{C}$ be an arbitrary function, $\varphi : \mathbb{Z}_t \to \mathbb{Z}_d^u$ be the digit decomposition, and $\psi : \mathbb{Z}_d \to X = \{1, \omega, \ldots, \omega^{d-1}\}$ be defined as $x \mapsto \omega^{2\pi i x}$ where $\omega$ is a complex $d$th root of unity. Let $p \in \mathbb{C}[x_0, \ldots, x_{u-1}]$ be a multivariate polynomial that interpolates

the function $f \circ \varphi^{-1} \circ \psi^{-1} : X^u \to \mathbb{C}$. Then the homomorphic evaluation of $p$ instantiates $f$ (starting from complex roots of unity). We denote such operation as $\mathsf{MLUT}_f$. We illustrate the algorithm in Algorithm 3.

**Algorithm 3:** Arbitrary Function Evaluation [AKP24]

**Input** : $\mathsf{ct} = (\mathsf{ct}_i)_{0 \le i < u} \in (\mathcal{R}_Q^2)^u$ a ciphertext vector encrypting $\vec{z} \in \mathbb{Z}_t^{N/2}$. $f : \mathbb{Z}_t \to \mathbb{C}$ an arbitrary function.

**Output**: $\mathsf{ct}_{\mathsf{out}}$ encrypting $f(\vec{z})$.

1 **for** $i \leftarrow 0$ **to** $u - 1$ **do**
2 $\quad \mathsf{ct}'_i \leftarrow \mathsf{EvalExp} \circ \mathsf{CtS} \circ \mathsf{ModRaise} \circ \mathsf{StC}(\mathsf{ct}_i)$;
3 **end for**
4 $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{MLUT}_f(\mathsf{ct}'_0, \mathsf{ct}'_1, \ldots, \mathsf{ct}'_{u-1})$;
5 **return** $\mathsf{ct}_{\mathsf{out}}$.

It is worth mentioning that such functionality cannot be supported if one handles large integers directly inside CKKS. One reason is that univariate interpolation consumes much more multiplicative depths than multivariate interpolation, and another reason is that large precision polynomial interpolation is numerically unstable.

## 4 Experiments

We provide proof-of-concept implementations for the algorithms described in the previous section. We developed our code upon the lattigo library [lat24]. The experiments are run single-threaded on Intel i7-1360P at 2.6GHz with 9.72GB of RAM, running Ubuntu 22.04 with WSL. All of our FHE parameters satisfy $\approx 128$ bit security according to [BTPH22].

### 4.1 Description

We start with providing a parameter set used for our experiments, in Table 4. Here $N$ denotes the RLWE ring dimension, $\log QP$ denotes the maximum RLWE modulus for switching keys, $(h, \tilde{h})$ denotes the Hamming weights of the dense and sparse secrets [BTPH22], and *dnum* denotes the gadget rank for the switching keys. The lower table describes the moduli chain, where $\log q_i$ denotes the ciphertext modulus and $\log p_j$ denotes the auxiliary modulus for key switching. Base, StC, Mult, LUT, EvalExp, and CtS denote the moduli reserved for the corresponding operations in the discrete bootstrapping framework [BKSS24]. When denoted as $X \times Y$, it refers to using $Y$ many $X$-bit (NTT) moduli.

| $\log N$ | $\log QP$ | $(h, \tilde{h})$ | *dnum* |
|---|---|---|---|
| 15 | 768 | (192, 32) | 7 |

| $\log q_i$ | | | | | | $\log p_j$ |
|---|---|---|---|---|---|---|
| Base | StC | Mult | LUT | EvalExp | CtS | |
| 34 | $24 \times 3$ | 34 | $34 \times 5$ | $34 \times 8$ | $28 \times 3$ | $34 \times 3$ |

**Table 4: Parameter set for the experiment.**

Next, we elaborate on the scheme parameters of our experiments. We targeted 64-bit operations, decomposing it into 4-bit pieces, resulting in a total of 16 pieces. In other words, $k = 64$, $\ell = 4$, and $u = 16$. Note that $k \le 2^\ell \cdot \ell$, which satisfies the condition for Theorem 3.7. For the homomorphic modular reduction instantiation, we followed [KN24] which relies on the discrete bootstrapping from [BKSS24]. In particular, we used the cleaning interpolation (via Hermite interpolation) so that our bootstrapping achieves both modulus raising and cleaning.

We implemented addition, subtraction, multiplication, comparison, and right shift operation relying on the algorithms in Section 3.2. In particular, we used the strategy to bootstrap every time we compute $\mathsf{Carry}_d$, which was mainly used in the correctness proofs. This corresponds to the equality case in Table 3.

