

# Efficient Homomorphic Arithmetic Logical Units with $O(1)$ Bootstrapping

Xuan Shen<sup>1,2</sup>, Zhihao Li<sup>\*3</sup>, Ruida Wang<sup>1,2</sup>, and Xianhui Lu<sup>1,2</sup>

<sup>1</sup> State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS

<sup>2</sup> School of Cyber Security, University of Chinese Academy of Sciences

<sup>3</sup> Ant Group

**Abstract.** Arithmetic logic units (ALU) combine word-level arithmetic with bitwise operations over encrypted data. The state-of-the-art scheme, GZ26 (CRYPTO'26), constructs an ALU in CKKS via triangle encoding, but has two limitations: the radix-2 design restricts packing capacity, while the arithmetic-to-Boolean (A2B) conversion requires  $O(\ell)$  functional bootstrappings for  $\ell$ -bit message without batching independent ciphertexts.

In this paper, we first generalize triangle encoding to arbitrary integer bases while preserving leveled modular arithmetic in CKKS. Larger bases reduce the number of slots required per message, allowing more messages to be packed into each ciphertext. For power-of-two digit bases, we introduce an A2B conversion using two functional bootstrapping after overflow refreshing, without batching independent ciphertexts. Our key insight is to transform triangle encoding into radix representation with unknown carries, from which candidate bits can be extracted using a single multi-value functional bootstrapping. Specifically, we first cancel the overflow shared by neighboring windows using one functional bootstrapping followed by linear operations, leaving expressions involving only radix digits and adjacent binary carries. Then we use a multi-value functional bootstrapping to jointly extract candidate bits and evaluate the local carry state, which determines how carries propagate between windows. Finally, we recover the canonical bits through parallel carry propagation and bit correction, using only leveled homomorphic operations.

We implement our scheme in OpenFHE and evaluate it for 64-bit, 128-bit, and 256-bit message. The experimental results show that using  $d = 2$  achieves the lowest latency for A2B conversion, yielding  $3.49\times-9.71\times$  speedups over GZ26 at the same packing capacity. In addition, using  $d = 32$  increases the packing capacity and achieves the best amortized performance, yielding  $8.15\times-28.47\times$  speedups.

**Keywords:** Fully homomorphic encryption · CKKS · triangle encoding · arithmetic-to-Boolean conversion · functional bootstrapping.

---

\* Corresponding author: [lzh458070@antgroup.com](mailto:lzh458070@antgroup.com)

# 1 Introduction

Homomorphic encryption enables computation directly on encrypted data without decryption first, thereby preserving data privacy throughout the computation. Gentry introduced bootstrapping [25], enabling homomorphic encryption to evaluate circuits of arbitrary depth and thereby realizing fully homomorphic encryption (FHE). Since then, a variety of FHE schemes have been developed with different plaintext spaces and computational models. BGV/BFV schemes [10,12,22] provide exact modular arithmetic over finite rings and naturally support SIMD-style computation (Single Instruction Multiple data). FHEW/TFHE schemes [17,18,20] are primarily designed for bit-level message and provide efficient bootstrapping for Boolean and lookup-table evaluation. CKKS schemes [15,16,14] were originally designed for approximate arithmetic over real or complex values together with SIMD ability, while recent works [1,5,4,21,6,29,19] have extended CKKS to support discrete integer computation and functional bootstrapping.

The ability to combine SIMD packing with discrete function evaluation makes CKKS a promising scheme for high-throughput encrypted computation. However, supporting general purpose programs requires an abstraction interface that supports both arithmetic and boolean evaluation. An encrypted arithmetic logic unit (ALU) provides a natural abstraction for this purpose by supporting arithmetic, comparison, shift, and bitwise operations over encrypted words. Nevertheless, designing such an encrypted ALU raises two challenges. First, modern processors commonly operate on 32 and 64 bit words, whereas efficient discrete homomorphic computation is typically performed over much smaller message precision. This necessitates a representation of high-precision messages that admits efficient homomorphic evaluation. Second, arithmetic and Boolean operations favor different representations. Word-mode representations support efficient addition and multiplication, whereas Boolean operations are more naturally evaluated on canonical bits. Therefore, an encrypted ALU must support efficient conversion between representations suited to arithmetic and Boolean operations.

Recent FHE systems for large integers adopt different representations with distinct properties, as summarized in Table 1. Among them, Gao and Zheng [23] build on a polynomial quotient ring isomorphic to integer ring and proposed triangle encoding to realize the corresponding integer arithmetic in CKKS. This construction supports leveled operations in both arithmetic and Boolean mode with SIMD, together with conversions between triangle encoding and a canonical bit representation. It is worth noting that Boolean-to-arithmetic (B2A) conversion requires only homomorphic linear transformations and scalar multiplications, without bootstrapping. However, for arithmetic-to-Boolean (A2B) conversion, the scheme employs iterative window extraction. Consequently, converting an  $\ell$ -bit message requires  $O(\ell)$  sequential functional bootstrappings without batching different ciphertexts. This naturally raises the following question:

*Can we construct a more efficient ALU based on triangle encoding that supports leveled arithmetic and Boolean operations and requires only  $O(1)$  functional bootstrappings for A2B conversion?*

## 1.1 Contribution

This paper answers this question affirmatively by constructing a homomorphic ALU that combines general-base triangle encoding with an efficient A2B conversion, as shown in Figure 1. And our contributions are threefold.

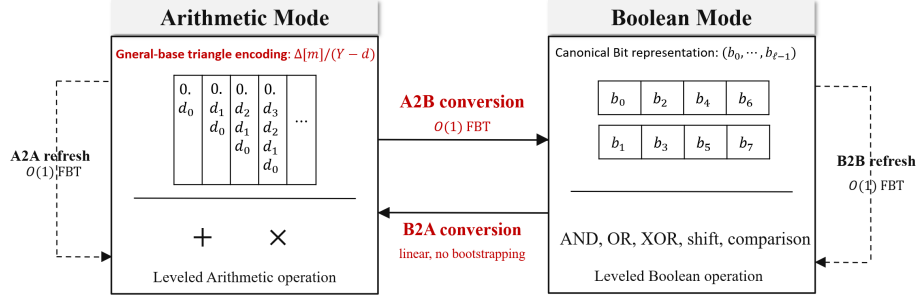


**Fig. 1.** Overview of the homomorphic ALU framework. Both modes represent the same message  $m \in \mathbb{Z}_{d^n}$ . The arithmetic mode uses the general-base triangle encoding, whereas the Boolean mode uses canonical bit representation. A2A refresh, A2B conversion, and B2B refresh each require  $O(1)$  functional bootstrapping, while B2A conversion is linear without bootstrapping.

**General-base triangle encoding.** We extend binary triangle encoding to arbitrary integer bases  $d \geq 2$  using the quotient ring  $\mathcal{Z}_t = \mathbb{Z}[Y]/(Y^n - Y + d, t)$  with  $t = Y - d$ , and prove that the resulting encoding preserves the triangular coefficient structure and supports leveled modular integer arithmetic with SIMD packing in CKKS. Larger digit bases reduce the encoding dimension, allowing more messages to be packed into each ciphertext. However, it also increase the amplification of CKKS encoding errors during message decoding. Therefore, we derive an explicit bound on the resulting decoding error and establish sufficient conditions for correct decoding. Our analysis provides explicit criteria for selecting the digit base to achieve the desired packing capacity while ensuring correct decoding.

**A2B conversion with  $O(1)$  bootstrapping.** Following integer overflow refreshing, our A2B conversion requires two functional bootstrappings, independent of the message precision and without batching different ciphertexts. Our main insight is to convert triangle encoding into radix encoding, allowing all bits to be extracted with a single small precision functional bootstrapping. However, the special form of the constant coefficient and the integer overflow introduced by arithmetic evaluation prevent scaled coefficient differences from directly yielding radix digits. We handle the constant coefficient through linear preprocessing.

After refreshing the overflow, we choose the window size according to the refreshed overflow bound and cancel the overflow shared by neighboring windows using one functional bootstrapping followed by linear operations. This choice of window size ensures that the resulting expressions involve only radix digits and adjacent binary carries. Then we use multi-value functional bootstrapping to jointly extract candidate bits and evaluates the carry states. Finally, we recover the canonical bits through parallel carry propagation and bit correction, using only leveled homomorphic operations.

**Implementation and evaluation.** We implement the proposed construction in OpenFHE and evaluate it for message sizes of 64, 128, and 256 bits. Among the evaluated bases, A2B achieves the lowest latency with  $d = 2$ , requiring 71.47 seconds for 64-bit messages and 104.02 seconds for 256-bit messages. In contrast,  $d = 32$  minimizes amortized latency, requiring 29.84 and 138.51 milliseconds per message at 64 and 256 bits, respectively. Compared with GZ26 [23], the corresponding latency and amortized speedups are  $3.49\times$  and  $8.15\times$  at 64 bits, increasing to  $9.71\times$  and  $28.47\times$  at 256 bits.

## 1.2 Technique overview

**General-base triangle encoding.** Our generalization consists of the following two parts.

*Algebraic construction.* We generalize the algebraic construction of GZ26 [23] by choosing  $F(Y) = Y^n - Y + d$  and  $t = Y - d$ . Since  $F(d) = d^n$ , evaluation at  $Y = d$  identifies  $\mathbb{Z}[Y]/(F, t)$  with  $\mathbb{Z}_{d^n}$ . Then, arithmetic on the flattened digit polynomials produces additional terms in the ideal  $(t)$ . Dividing by  $t$  in  $\mathcal{Z}_{\mathbb{R}} = \mathbb{R}[Y]/(F)$  converts these terms into integer overflow, giving the representation  $f = \Delta I + \frac{\Delta}{t}[m] + e$ , and

$$f_0 = \Delta I_0 + \frac{\Delta m}{d^n} - \frac{\Delta d_0}{d} + e_0, \quad f_i = \Delta I_i - \Delta \sum_{j=0}^i \frac{d_j}{d^{i-j+1}} + e_i, \quad 1 \leq i < n.$$

The resulting encoding preserves the triangular coefficient structure and supports leveled modular arithmetic, with correct decoding guaranteed whenever  $\|te\|_{\infty} < \Delta/2$ .

*CKKS embedding and error control.* For even  $n$ , evaluating a triangle polynomial at one root from each conjugate pair of  $F$  represents a message using  $n/2$  complex values. This embedding converts polynomial addition and multiplication into componentwise operations, allowing CKKS to process  $N/n$  messages in SIMD when  $n \mid N$ . A larger digit base  $d$  can reduce the number of CKKS slots needed per message, but amplifies the effect of plaintext rounding error during decoding. Our analysis shows that  $\Delta > 4dN$  suffices for correct decoding after CKKS plaintext rounding, giving an explicit relation between the digit base  $d$  and the required scale  $\Delta$ .

**A2B conversion with  $O(1)$  bootstrapping.** Our A2B conversion proceeds in following steps.

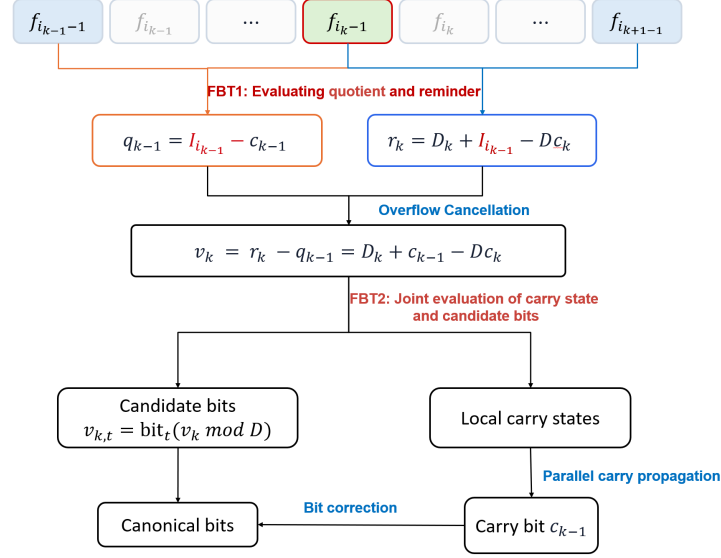


**Fig. 2.** Overview of our A2B conversion via shared overflow cancellation. We assume that the arithmetic overflow has already been refreshed using  $\text{Refresh}_I$ . Blue text denotes leveled homomorphic operations.

*Coefficient preprocessing and overflow refreshing.* We note that without integer overflow and the additional  $m/d^n$  term in the normalized constant coefficient  $f_0$ , scaling and subtracting consecutive triangle coefficients recovers the radix digits independently. To deal with this problem, we apply  $\text{Refresh}_I$  to preprocess the coefficients as follows. Since  $m/d^n$  appears in the constant and the  $n-1$ -th coefficients, we replace coefficient  $f_0$  by  $f_0 + f_{n-1}$ , cancelling the  $m/d^n$  term in  $f_0$  and the remaining message term is  $-d_0/d$ . We then remove the existing integer overflow  $I$  by truncation and introduce fresh overflow through modulus raising with sparse secret encapsulation, returning the preprocessed coefficients in CKKS slots.

*Overflow cancellation with binary carries.* Assuming  $\|I\|_\infty < B/2$  after  $\text{Refresh}_I$ , we add a public offset  $B/2$  so that  $I'_i = I_i + B/2 \in (0, B)$  and choose a window length  $\omega \mid n$  such that  $D = d^\omega \geq B$ , dividing the message into  $L = n/\omega$  windows. Let  $D_k \in [0, D-1]$  denote the value of window  $k$ , and  $i_k = (k+1)\omega - 1$  be its last coefficient index. Ignore the error  $e$ , the scaled coefficient differences satisfy

$$s_0 = D_0 - DI'_{i_0}, \quad s_k = D_k + I'_{i_{k-1}} - DI'_{i_k}, \quad 1 \leq k < L.$$

We observe that the remainder  $r_k$  and the preceding quotient  $q_{k-1}$  contain the same overflow term,

$$r_k = s_k \bmod D = D_k + I'_{i_{k-1}} - Dc_k, \quad q_{k-1} = \frac{r_{k-1} - s_{k-1}}{D} = I'_{i_{k-1}} - c_{k-1},$$

where  $c_0 = 0$  and  $c_k = \lfloor (D_k + I'_{i_{k-1}}) / D \rfloor$  for  $1 \leq k < L$ . Since  $0 \leq D_k + I'_{i_{k-1}} < 2D$ , each  $c_k$  is a bit. We use functional bootstrapping to compute all remainders  $r_k = s_k \bmod D$  and recover the quotients  $q_k = (r_k - s_k) / D$  using only linear operations. Subtracting the preceding quotient from the current remainder eliminates their shared overflow, yielding

$$v_k = r_k - q_{k-1} = D_k + c_{k-1} - Dc_k. \quad (1)$$

*Joint evaluation of carry states and candidate bits.* Our second observation is that the sign of  $v_k$  determines the outgoing carry  $c_k$ . Specifically, by eq. (1),  $v_k < 0$  implies  $c_k = 1$ ,  $v_k > 0$  implies  $c_k = 0$  and when  $v_k = 0$ , the outgoing carry satisfies  $c_k = c_{k-1}$ . Thus, the sign of  $v_k$  characterizes the dependence of  $c_k$  on  $c_{k-1}$  without requiring the value of  $c_{k-1}$  to be known. Moreover,  $v_k \bmod D = (D_k + c_{k-1}) \bmod D$ . Hence,  $D_k = v_k \bmod D$  when  $c_{k-1} = 0$ , whereas  $D_k = ((v_k \bmod D) - 1) \bmod D$  when  $c_{k-1} = 1$ . Therefore the binary expansion of  $v_k \bmod D$  provides candidate bits for window  $k$ . Both the carry state and the candidate bits are functions of the same input  $v_k$ , allowing us to use one multi-value functional bootstrapping to evaluate carry states and candidate bits jointly for all windows in parallel.

*Parallel carry recovery and bit correction.* We recover all carries through parallel carry propagation, combining the carry states obtained above in a binary-tree structure. At each level of the tree, the carry states of adjacent blocks are combined to obtain the carry state of each larger block. Since this composition is associative, the combined carry behavior from the initial window to each subsequent window can be computed in parallel using  $O(\log \ell)$  multiplicative levels. With  $c_0 = 0$ , all carry bits  $c_k$  are then recovered from these states through linear operations.

For each window  $k \geq 1$ , we have the candidate bits represent  $v_k \bmod D$ . The recovered incoming carry determines whether the window's candidate value needs to be decreased by one. Specifically, if  $c_{k-1} = 0$ , these bits already represent  $D_k$ , otherwise, recovering  $D_k$  requires subtracting one modulo  $D$ . Starting from the least significant position, this subtraction changes consecutive zero bits to ones, changes the first nonzero bit to zero, and leaves all more significant bits unchanged. If all candidate bits are zero, the modular subtraction produces all ones. This correction can be evaluated with leveled additions and multiplications and the initial window requires no correction. Consequently, both carry recovery and bit correction use only leveled homomorphic operations, yielding the canonical bits without an additional functional bootstrapping.

### 1.3 Related Work

Existing works supporting homomorphic large integer computation can be broadly classified into three categories according to their underlying FHE scheme and encoding representations. Table 1 compares our work to previous works for supporting large integer computation.

*Radix-based representations.* The first class of schemes [27,28,13] represents large integer as a vector of digits obtained through radix decomposition. Kim [27] encrypts different digits of a large integer in separate CKKS ciphertexts. Arithmetic is performed using grade school addition and multiplication that computes iteratively from the lowest digit. A subsequent work [28] instead embeds the digits into polynomial coefficients, allowing native polynomial multiplication to perform digit convolution. This scheme uses lazy reduction to obtain redundant representatives, but still requires sequential carry propagation to recover canonical digits. Then, Cha et al. [13] further combines lightweight lazy carry with parallel carry propagation to reduce the cost of digit canonicalization. Once canonical radix digits are obtained, all bits can be extracted using only one multi-value functional bootstrapping.

*RNS-based representations.* The second class of schemes adopts Residue Number System (RNS) decomposition for large integer representation, enabling arithmetic to be performed independently over the residue components. Boneh and Kim [7] represent a large integer through nested residue-number systems across CKKS slots. The core algorithm, homomorphic base conversion, combines small residue components into a representation that supports arithmetic modulo large, possibly non-smooth moduli while retaining many independent instances in one RLWE ciphertext. In addition, this scheme also uses lazy modular reduction and returns redundant representatives and does not provide efficient support for non-arithmetic operations such as comparison.

*Polynomial quotient ring representations.* Another line of work [11,24] uses polynomial plaintext moduli to construct polynomial quotient rings isomorphic to the target integer rings. Under the ring isomorphism, large integer addition and multiplication can be realized directly through polynomial quotient ring operations and evaluated using only leveled homomorphic operations. REFHE [11] instantiates this approach through a BGV-style construction over a non-cyclotomic ring, with an ideal plaintext modulus enabling native arithmetic over  $\mathbb{Z}_{2^\ell}$ . Its bootstrapping procedure uses TFHE-style programmable bootstrapping to extract the message bits sequentially, from least to most significant. The paper considers a single machine word per ciphertext and does not explore SIMD batching. Geelen and Vercauteren [24] introduce GBFV, which generalizes BFV by replacing the scalar plaintext modulus with a polynomial while retaining a cyclotomic ring. The scheme supports native SIMD arithmetic modulo large cyclotomic primes with lower noise growth than conventional BFV. The paper also presents a BFV-based bootstrapping procedure that combines conversion between GBFV and BFV with adapted digit removal.