The experimental results are illustrated in Table 5. Here the error is measured by subtracting the decrypted result with the desired result. The precision denotes mean and minimum precision, which is computed as $-\log_2$ of the mean and maximum error. We observe that the total running time is roughly proportional to the number of bootstrappings as expected.

| | # Bootstrap | Running time | Precision |
|---|---|---|---|
| Addition | 31 | 191 sec | (14.8, 7.25) |
| Subtraction | 31 | 194 sec | (14.8, 7.89) |
| Multiplication | 46 | 293 sec | (11.9, 1.41) |
| Comparison | 32 | 197 sec | (13.9, 7.73) |

**Table 5: Experimental results for 64-bit homomorphic operations, based on the algorithms in Section 3.2 except shifting.**

To check the cost of left and right shift operations, we checked if it is linear in $x = \lfloor m/d \rfloor$ as expected. We fixed $y = 1$ for the right shift and $y = 2$ for the left shift, and plotted the graph in Figure 3. Not only the linearity but also the fact that the cost of shift is independent of $y$ was observed.

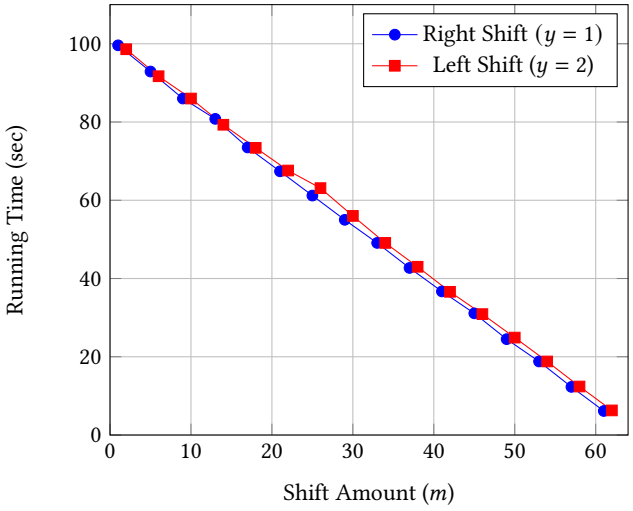


**Figure 3: Running time of right/left shift operations.**

## 4.2 Comparison with Prior Works

We compare our experimental results with previous works. We mostly focus on versatile schemes that efficiently support multiplication, comparison, and bootstrapping at the same time.

We mainly compare our algorithm with [Zam22] as it supports all the operations we discussed in Section 3.2. We ran their benchmark[6] in the same environment as our experiments (e.g. same machine, single-thread, etc.). The figures are illustrated in Table 6. Our algorithm outperforms [Zam22] by 2-3 orders of magnitude depending on the operations, in terms of throughput. Notably, our homomorphic 64-bit multiplication is more than 1700 times faster. To compare latency, our multiplication becomes favorable as soon as we have $\geq 10$ parallelism.

| | Latency (sec) | | Amortized time (ms) | | |
|---|---|---|---|---|---|
| | [Zam22] | Ours | [Zam22] | Ours | |
| Addition | 1.13 | 191 | 1130 | 11.7 | **96.6**x |
| Subtraction | 1.14 | 194 | 1140 | 11.8 | **96.6**x |
| Multiplication | 30.9 | 293 | 30900 | 17.9 | **1730**x |
| Comparison | 0.852 | 197 | 852 | 12.0 | **71.0**x |
| Right Shift[7] | 0.582 | 99.6 | 582 | 6.08 | **95.7**x |
| Left Shift | 0.581 | 98.6 | 581 | 6.02 | **96.5**x |

**Table 6: Comparison with TFHE-rs [Zam22] on 64-bit operations.**

Next, we dive deeper into homomorphic multiplications and see how the target modulus $t = 2^k$ affects the performance. We compare our multiplication with [Zam22] for $k = 4, 8, 16, 32, 64$, and check both latency and throughput. The figures can be seen in Table 7. Since our method is asymptotically almost linear in $k$ but [Zam22] is not, the performance difference is widened for large $k$. Although we experimented $k \leq 64$ (due to the constraint $k \leq \ell \cdot 2^\ell$), our method can be easily extended by increasing $\ell$. We expect that our method should perform even better for larger $k$.

| $k = \log_2 t$ | Latency (sec) | | Amortized time (ms) | | |
|---|---|---|---|---|---|
| | [Zam22] | Ours | [Zam22] | Ours | |
| 4 | 0.107 | 6.03 | 107 | 0.368 | **291**x |
| 8 | 0.446 | 25.3 | 446 | 1.54 | **290**x |
| 16 | 1.98 | 63.5 | 1980 | 3.88 | **510**x |
| 32 | 7.83 | 144 | 7830 | 8.79 | **891**x |
| 64 | 30.9 | 293 | 30900 | 17.9 | **1730**x |

**Table 7: Comparison with TFHE-rs [Zam22] on multiplications of different sizes.**

We move on to other possible approaches based on SIMD schemes. Note that most works do not cover multiplication, comparison, and bootstrapping at the same time. For instance, [ZYZ+24] achieves good performance for both homomorphic multiplication and comparison, but the encoding formats for the two operations are different and do not have efficient conversions from one to another. In this regard, we focus on decomposition-based works which seem to naturally support all three operations we are interested in.