*Concurrent work.* During the same period, Wei et al. [32] concurrently achieve A2B conversion via redundant binary digit canonicalization, using two functional bootstrappings after overflow refreshing for power-of-two moduli. Their implementation uses a higher CKKS multiplicative depth than our  $d = 2$  configurations.

**Table 1.** Comparison of FHE schemes for large integer computation.

| Scheme           | Representation      | FHE  | Type   | Arith                | Bool              | A2B       | B2A    |
|------------------|---------------------|------|--------|----------------------|-------------------|-----------|--------|
| Kim25a [27]      | Radix decomposition | CKKS | SIMD   | $O(k \log k)$        | Leveled w/ $O(1)$ | $O(1)$    | Linear |
| Kim25b [28]      |                     | CKKS | SIMD   | $O(\log k)$          | Leveled w/ $O(1)$ | $O(k)$    | Linear |
| CPL25 [13]       |                     | CKKS | SIMD   | $O(\log k)$          | Leveled w/ $O(1)$ | $O(1)$    | Linear |
| BK25 [7]         | RNS decomposition   | CKKS | SIMD   | Leveled              | N/A               | N/A       | N/A    |
| REFHE [11]       | Polynomial          | BGV  | Scalar | Leveled w/ $O(\ell)$ | $O(\ell)$         | $O(\ell)$ | Repack |
| GZ26 [23]        | quotient ring       | CKKS | SIMD   | Leveled w/ $O(1)$    | Leveled w/ $O(1)$ | $O(\ell)$ | Linear |
| <b>This work</b> |                     | CKKS | SIMD   | Leveled w/ $O(1)$    | Leveled w/ $O(1)$ | $O(1)$    | Linear |

Here,  $\ell$  is the word precision,  $k$  is the number of radix digits “Leveled” means that evaluation does not require immediate bootstrapping; “w/” specifies the refreshing cost. “Linear” denotes conversion using only linear operations.

By using larger bases for triangle encoding, our construction further increases packing capacity, improving amortized performance.

## 1.4 Organization

Section 2 introduces the notation, CKKS, and functional bootstrapping. Section 3 develops general-base triangle encoding, its CKKS embedding, and the associated homomorphic operations and linear transformations. Section 4 presents our A2B conversion via shared overflow cancellation, analyzes its correctness and complexity, and outlines the linear B2A conversion. Section 5 reports the implementation and performance evaluation and Section 6 concludes the paper.

## 2 Preliminaries

### 2.1 Notation

For an integer  $a$  and a positive integer  $q$ , we write  $a \bmod q \in \{0, \dots, q-1\}$  for the least nonnegative residue of  $a$  modulo  $q$ . For a nonnegative integer  $a$ , we write  $\text{bits}_i(a) = \lfloor a/2^i \rfloor \bmod 2$  for its  $i$ -th bit. All logarithms are to base two. Scalars are written in lowercase, whereas vectors are written in boldface. For a nonnegative integer  $d$ , we write  $[0, d] = \{0, \dots, d\}$ . Let  $N$  be a power of two and define  $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ . The rounding operator  $\lfloor \cdot \rfloor$  denotes rounding to the nearest integer and is applied coefficientwise to polynomials and elementwise to vectors. We use the infinity norm  $\|\cdot\|_\infty$  for the norm operator  $\|\cdot\|$  on polynomials, vectors, or matrices.

### 2.2 CKKS scheme

CKKS [16,15] is an RLWE-based homomorphic encryption scheme supporting approximate SIMD arithmetic over complex numbers. Let  $q'_0, \dots, q'_L$  be primes

satisfying  $q'_i \equiv 1 \pmod{2N}$  and define  $Q'_\ell = \prod_{i=0}^{\ell} q'_i$ ,  $\mathcal{R}_{Q'_\ell} = \mathcal{R}/Q'_\ell \mathcal{R}$ . At level  $\ell$ , a CKKS ciphertext is  $\text{ct} = (a, b) \in \mathcal{R}_{Q'_\ell}^2$ . Let  $\mathcal{R}_{\mathbb{R}} = \mathcal{R} \otimes_{\mathbb{Z}} \mathbb{R}$  and  $\zeta_{2N} = \exp(\pi i/N)$ . The full canonical embedding  $\sigma : \mathcal{R}_{\mathbb{R}} \rightarrow \mathbb{H}$  maps  $a$  to  $(a(\zeta_{2N}^j))_{j \in \mathbb{Z}_{2N}^*}$ , where  $\mathbb{H} = \{(z_j)_{j \in \mathbb{Z}_{2N}^*} \in \mathbb{C}^N : z_{-j} = \overline{z_j}\}$ . Let  $\tau_N : \mathbb{H} \rightarrow \mathbb{C}^{N/2}$  retain one coordinate from each conjugate pair and denote  $\sigma_{\mathcal{R}} = \tau_N \circ \sigma$  for the slot embedding.

- **Encode**. Given  $\mathbf{m} \in \mathbb{C}^{N/2}$ , the encoding procedure maps the slot vector to a plaintext polynomial as  $\text{pt} = \text{Encode}(\mathbf{m}) = \lfloor \sigma^{-1}(\Delta \cdot \tau_N^{-1}(\mathbf{m})) \rfloor$ .
- **Decode**. Given an encoded plaintext polynomial  $\text{pt} \in \mathcal{R}$ , the decoding procedure returns  $\text{Decode}(\text{pt}) = \tau_N(\sigma(\text{pt})/\Delta)$ .
- **Encrypt**. Given an encoded plaintext polynomial  $\text{pt} \in \mathcal{R}$  and a secret key  $s$ , the encryption procedure samples  $a \leftarrow \mathcal{R}_{Q'_L}$  uniformly and a small error  $e$ , and outputs  $\text{Enc}_s(\text{pt}) = \text{ct} = (a, b) = (a, -as + \text{pt} + e) \bmod Q'_L$ .
- **Decrypt**. Given  $\text{ct} = (a, b) \in \mathcal{R}_{Q'_\ell}^2$ , decryption first computes  $\text{pt}' = b + as \bmod Q'_\ell$  and then returns  $\text{Decode}(\text{pt}')$ , which recovers an approximation of the original slot vector.

CKKS provides the following homomorphic operations.

- **CKKS.Add** computes an approximate slotwise sum of two encrypted vectors.
- **CKKS.Mult** computes their approximate slotwise product.
- **CKKS.Rescale $_{q'_\ell}$**  removes  $q'_\ell$  from the ciphertext modulus, divides the scale accordingly, and consumes one level.
- **CKKS.Conjugate** computes an approximate encryption of the componentwise complex conjugate.
- **CKKS.Rotate(ct,  $r$ )** cyclically rotates the encrypted slot vector by  $r$  positions to the right.
- **CKKS.Truncate $_{Q'_\ell \rightarrow q'_0}$**  removes integer overflow from a coefficient-encoded plaintext while retaining its fractional message.
- **CKKS.ModRaise $_{q'_0 \rightarrow Q'_L}$**  lifts the ciphertext modulus without rescaling and may introduce an additional term  $q'_0 I'$ .
- **CKKS.ModSwitch(ct,  $q$ )** switches the ciphertext to modulus  $q$  while tracking its scale.

### 2.3 Discrete and Multi-Value Functional Bootstrapping

Functional bootstrapping refreshes a ciphertext while evaluating a public real-valued LUT  $f : \mathbb{Z}_p \rightarrow \mathbb{R}$  independently in every slot  $[1, 5, 4, 21, 6, 29, 19]$ . We assume that the input and output ciphertext are slot encoded. We first apply **SlotsToCoeffs** to move the message to the coefficient representation. After modulus raising, we apply **CoeffsToSlots** to obtain a ciphertext whose slots encoding  $\frac{m}{p} + I + \varepsilon$ , where  $I \in \mathbb{Z}$  is the integer overflow introduced by modulus raising. The periodic map  $x \mapsto \exp(2\pi i x)$  removes  $I$  and maps the  $p$  possible messages to  $p$ -th roots of unity. Each lookup table is represented by a first-order Hermite interpolation polynomial and are evaluated using the Paterson–Stockmeyer method [31]. When evaluating multiple polynomials on the same input, this method allows the powers indexed by  $\mathcal{I}_{\text{PS}}$  to be precomputed once and shared across all polynomial evaluations.

---

**Algorithm 1:** Multi-value functional bootstrapping

---

**Input:** A slot-encoded RLWE ciphertext  $\text{ct} = \text{Enc}(\mathbf{m})$  with  $\mathbf{m} \in \mathbb{Z}_p^{N/2}$ ;  
functions  $f_i : \mathbb{Z}_p \rightarrow \mathbb{R}$  for  $1 \leq i \leq k$ ; and output ciphertext modulus  $Q$ .  
**Output:** Slot-encoded ciphertexts  $\{\text{ct}_{\text{out},i}\}_{i=1}^k$  encrypting  $\{f_i(\mathbf{m})\}_{i=1}^k$ .

```

1  $\text{ct}_0 \leftarrow \text{SlotsToCoeffs}(\text{ct})$  ; // Move the slot-encoded message to the coefficient representation
2  $\text{ct}_1 \leftarrow \text{CKKS.ModSwitch}(\text{ct}_0, q'_0)$  ; // Switch to the base modulus of the bootstrap
3  $\text{ct}_2 \leftarrow \text{CKKS.ModRaise}(\Delta \text{ct}_1 / q'_0, Q'_L)$  ; // Normalize the message and raise the ciphertext
   modulus
4  $\text{ct}_3 \leftarrow \text{CoeffsToSlots}(\text{ct}_2)$  ; // Expose the normalized values  $m/p + I$  in the CKKS slots
5  $\text{ct}_{\text{exp}} \leftarrow \text{EvalExp}_p(\text{ct}_3)$  ; // Approximate  $x \mapsto \exp(2\pi i x)$  on the normalized nodes
6  $\{\text{ct}_{\text{pow},j}\}_{j \in \mathcal{I}_{\text{PS}}} \leftarrow \text{EvalPower}(\text{ct}_{\text{exp}}, p)$  ; // Precompute the powers shared by all lookup tables
7 for  $i \in \{1, \dots, k\}$  do
8    $\text{ct}_{4,i} \leftarrow \text{PolyPS}(\{\text{ct}_{\text{pow},j}\}_{j \in \mathcal{I}_{\text{PS}}}; \text{LUT}_{f_i})$  ; // Evaluate the interpolation polynomial
   representing  $f_i$ 
9    $\text{ct}_{5,i} \leftarrow \frac{1}{2}(\text{ct}_{4,i} + \text{CKKS.Conjugate}(\text{ct}_{4,i}))$  ; // Retain the real part of the function
   output
10   $\text{ct}_{\text{out},i} \leftarrow \text{CKKS.ModSwitch}(\text{ct}_{5,i}, Q)$  ; // Return the slot ciphertext to the target
   ciphertext modulus
11 return  $\{\text{ct}_{\text{out},i}\}_{i=1}^k$ 

```

---

For  $k = 1$ , Algorithm 1 is an ordinary functional bootstrap, and we write

$$\text{FBT}_p(\text{ct}; \text{LUT}_f) = \text{MV-FBT}_p(\text{ct}; \{\text{LUT}_f\}).$$

For  $k > 1$ , the same bootstrap preprocessing is shared by all  $k$  lookup tables.

### 3 General-Base Triangle Encoding in CKKS

We generalize the triangle encoding of [23] from  $t = Y - 2$  to  $t = Y - d$  and establish that the triangular structure and arithmetic properties extend to general bases. Within the CKKS framework, we derive the corresponding embedding and homomorphic operations for general arithmetic modulo.

#### 3.1 General-Base Triangle Encoding

Let  $d \geq 2$  be the digit base and let  $n \geq 2$  be the number of base- $d$  digits in one message. To avoid notational ambiguity with the CKKS plaintext space defined over the indeterminate  $X$ , we define the arithmetic ring  $\mathcal{Z}$  using the indeterminate  $Y$  as

$$F(Y) = Y^n - Y + d, \quad \mathcal{Z} = \mathbb{Z}[Y]/(F), \quad \mathcal{Z}_{\mathbb{R}} = \mathcal{Z} \otimes_{\mathbb{Z}} \mathbb{R} \cong \mathbb{R}[Y]/(F).$$

In the arithmetic ring  $\mathcal{Z}$ , reduction modulo  $F(Y)$  gives  $Y^{n+r} = (Y - d)Y^r$  for  $r \geq 0$ . Setting  $t = Y - d$ , we further define the quotient ring of  $\mathcal{Z}$  by the ideal  $(t)$

$$\mathcal{Z}_t = \mathcal{Z}/(t) = \mathbb{Z}[Y]/(Y^n - Y + d, Y - d).$$

In this quotient, polynomials congruent modulo the ideal  $(t)$  represent the same message. Then, for an integer  $0 \leq m < d^n$  with radix- $d$  representation  $m = \sum_{i=0}^{n-1} d_i d^i$ , where  $d_i \in [0, d-1]$ , its base- $d$  flattened polynomial is as follows

$$[m] = \sum_{i=0}^{n-1} d_i Y^i \in \mathbb{Z}[Y].$$

**Theorem 1 (Ring isomorphism).** *Let  $\mathbb{Z}_{d^n} = \mathbb{Z}/d^n\mathbb{Z}$  denote the ring of integers modulo  $d^n$ . The rings  $\mathbb{Z}_{d^n}$  and  $\mathcal{Z}_t$  are isomorphic and the isomorphism is given by*

$$\mathbb{Z}_{d^n} \cong \mathcal{Z}_t, \quad m \longleftrightarrow [m].$$

*Proof.* In the quotient by  $t = Y - d$ , we have  $Y = d$ . Substituting  $Y = d$  into  $F(Y) = Y^n - Y + d$  gives  $F(d) = d^n$ . Therefore, evaluation at  $Y = d$  has

$$\mathcal{Z}_t = \mathbb{Z}[Y]/(F, Y - d) \cong \mathbb{Z}/(F(d)) = \mathbb{Z}/d^n\mathbb{Z} = \mathbb{Z}_{d^n},$$

Furthermore, we have  $[m](d) = \sum_{i=0}^{n-1} d_i d^i = m$ , it maps  $[m]$  to  $m$  and its inverse therefore maps each message  $m \in \mathbb{Z}_{d^n}$  to  $[m]$  in  $\mathcal{Z}_t$ , as claimed.  $\square$

By Theorem 1, addition and multiplication in  $\mathcal{Z}_t$  correspond to integer addition and multiplication modulo  $d^n$ . We now consider these operations in  $\mathcal{Z}$ , where computations are reduced modulo  $F$  but not modulo  $t$ . For messages  $m_1, m_2 \in \mathbb{Z}_{d^n}$ , denote  $m_{\text{add}} = (m_1 + m_2) \bmod d^n$ , and  $m_{\text{mult}} = (m_1 m_2) \bmod d^n$ . There exist  $C_{\text{add}}, C_{\text{mult}} \in \mathcal{Z}$  such that

$$[m_1] + [m_2] = [m_{\text{add}}] + tC_{\text{add}}, \quad [m_1][m_2] = [m_{\text{mult}}] + tC_{\text{mult}}. \quad (2)$$

To recover the canonical representation of the result  $[m_{\text{add}}]$  and  $[m_{\text{mult}}]$ , we need to remove the extra terms  $tC_{\text{add}}$  and  $tC_{\text{mult}}$ , which become zero after reduction modulo  $t$ .

In the binary case  $d = 2$ , [23] handles the additional terms  $tC$  by dividing  $t$  in the arithmetic ring  $\mathcal{Z}_{\mathbb{R}} = \mathbb{R}[Y]/(F)$ . This transforms  $[m] + tC$  into  $[m]/t + C$ . We extend this approach to arbitrary integer bases  $d \geq 2$  and define the corresponding encoding below. Since  $F(d) = d^n \neq 0$ , the element  $t = Y - d$  is invertible in  $\mathcal{Z}_{\mathbb{R}}$ .

**Definition 1 (Triangle encoding).** *For a message  $m \in \mathbb{Z}_{d^n}$ , the triangle encoding of  $m$  is*

$$\text{TriEnc}(m) = \frac{\Delta}{t}[m] = \Delta \left( \frac{m}{d^n} - \sum_{i=0}^{n-1} m'_i Y^i \right) \in \mathcal{Z}_{\mathbb{R}}.$$

where  $m_{\leq i} = m \bmod d^{i+1} = \sum_{j=0}^i d_j d^j$ ,  $m'_i = \frac{m_{\leq i}}{d^{i+1}}$ , and  $\Delta$  is the scale factor. We call  $m'_i$  the scaled residue modulo  $d^{i+1}$ . During arithmetic evaluation, we use triangle encodings of the general form

$$f = \Delta I + \frac{\Delta}{t}[m] + e, \quad (3)$$

where  $m \in \mathbb{Z}_{d^n}$  is the encoded message,  $I \in \mathcal{Z}$  is an integer polynomial, and  $e \in \mathcal{Z}_{\mathbb{R}}$  is the approximation error. Then, we define the decoding for the general-base triangle encoding.

**Definition 2 (Triangle decoding).** For  $f$  satisfying eq. (3), define its triangle decoding as

$$\text{TriDec}(f) = \left( \left\lfloor \frac{tf}{\Delta} \right\rfloor \right) (d) \bmod d^n,$$

We first round the coefficients of  $\frac{tf}{\Delta}$  to the nearest integers and then evaluate the resulting polynomial at  $Y = d$ . This order is important, since evaluating at  $Y = d$  before rounding would amplify the coefficient-wise approximation errors by powers of  $d$ . Finally, the resulting integer is reduced modulo  $d^n$  to obtain the decoded message in  $\mathbb{Z}_{d^n}$ .

**Theorem 2 (Triangle decoding correctness).** Let  $f$  be the general triangle encoding of a message  $m \in \mathbb{Z}_{d^n}$ . Suppose that the approximation error  $e$  satisfies  $\|te\|_{\infty} < \frac{\Delta}{2}$ , then  $\text{TriDec}(f) = m$ .

*Proof.* Multiplication by  $t$  and dividing by  $\Delta$  gives  $\frac{tf}{\Delta} = tI + [m] + \frac{te}{\Delta}$ . Since  $\left\| \frac{te}{\Delta} \right\|_{\infty} < \frac{1}{2}$  and  $tI + [m]$  has integer coefficients, coefficientwise rounding yields  $\left\lfloor \frac{tf}{\Delta} \right\rfloor = tI + [m]$ . Moreover, evaluation at  $Y = d$  the term  $tI$  is zero modulo  $F(d) = d^n$ . Hence,  $(tI + [m])(d) \equiv m \pmod{d^n}$ , and the final reduction modulo  $d^n$  returns  $m$ .  $\square$

The following theorem shows that arithmetic on triangle encodings in  $\mathcal{Z}_{\mathbb{R}}$  preserves arithmetic operations  $\mathbb{Z}_{d^n}$ , as described below.

**Theorem 3 (Preservation of arithmetic operations).** Let  $f^{(1)} = \Delta I^{(1)} + \frac{\Delta}{t}[m_1] + e^{(1)}$  and  $f^{(2)} = \Delta I^{(2)} + \frac{\Delta}{t}[m_2] + e^{(2)}$  be triangle encodings of  $m_1, m_2 \in \mathbb{Z}_{d^n}$ , respectively.

(i) For triangle encodings, addition is defined by

$$f_{\text{add}} = f^{(1)} + f^{(2)} = \Delta I_{\text{add}} + \frac{\Delta}{t}[m_{\text{add}}] + e_{\text{add}},$$

where  $I_{\text{add}} = I^{(1)} + I^{(2)} + C_{\text{add}}, e_{\text{add}} = e^{(1)} + e^{(2)}$ .