For arithmetic operations (addition and multiplication), we compare with [HZY+22], the state-of-the-art method for radix-2 homomorphic addition/multiplication. We borrowed the figures from [HZY+22, Table V and VIII] which uses a similar environment for experiments. As illustrated in Table 8, our addition is $2-4$ times faster for computing 32/64-bit additions, and our multiplication is more than two orders of magnitude faster for evaluating 32-bit multiplication.

| | | $k$ | $\lambda$ | amortized time (ms) |
|---|---|---|---|---|
| Addition | [HZY+22] | 32 | 80 | 24 |
| | | 64 | | 24 |
| | Ours | 32 | $\approx 128$ | 5.98 |
| | | 64 | | 11.7 |
| Multiplication | [HZY+22] | 32 | 80 | 1020 |
| | Ours | 32 | $\approx 128$ | 8.79 |

**Table 8: Comparison with [HZY+22] on arithmetic operations.**

For homomorphic comparison, we compare with [TLW+21] which introduces the vector of field elements (VFE) encoding to efficiently evaluate large precision comparison. Although their latency is much better than ours, we outperform in terms of throughput due to the large parallelism of CKKS. The details are illustrated in Table 9.

| | $\lambda$ | # slots | latency (sec) | amortized time (ms) |
|---|---|---|---|---|
| [TLW+21][8] | $> 80$ | 256 | 20.2 | 78.8 |
| | | 128 | 20.5 | 160 |
| | | 16 | 4.75 | 297 |
| Ours | $\approx 128$ | 16384 | 197 | 12.0 |

**Table 9: Homomorphic comparison over $\mathbb{Z}_{2^{64}}$, compared with [TLW+21].**

## Acknowledgement

The work was supported by 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. We thank Professor Dan Boneh for helpful comments and suggestions, especially on applications and experiments.

## References

[AJLA+12] G. Asharov, A. Jain, A. López-Alt, E. Tromer, V. Vaikuntanathan, and D. Wichs. Multiparty computation with low communication, computation and interaction via threshold fhe. In *EUROCRYPT*, 2012.

[AKP24] A. Alexandru, A. Kim, and Y. Polyakov. General functional bootstrapping using CKKS. Cryptology ePrint Archive, Paper 2024/1623, 2024.

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

[6] We executed integer_bench from TFHE-rs [Zam22], commit hash ba105cd.

[7] For right/left shift, we considered the smallest $x$ which gives the largest running time for our method.

[8] Borrowed from [TLW+21, Table 1]. Excluded Step (c) from the total time. Note that [IZ21] provides a more efficient solution ($\approx$ 2x faster), but they did not measure timings for modulo $2^{64}$.

[BCK+23] Y. Bae, J. H. Cheon, J. Kim, J. H. Park, and D. Stehlé. HERMES: Efficient ring packing using MLWE ciphertexts and application to transciphering. In *CRYPTO*, 2023.

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

[BGG+18] D. Boneh, R. Gennaro, S. Goldfeder, A. Jain, S. Kim, P. M. R. Rasmussen, and A. Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In *CRYPTO*, 2018.

[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.

[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.

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

[CKK16] J. H. Cheon, M. Kim, and M. Kim. Optimized search-and-compute circuits and their application to query evaluation on encrypted data. *TIFS*, 2016.

[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.

[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.

[HZY+22] Z. Hou, N. Zhang, B. Yang, H. Wang, M. Zhu, S. Yin, S. Wei, and L. Liu. Efficient fhe radix-2 arithmetic operations based on redundant encoding. *IEEE TCAD*, 2022.

[IZ21] I. Iliashenko and V. Zucca. Faster homomorphic comparison operations for BGV and BFV. *PoPET*, 2021.

[KN24] J. Kim and T. Noh. Modular reduction in CKKS. Cryptology ePrint Archive, Paper 2024/1638, 2024.

[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, 2024. EPFL-LDS, Tune Insight SA.

[LM21] B. Li and D. Micciancio. On the security of homomorphic encryption on approximate numbers. In *EUROCRYPT*, 2021.

[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.

[QZL+19] Q. Qin, N. Zhang, L. Liu, S. Yin, and S. Wei. Addition circuit optimization using carry-lookahead and simd for homomorphic encryption. In *EDSSC*, 2019.

[TLW+21] B. H. M. Tan, H. T. Lee, H. Wang, S. Ren, and K. M. M. Aung. Efficient private comparison queries over encrypted databases using fully homomorphic encryption with finite fields. *IEEE TDSC*, 2021.

[XCWF16] C. Xu, J. Chen, W. Wu, and Y. Feng. Homomorphically encrypted arithmetic operations over the integer ring. In *ISPEC*, 2016.

[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.

[ZQH+21] N. Zhang, Q. Qin, Z. Hou, B. Yang, S. Yin, S. Wei, and L. Liu. Efficient comparison and addition for fhe with weighted computational complexity model. *IEEE TCAD*, 2021.

[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.