(ii) For triangle encodings, multiplication is defined by

$$f_{\text{mult}} = \frac{t}{\Delta} f^{(1)} f^{(2)} = \Delta I_{\text{mult}} + \frac{\Delta}{t}[m_{\text{mult}}] + e_{\text{mult}},$$

where

$$I_{\text{mult}} = tI^{(1)}I^{(2)} + I^{(1)}[m_2] + I^{(2)}[m_1] + C_{\text{mult}},$$

$$e_{\text{mult}} = (tI^{(1)} + [m_1])e^{(2)} + (tI^{(2)} + [m_2])e^{(1)} + \frac{t}{\Delta}e^{(1)}e^{(2)}.$$

*Proof.* The result follows immediately from the definitions of  $m_{\text{add}}$  and  $m_{\text{mult}}$  and eq. (2).  $\square$

In the multiplication operation, the factor  $\frac{t}{\Delta}$  is required to preserve the triangle form. In both addition and multiplication, the terms  $tC_{\text{add}}$  and  $tC_{\text{mult}}$  are incorporated into  $I_{\text{add}}$  and  $I_{\text{mult}}$ , respectively. Consequently, each output again satisfies eq. (3) and can be used in subsequent arithmetic operations.

*Remark 1 (Arbitrary plaintext moduli).* The algebraic encoding construction can be extended to arbitrary integer moduli. For any  $M \geq 2$ , choose a monic polynomial  $G \in \mathbb{Z}[Y]$  satisfying  $G(d) = M$ . Setting  $t = Y - d$  gives  $\mathbb{Z}[Y]/(G, t) \cong \mathbb{Z}_M$ . Since  $G(d) \neq 0$ , the element  $t$  is invertible in  $\mathbb{R}[Y]/(G)$ , allowing messages to be encoded as  $m/t$  in an algebraic form analogous to triangle encoding.

*Remark 2 (Focus on the plaintext modulus  $d^n$ ).* Although the construction can be defined directly for an arbitrary plaintext modulus  $M$ , we focus on the modulus  $d^n$  for two reasons. First, arithmetic modulo  $M$  can be realized within arithmetic modulo  $d^n$ . Specifically, we choose  $n$  sufficiently large so that all intermediate and final results of the corresponding integer computation lie in  $[0, d^n)$ . This allows the computation to use  $F(Y) = Y^n - Y + d$  and retain its simple triangular encoding structure. To recover the result modulo  $M$ , we first apply the triangle decoding `TriDec`, which returns a value modulo  $d^n$ , and we further reduce the result modulo  $M$ . Second, when  $d$  is a power of two, arithmetic modulo  $d^n = 2^\ell$ , with  $\ell = n \log_2 d$ , matches the modular arithmetic on machine words in modern processors.

### 3.2 Embedding into CKKS

The preceding subsection introduced the triangle encoding, which encodes the elements of  $\mathbb{Z}_{d^n}$  to representations in the arithmetic ring  $\mathcal{Z}_{\mathbb{R}}$ . In this subsection, we further embed the triangle encoding into complex vectors while preserving arithmetic operation and present the homomorphic realization in the CKKS scheme. We first define the canonical embedding by evaluating each polynomial at roots of  $F$ . For even  $n$ , the polynomial  $F$  has  $n$  distinct complex roots, denoted  $\xi_0, \dots, \xi_{n/2-1}, \bar{\xi}_0, \dots, \bar{\xi}_{n/2-1}$ . Formally, we have the following proposition.

**Proposition 1 (Canonical embedding of  $\mathcal{Z}_{\mathbb{R}}$ ).** *Define the canonical embedding of  $\mathcal{Z}_{\mathbb{R}}$  by*

$$\sigma_{\mathcal{Z}} : \mathcal{Z}_{\mathbb{R}} \longrightarrow \mathbb{C}^{n/2}, f \longmapsto (f(\xi_0), \dots, f(\xi_{n/2-1})).$$

*Then  $\sigma_{\mathcal{Z}}$  preserves addition and multiplication. For all  $f, g \in \mathcal{Z}_{\mathbb{R}}$ , we have*

$$\sigma_{\mathcal{Z}}(f + g) = \sigma_{\mathcal{Z}}(f) \boxplus \sigma_{\mathcal{Z}}(g), \quad \sigma_{\mathcal{Z}}(fg) = \sigma_{\mathcal{Z}}(f) \odot \sigma_{\mathcal{Z}}(g),$$

*where  $\boxplus$  and  $\odot$  denote componentwise addition and multiplication, respectively.*

The triangle encoding together with the canonical embedding  $\sigma_{\mathcal{Z}}$  represents each message  $\in \mathbb{Z}_{d^n}$  using  $n/2$  complex slots. Since a CKKS ciphertext offers  $N/2$  slots, we can therefore pack  $N/n$  words into a single ciphertext. Given a message vector  $\mathbf{m} = (m_0, \dots, m_{N/n-1}) \in (\mathbb{Z}_{d^n})^{N/n}$ , we first apply the triangle encoding  $\text{TriEnc}$  and then the canonical embedding  $\sigma_{\mathcal{Z}}$  to each message. The resulting packed vector is

$$(\sigma_{\mathcal{Z}}(\text{TriEnc}(m_0)), \dots, \sigma_{\mathcal{Z}}(\text{TriEnc}(m_{N/n-1}))) \in \mathbb{C}^{N/2}. \quad (4)$$

The packed vector can then be further applying the inverse CKKS canonical embedding via  $\sigma_{\mathcal{R}}$  to a CKKS plaintext polynomial, and we define

$$\text{Enc}_{\mathcal{Z}_{\mathbb{R}} \rightarrow \mathcal{R}}(f_0, \dots, f_{N/n-1}) = \lfloor \sigma_{\mathcal{R}}^{-1}(\sigma_{\mathcal{Z}}(f_0), \dots, \sigma_{\mathcal{Z}}(f_{N/n-1})) \rfloor \in \mathcal{R},$$

where  $f_i = \text{TriEnc}(m_i) \in \mathcal{Z}_{\mathbb{R}}$  is the triangle encoding of message  $m_i$ . Conversely, for a plaintext polynomial  $\text{pt} \in \mathcal{R}$ , we define

$$\text{Dec}_{\mathcal{R} \rightarrow \mathcal{Z}_{\mathbb{R}}}(\text{pt}) = \left( \sigma_{\mathcal{Z}}^{-1}((\sigma_{\mathcal{R}}(\text{pt}))_{0:n/2-1}), \dots, \sigma_{\mathcal{Z}}^{-1}((\sigma_{\mathcal{R}}(\text{pt}))_{N/2-n/2:N/2-1}) \right) \in \mathcal{Z}_{\mathbb{R}}^{N/n}.$$

The encoding map  $\text{Enc}_{\mathcal{Z}_{\mathbb{R}} \rightarrow \mathcal{R}}$  applies coefficientwise rounding in  $\mathcal{R}$ . Consequently, when recovering the triangle representation from a CKKS plaintext, the rounding error may be amplified by the two linear transform  $\sigma_{\mathcal{R}}$  and  $\sigma_{\mathcal{Z}}^{-1}$ . We analyze the resulting error  $e_{\text{enc}}$  in the recovered triangle representation and derive a sufficient condition under which triangle decoding still recovers the original message in  $\mathbb{Z}_{d^n}$  exactly.

**Theorem 4 (Correctness of  $\text{Dec}_{\mathcal{R} \rightarrow \mathcal{Z}_{\mathbb{R}}}$ ).** *For  $f_0, \dots, f_{N/n-1} \in \mathcal{Z}_{\mathbb{R}}$ , and  $\text{pt} = \text{Enc}_{\mathcal{Z}_{\mathbb{R}} \rightarrow \mathcal{R}}(f_0, \dots, f_{N/n-1})$ . Every triangle representation recovered by  $\text{Dec}_{\mathcal{R} \rightarrow \mathcal{Z}_{\mathbb{R}}}(\text{pt})$  has the form  $f + e_{\text{enc}}$ , then  $\|e_{\text{enc}}\|_{\infty} < N$ .*

*Proof.* Coefficientwise rounding gives  $\|\text{pt} - \sigma_{\mathcal{R}}^{-1}(\sigma_{\mathcal{Z}}(f_0), \dots, \sigma_{\mathcal{Z}}(f_{N/n-1}))\|_{\infty} \leq \frac{1}{2}$ . By linearity of  $\sigma_{\mathcal{R}}$ , plaintext decoding yields a slot vector satisfying  $\|\sigma_{\mathcal{R}}(\text{pt}) - (\sigma_{\mathcal{Z}}(f_0), \dots, \sigma_{\mathcal{Z}}(f_{N/n-1}))\|_{\infty} \leq \frac{1}{2} \|\sigma_{\mathcal{R}}\|$ . Therefore applying  $\sigma_{\mathcal{Z}}^{-1}$  gives  $\|e_{\text{enc}}\|_{\infty} \leq \frac{1}{2} \|\sigma_{\mathcal{Z}}^{-1}\| \|\sigma_{\mathcal{R}}\|$ .

We have  $\|\sigma_{\mathcal{Z}}^{-1}\| < 2$  and  $\|\sigma_{\mathcal{R}}\| \leq N$ , with both bounds proved in Appendix A. Combining these bounds yields  $\|e_{\text{enc}}\|_{\infty} < N$ .  $\square$

By Theorem 2, a recovered triangle representation can be decoded correctly whenever its error satisfies  $\|te_{\text{enc}}\|_{\infty} < \frac{\Delta}{2}$ . Multiplication by  $t = Y - d$  in  $\mathcal{Z}_{\mathbb{R}}$  satisfies  $\|te_{\text{enc}}\|_{\infty} \leq 2d \|e_{\text{enc}}\|_{\infty} < 2dN$ . Therefore, under the condition  $\Delta > 4dN$  we can recover all the original messages exactly. Table 5 provides a numerical experiments of the decoding error bounds across different encoding bases.

**Proposition 2 (Addition in  $\mathcal{R}$  is addition in  $\mathcal{Z}_{\mathbb{R}}$ ).** *Let  $\text{ct}^{(0)}$  and  $\text{ct}^{(1)}$  be two ciphertexts whose decrypted plaintexts encode  $f^{(0)}, f^{(1)} \in \mathcal{Z}_{\mathbb{R}}$ , respectively. Then the CKKS addition  $\text{ct}_{\text{add}} = \text{CKKS.Add}(\text{ct}^{(0)}, \text{ct}^{(1)})$  decrypts to a plaintext encoding  $f^{(0)} + f^{(1)}$  in  $\mathcal{Z}_{\mathbb{R}}$ , up to the accumulated CKKS approximation error.*

**Proposition 3 (Multiplication in  $\mathcal{R}$  is multiplication in  $\mathcal{Z}_{\mathbb{R}}$ ).** *Let  $\text{ct}^{(0)}$  and  $\text{ct}^{(1)}$  be two ciphertexts whose decrypted plaintexts encode  $f^{(0)}, f^{(1)} \in \mathcal{Z}_{\mathbb{R}}$ , respectively. Then the CKKS multiplication  $\text{ct}_{\text{mult}} = \text{CKKS.Mult}(\text{ct}^{(0)}, \text{ct}^{(1)})$  decrypts to a plaintext encoding  $f^{(0)}f^{(1)}$  in  $\mathcal{Z}_{\mathbb{R}}$ .*

**Proposition 4 (Rescaling in  $\mathcal{R}$  is rescaling in  $\mathcal{Z}_{\mathbb{R}}$ ).** *Let  $\text{ct}$  be a ciphertext at scale  $\Delta'$  whose decrypted plaintext encodes  $f \in \mathcal{Z}_{\mathbb{R}}$ . The CKKS rescale  $\text{ct}_{\text{rs}} = \text{CKKS.Rescale}_q(\text{ct})$  decrypts to a plaintext encoding  $\frac{f}{q}$  in  $\mathcal{Z}_{\mathbb{R}}$  up to the accumulated CKKS approximation error and has output scale  $\Delta = \frac{\Delta'}{q}$ .*

The preceding propositions describe the native RLWE operations in  $\mathcal{Z}_{\mathbb{R}}$ . However, the product  $f^{(0)}f^{(1)}$  obtained from RLWE multiplication does not have the triangle form. To restore this form, we multiply by the plaintext  $\Delta t$  and perform two rescalings by  $\Delta$ . Accordingly, multiplication of triangle encodings is implemented homomorphically as

$$f_{\text{mult}} = \frac{1}{\Delta^2}(\Delta t)f^{(0)}f^{(1)} = \frac{t}{\Delta}f^{(0)}f^{(1)} = \Delta I_{\text{mult}} + \frac{\Delta}{t}[m_{\text{mult}}] + e_{\text{mult}}.$$

The two rescaling operations consume two levels of the ciphertext modulus chain.

**Proposition 5 (Rotation of Messages).** *Let  $\mathbf{m} = (m_0, \dots, m_{N/n-1}) \in (\mathbb{Z}_{d^n})^{N/n}$ , and let  $\text{ct}$  be a ciphertext whose decrypted plaintext encodes the triangle representations  $\text{TriEnc}(m_0), \dots, \text{TriEnc}(m_{N/n-1})$ , each occupying  $n/2$  consecutive CKKS slots. Then, for  $0 \leq s < N/n$ , the CKKS rotation  $\text{ct}_{\text{rot}} = \text{CKKS.Rotate}(\text{ct}, s(n/2))$  decrypts to a plaintext encoding the triangle representations of message vector  $(m_s, \dots, m_{N/n-1}, m_0, \dots, m_{s-1})$ .*

A cyclic rotation within every message can be implemented using two RLWE rotations together with complementary plaintext masks.

### 3.3 Linear Transformations

This section shows the linear transformations between the  $\mathcal{R}_{\mathbb{R}}$  and  $\mathcal{Z}_{\mathbb{R}}$  representations for general-base triangle encoding.

*$\mathcal{Z}$ -to- $\mathbb{C}$  and  $\mathbb{C}$ -to- $\mathcal{Z}$ .* The  $\mathcal{Z}$ -to- $\mathbb{C}$  transformation takes a CKKS ciphertext whose slots encode the canonical embeddings of the triangle encodings and outputs two ciphertexts whose slots encode the corresponding triangle coefficients. For the standard transformation,  $\mathbb{C}$ -to- $\mathcal{Z}$  performs the reverse conversion. For  $0 \leq j < N/n$ , write  $f^{(j)} = \sum_{i=0}^{n-1} f_i^{(j)} Y^i \in \mathcal{Z}_{\mathbb{R}}$  and define

$$\mathbf{f}^{(j)} = (f_0^{(j)}, \dots, f_{n-1}^{(j)}), \quad \mathbf{f}_{\text{pack}} = (\mathbf{f}^{(0)}, \dots, \mathbf{f}^{(N/n-1)}) \in \mathbb{R}^N.$$

Let  $V \in \mathbb{C}^{n \times (n/2)}$  have entries  $V_{i,s} = \xi_s^i$ , and set  $V_{N/n} = I_{N/n} \otimes V \in \mathbb{C}^{N \times (N/2)}$ . Since  $F$  has  $n$  distinct roots,  $(V_{N/n}, \overline{V_{N/n}})$  is invertible. For the packed canonical embedding vector  $\mathbf{m} \in \mathbb{C}^{N/2}$  defined in eq. (4), we obtain

$$\mathbf{f}_{\text{pack}} = (\mathbf{m}, \overline{\mathbf{m}})(V_{N/n}, \overline{V_{N/n}})^{-1}.$$

Thus,  $\mathcal{Z}$ -to- $\mathbb{C}$  acts independently on each message and places the lower and upper coefficient halves in two output CKKS ciphertexts, respectively, with one coefficient per slot. Conversely,  $\mathbb{C}$ -to- $\mathcal{Z}$  combines the two coefficient halves corresponding to each message and reconstructs the packed canonical embeddings through  $\mathbf{m} = \mathbf{f}_{\text{pack}} V_{N/n}$ .

We will use a special- $d_0$  variant of  $\mathcal{Z}$ -to- $\mathbb{C}$  later. For  $f = \Delta I + \Delta[m]/t + e$ , the constant and highest-degree coefficients satisfy

$$f_0 = \Delta I_0 + \frac{\Delta m}{d^n} - \frac{\Delta d_0}{d} + e_0, \quad f_{n-1} = \Delta I_{n-1} - \frac{\Delta m}{d^n} + e_{n-1}.$$

Since the term  $m/d^n$  appears in these two coefficients with opposite signs, we eliminate it from the constant coefficient by replacing  $f_0$  with their sum,

$$f_0^{\text{sp}} = f_0 + f_{n-1} = \Delta(I_0 + I_{n-1}) - \frac{\Delta d_0}{d} + (e_0 + e_{n-1}).$$

The modified constant coefficient therefore represents  $-d_0/d$  up to integer overflow and approximation error, while the remaining coefficients are unchanged. We incorporate this modification directly into the  $\mathcal{Z}$ -to- $\mathbb{C}$  matrix. Under our row-vector convention, we replace the column  $jn$  of  $(V_{N/n}, \bar{V}_{N/n})^{-1}$  by the sum of its original columns  $jn$  and  $jn + n - 1$ , where  $j \in [0, N/n - 1]$ , while all other columns remain unchanged.

*$\mathcal{Z}$ -to- $\mathcal{R}$  and  $\mathcal{R}$ -to- $\mathcal{Z}$ .* The CKKS  $\mathbb{C}$ -to- $\mathcal{R}$  transformation places the two real slot vectors in the lower and upper halves of the normalized plaintext coefficient vector. Composing it with  $\mathcal{Z}$ -to- $\mathbb{C}$  gives  $\mathcal{Z}$ -to- $\mathcal{R} = (\mathbb{C}\text{-to-}\mathcal{R}) \circ \mathcal{Z}\text{-to-}\mathbb{C}$ . For A2B, we define the special- $d_0$  variant of  $\mathcal{Z}$ -to- $\mathcal{R}$  by replacing  $\mathcal{Z}$ -to- $\mathbb{C}$  in this composition with its special- $d_0$  variant. The inverse of the standard  $\mathcal{Z}$ -to- $\mathcal{R}$  transformation is  $\mathcal{R}$ -to- $\mathcal{Z} = \mathbb{C}\text{-to-}\mathcal{Z} \circ (\mathcal{R}\text{-to-}\mathbb{C})$ , where  $\mathcal{R}$ -to- $\mathbb{C}$  recovers the two coefficient halves in CKKS slots.

## 4 A2B Conversion via Shared Overflow Cancellation

This section develops an A2B conversion for power-of-two digit bases, mapping the triangle encoding of a message  $m$  to its canonical bit encoding. We first derive shared overflow cancellation and select a window size that yields binary carries. We then jointly extract candidate bits and evaluate carry states using multi-value functional bootstrapping, followed by leveled carry recovery and bit correction without further bootstrapping. Finally, we present the homomorphic algorithm and analyze its correctness and complexity.

Conversely, B2A linearly combines each group of  $\log_2 d$  consecutive encrypted bits into a base- $d$  digit, forms the digit polynomial  $[m]$ , and maps it to the triangle encoding  $\Delta[m]/t$ , where  $t = Y - d$ , using only homomorphic linear operations and no bootstrapping.

#### 4.1 Warm-up

The special  $d_0$ - $\mathbb{Z}$ -to- $\mathbb{C}$  transform yields the triangle coefficient representation  $\mathbf{f} = (f_0, \dots, f_{n-1})$  in the slots form of CKKS. We first consider the ideal case  $I = 0$ . The  $i$ -th triangle coefficient then satisfies  $f_i = -m'_i$ , where  $m'_i = m \bmod d^{i+1}/d^{i+1}$  is the scaled residue modulo  $d^{i+1}$ . According the structure of triangle encoding, we have

$$dm'_i - m'_{i-1} = d_i, \quad 0 < i \leq n-1.$$

That means the base- $d$  digits satisfy

$$d_0 = -df_0, \quad d_i = f_{i-1} - df_i, \quad 0 < i \leq n-1.$$

Hence, when integer overflow is ignored, all base- $d$  digits can be recovered independently by scaling and subtracting triangle coefficients. The recovered digits give the base- $d$  radix encoding of  $m$ . Finally, a single multi-value functional bootstrap with  $\log_2 d$ -bit input precision can extract all bits of  $m$ .

However, homomorphic arithmetic introduces an integer overflow polynomial  $I$ . Then, the  $i$ -th triangle coefficient satisfies  $f_i = I_i - m'_i$ , where  $I_i \in \mathbb{Z}$ . Define the scaled difference between the  $i$ -th and  $(i+1)$ -th triangle coefficients is

$$s_0 = -df_0 = dm'_0 - dI_0, \quad s_i = f_{i-1} - df_i = d_i + I_{i-1} - dI_i, \quad 0 < i \leq n-1.$$

The integer overflow  $I_{i-1}$  and  $I_i$  prevents the base- $d$  digit  $d_i$  from being recovered directly.

Consider two consecutive scaled differences  $s_{i-1}$  and  $s_i$ ,

$$s_{i-1} = d_{i-1} + I_{i-2} - dI_{i-1}, \quad s_i = d_i + I_{i-1} - dI_i.$$

Both differences contain the same integer overflow  $I_{i-1}$  with different scaling factors. Directly scaling and adding the two differences  $ds_i + s_{i-1}$  to cancel this shared term would also mix the digits  $d_{i-1}$  and  $d_i$ . The resulting value would therefore not recover the desired digit.

For each scaled difference, define

$$r_i = s_i \bmod d, \quad q_i = \frac{r_i - s_i}{d}, \quad c_i = \left\lfloor \frac{d_i + I_{i-1}}{d} \right\rfloor.$$

The quotient of  $s_{i-1}$  and the remainder of  $s_i$  satisfy

$$\begin{aligned} q_{i-1} &= I_{i-1} - c_{i-1}, \\ r_i &= d_i + I_{i-1} - dc_i. \end{aligned}$$

Now they contain the same integer overflow  $I_{i-1}$  with the same factor and only the remainder contain the digit. Subtracting them gives

$$r_i - q_{i-1} = d_i + c_{i-1} - dc_i.$$

This subtraction cancels the shared integer overflow, leaving a single base- $d$  digit together with two adjacent integer carry terms. Therefore, recovering the digits

requires determining the corresponding carries. Since  $0 \leq d_i < d$ , the carries can be recovered recursively as  $c_i = \left\lceil \frac{c_{i-1} - (r_i - q_{i-1})}{d} \right\rceil$ . Thus, computing  $c_i$  requires the value of  $c_{i-1}$ . With  $c_0 = 0$ , recovering all carries for an  $n$ -digit base- $d$  message therefore requires  $O(n)$  sequential functional bootstraps, giving a bootstrapping linear in the message precision.

## 4.2 Parallel Bit Carry Recovery

We eliminate the sequential dependency in Section 4.1 by choosing a window length  $\omega$  such that the reduction modulo  $D = d^\omega$  for each window produces a binary carry. We then recover all carries using parallel binary carry propagation.

The range of the integer carry  $c_i$  depends on the overflow magnitude  $\|I\|_\infty$ . Building on the arithmetic integer overflow refreshing method of GZ26 [23], we construct an overflow refreshing procedure for A2B, denoted by  $\text{Refresh}_I$  and summarized in Algorithm 2. Our adaptation incorporates the special- $d_0$   $\mathcal{Z}$ -to- $\mathcal{R}$  transformation before truncation. The transformed integer overflow is therefore removed by truncation, and subsequent modulus raising introduces fresh overflow controlled by sparse secret encapsulation [9]. The procedure outputs the preprocessed coefficients in CKKS slots. We retain the notation  $I$  for the refreshed overflow, whose magnitude can be bounded in terms of the ephemeral secret key distribution.

---

### Algorithm 2: Integer overflow refreshing for A2B ( $\text{Refresh}_I$ )

---

**Input:** An arithmetic-mode ciphertext  $\text{ct}_A = \text{Enc}_s(\Delta I + \frac{\Delta}{t}[m])$  at modulus  $Q'_\ell$ ; key-switching keys  $\text{ksk}_{s \rightarrow \tilde{s}}$  and  $\text{ksk}_{\tilde{s} \rightarrow s}$ .

**Output:** Two ciphertexts  $(\text{ct}_{f,0}, \text{ct}_{f,1})$  containing the lower and upper halves of each word's preprocessed coefficient vector in CKKS slots

```

1  $\text{ct}_R \leftarrow \text{special-}d_0 \text{ } \mathcal{Z}\text{-to-}\mathcal{R}(\text{ct}_A)$ ;
2  $\text{ct}_0 \leftarrow \text{CKKS.Truncate}_{Q'_\ell \rightarrow q'_0}(\text{ct}_R)$ ;
3  $\text{ct}_1 \leftarrow \text{KeySwitch}_{s \rightarrow \tilde{s}}(\text{ct}_0; \text{ksk}_{s \rightarrow \tilde{s}})$ ;
4  $\text{ct}_2 \leftarrow \text{CKKS.ModRaise}_{q'_0 \rightarrow Q'_L}(\text{ct}_1)$ ;
5  $\text{ct}_3 \leftarrow \text{KeySwitch}_{\tilde{s} \rightarrow s}(\text{ct}_2; \text{ksk}_{\tilde{s} \rightarrow s})$ ;
6  $(\text{ct}_{f,0}, \text{ct}_{f,1}) \leftarrow \mathcal{R}\text{-to-}\mathcal{C}(\text{ct}_3)$ ;
7 return  $(\text{ct}_{f,0}, \text{ct}_{f,1})$ ;

```

---

After apply  $\text{Refresh}_I$  to refresh the accumulated overflow and assume  $\|I\|_\infty < B/2$ , we add the public offset  $B/2$  to each coefficient of  $I$  and define  $I'_i = I_i + \frac{B}{2}$ ,  $f'_i = f_i + \frac{B}{2} = I'_i - m'_i$ ,  $0 \leq i < n$ . Since  $\|I\|_\infty < B/2$ , each shifted overflow coefficient satisfies  $I'_i \in (0, B)$ . Choose the smallest window length  $\omega$  such that  $D = d^\omega \geq B$ . First consider the case  $\omega \mid n$ , and let  $L = n/\omega$ . Grouping every  $\omega$  consecutive base- $d$  digits gives

$$m = \underbrace{(d_0 + \dots + d_{\omega-1}d^{\omega-1})}_{D_0} + \dots + d^{\omega(L-1)} \underbrace{(d_{(L-1)\omega} + \dots + d_{n-1}d^{\omega-1})}_{D_{L-1}},$$

where  $0 \leq D_k < D$ .

Following the construction in Section 4.1, replace the radix  $d$  with the window radix  $D$  and define

$$\begin{aligned} s_0 &= -Df'_{i_0}, & s_k &= f'_{i_{k-1}} - Df'_{i_k}, & 1 \leq k < L, \\ r_k &= s_k \bmod D, & q_k &= \frac{r_k - s_k}{D}, & 0 \leq k < L, \\ c_0 &= 0, & c_k &= \left\lfloor \frac{D_k + I'_{i_{k-1}}}{D} \right\rfloor, & 1 \leq k < L, \end{aligned}$$

where  $i_k = (k+1)\omega - 1$ . The same shared overflow cancellation gives  $r_0 = D_0$  and

$$v_k = r_k - q_{k-1} = D_k + c_{k-1} - Dc_k, \quad 1 \leq k < L. \quad (5)$$

**Theorem 5 (Binary carry propagation).** *Suppose  $0 < I'_i < B \leq D$  for all  $0 \leq i < n$ . Then  $c_k \in \{0, 1\}$  for every  $0 \leq k < L$ . Moreover,*

$$c_k = \begin{cases} 0, & v_k > 0, \\ 1, & v_k < 0, \\ c_{k-1}, & v_k = 0, \end{cases} \quad 1 \leq k < L. \quad (6)$$

*Proof.* For  $1 \leq k < L$ , the bound  $0 \leq D_k + I'_{i_{k-1}} < D + B \leq 2D$ , together with  $c_0 = 0$ , gives  $c_k \in \{0, 1\}$  for all  $0 \leq k < L$ .

By definition,  $v_k = r_k - q_{k-1}$ . Since  $r_k \in [0, D-1]$ ,  $q_{k-1} = I'_{i_{k-1}} - c_{k-1} \in [0, D-1]$ , we obtain  $-(D-1) \leq v_k \leq D-1$ .

By eq. (5),  $v_k = D_k + c_{k-1} - Dc_k$ , where  $D_k \in [0, D-1]$  and  $c_{k-1}, c_k \in \{0, 1\}$ . Since  $0 \leq D_k + c_{k-1} \leq D$  and  $Dc_k \in \{0, D\}$ ,  $v_k < 0$  implies  $c_k = 1$ , whereas  $v_k > 0$  implies  $c_k = 0$ . When  $v_k = 0$ , the identity  $D_k = Dc_k - c_{k-1}$ , together with  $0 \leq D_k < D$ , gives  $c_k = c_{k-1}$ .  $\square$

Theorem 5 shows that choosing a window size  $\omega$  such that  $D = d^\omega \geq B$  ensures that each window produces a binary carry. More importantly, the sign of  $v_k$  determines the current carry propagation behavior without requiring the incoming carry  $c_{k-1}$  to be known. Thus, the propagation behavior of all windows can be determined in parallel.

We distinguish three carry behaviors and encode them by  $u_k \in \{\frac{1}{2}, 0, \frac{1}{2}\}$  as follows, where  $I = \sqrt{-1}$ :

- When  $v_k > 0$ , we assign  $u_k = \frac{1}{2}$ . The outgoing carry  $c_k$  is 0, regardless of whether the incoming carry  $c_{k-1}$  is 0 or 1.
- When  $v_k < 0$ , we assign  $u_k = \frac{1}{2}$ . The outgoing carry  $c_k$  is 1, regardless of whether the incoming carry  $c_{k-1}$  is 0 or 1.
- When  $v_k = 0$ , we assign  $u_k = 0$ . The outgoing carry equals the incoming carry:  $c_k = 0$  when  $c_{k-1} = 0$ , and  $c_k = 1$  when  $c_{k-1} = 1$ . Thus, this behavior specifies how the window handles an incoming carry without determining its value.

All encoded carry behaviors  $u_k$  are computed in parallel using functional bootstrapping.

We define a bivariate interpolation function that takes the current carry behavior  $u_k$  and the incoming carry  $c_{k-1}$  and returns the outgoing carry  $c_k$ :

$$c_k = \begin{cases} 1, & u_k = \frac{1}{2}, \\ c_{k-1}, & u_k = 0, \\ 0, & u_k = \frac{1}{2}. \end{cases}$$

This function can be expressed as  $c_k = \chi(u_k) + \pi(u_k)c_{k-1}$ , where  $\pi(u_k)$  equals 1 when  $u_k = 0$  and 0 otherwise, while  $\chi(u_k)$  equals 1 when  $u_k = \frac{1}{2}$  and 0 otherwise. In particular,  $\pi(u_k) = 1 - 2(\text{Re}(u_k) + \text{Im}(u_k)) = 1 - (u_k + \overline{u_k}) + \text{I}(u_k - \overline{u_k})$ ,  $\chi(u_k) = u_k + \overline{u_k}$ . Therefore Both  $\pi(u_k)$  and  $\chi(u_k)$  can be computed without ciphertext multiplication and the evaluation of  $c_k$  requires one ciphertext multiplication.

Directly following above function, computing current carry requires previous carry, resulting in a sequential computation with  $O(L)$  multiplicative depth. We reduce this depth to  $O(\log L)$  by first combining the carry behaviors in parallel and then recovering all carries from the combined states.

Let  $u_{a:b}$  denote the carry behavior for computing the outgoing carry  $c_b$  from the incoming carry  $c_a$  across windows  $a + 1, \dots, b$ . In particular,  $u_{k-1:k} = u_k$ . For two adjacent blocks, we compute their combined state as

$$u_{a:k} = u_{b:k} + \pi(u_{b:k})(u_{a:b} - u_{b:k}), \quad a < b < k.$$

When  $u_{b:k} = 0$ , the current block propagates its incoming carry, so the combined state is  $u_{a:b}$ . Otherwise, the current block alone determines the outgoing carry, and the combined state is  $u_{b:k}$ .

Starting from the individual window states, we first combine adjacent states and then repeatedly double the step size. After  $\lceil \log_2 L \rceil$  iterations, we obtain the combined carry behavior  $u_{0:k}$  of windows  $1, \dots, k$  for all  $1 \leq k < L$ . Since  $c_0 = 0$ , all carries are then recovered in parallel as

$$c_k = \chi(u_{0:k}) = u_{0:k} + \overline{u_{0:k}}, \quad 1 \leq k < L.$$

Figure 3 illustrates the parallel composition of carry behaviors. The corresponding homomorphic implementation is presented in Algorithm 3.

*General Case.* In the general case, where  $\omega \nmid n$ , only the initial window requires special handling, all remaining windows are processed as  $\omega|n$  case. Choose  $\omega_0 \in [1, 2\omega) \setminus \{\omega\}$  and  $L$  such that  $n = \omega_0 + (L - 1)\omega$ . Set  $D_{\text{init}} = d^{\omega_0}$  and  $D_0 = \sum_{j=0}^{\omega_0-1} d_j d^j$ ,  $i_k = \omega_0 - 1 + k\omega$ ,  $k \in [0, L)$ .

If  $\omega_0 < \omega$ , the initial value is scaled to radix  $D$ :  $(-Df'_{i_0}) \bmod D = \frac{D}{D_{\text{init}}} D_0$ . Thus,  $D_0$  is multiplied by  $D/D_{\text{init}}$ , while the associated quotient remains  $I'_{i_0}$ .

If  $\omega_0 > \omega$ , the initial reduction instead uses  $D_{\text{init}}$ :  $(-D_{\text{init}}f'_{i_0}) \bmod D_{\text{init}} = D_0$ . In this case, both the modular reduction and the quotient division use  $D_{\text{init}}$ , and the associated quotient is again  $I'_{i_0}$ .

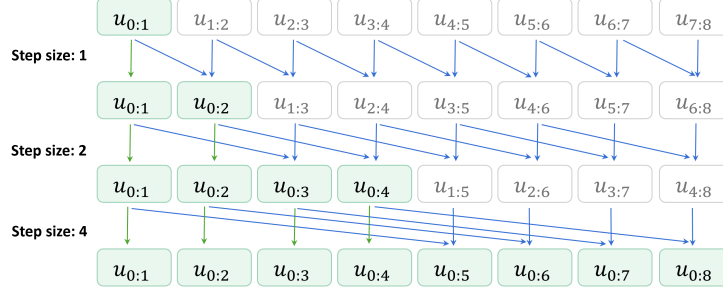


**Fig. 3.** Parallel combination of carry behaviors for eight windows example. The step sizes are 1, 2, and 4, respectively. Green boxes represent  $u_{0:k}$  and gray boxes represent  $u_{a:b}$  with  $a > 0$ . Green arrows indicate states retained unchanged, while blue arrows show the two inputs to each combination.

### 4.3 Bit Extraction

With all carries recovered in Section 4.2, subtracting the incoming carry from eq. (5) gives  $v_k - c_{k-1} = D_k - Dc_k$ , which recovers the radix digit  $D_k$  modulo  $D$ . A further multi-value functional bootstrapping can extract all bits of these digits. We avoid this third bootstrapping by extracting the bits of  $v_k \bmod D$  during the bootstrapping that computes the carry behaviors  $u_k$ . Once the carries are available, we correct these bits using leveled homomorphic operations.

The key insight is that the bits of  $v_k \bmod D$  can be extracted before the incoming carry is known. Let  $w = \omega \log_2 d$ , so that  $D = 2^w$ , and define

$$v_{k,t} = \text{bits}_t(v_k \bmod D), \quad 0 \leq t < w,$$

Since each  $v_{k,t}$  depends only on  $v_k$ , we can compute these bits jointly with the carry state  $u_k$  using multi-value bootstrapping. However,  $v_{k,t}$ 's are not yet the desired bits, as  $v_k \bmod D$  contains contributions from both  $D_k$  and  $c_{k-1}$ . It remains to correct the extracted bits  $v_{k,t}$  using the incoming carry  $c_{k-1}$ .

Reducing eq. (5) modulo  $D$  gives

$$v_k \bmod D = (D_k + c_{k-1}) \bmod D, \quad D_k = ((v_k \bmod D) - c_{k-1}) \bmod D.$$

Once  $c_{k-1}$  is known, we recover  $D_k$  by subtracting  $c_{k-1}$  from  $v_k \bmod D$ . Since  $c_{k-1} \in \{0, 1\}$ , this subtraction admits an efficient bitwise implementation.

Specifically, when  $c_{k-1} = 0$ , the desired bits are equal to the previously extracted bits, i.e.,  $d_{k,t} = v_{k,t}$ . When  $c_{k-1} = 1$ ,  $D_k = ((v_k \bmod D) - 1) \bmod D$ , and this subtraction can be implemented bitwise. If  $v_{k,0} = \dots = v_{k,t-1} = 0$  and  $v_{k,t} = 1$ , it generates borrows from the least significant bit through the  $(t-1)$ -th bit, and borrowing stops at the  $t$ -th bit. Consequently, the least significant  $t$  bits change from zero to one,  $d_{k,j} = 1 - v_{k,j}$  for  $0 \leq j < t$ . The  $t$ -th bit changes from one to zero,  $d_{k,t} = 1 - v_{k,t}$ . The remaining bits stay unchanged,  $d_{k,j} = v_{k,j}$  for  $t < j < w$ .

---

**Algorithm 3:** Parallel Carry Propagation

---

**Input:** A CKKS ciphertext  $\text{ct}_u$  encrypting  $u_k \in \{\frac{1}{2}, 0, \frac{1}{2}\}$  at positions  $i_k$  for  $1 \leq k < L$ , and zero elsewhere;  
plaintext masks  $\{\mu_s\}_{1 \leq s < L}$  and  $\mu_{\text{out}}$ , where  $(\mu_s)_{i_k} = 1$  for  $s < k < L$ , and  $(\mu_{\text{out}})_{i_k} = 1$  for  $1 \leq k < L$ , with all other entries equal to zero  
**Output:** A CKKS ciphertext  $\text{ct}_c$  encrypting  $c_k$  at positions  $i_k$  for  $1 \leq k < L$ , and zero elsewhere

```

1  $s \leftarrow 1$ ;
2 while  $s < L$  do
3    $\text{ct}_x \leftarrow \mu_s \odot \text{CKKS.Rotate}(\text{ct}_u, s\omega)$ 
4    $\text{ct}_\pi \leftarrow 1 + (L - 1)\text{ct}_u - (L + 1)\text{CKKS.Conjugate}(\text{ct}_u)$ 
5    $\text{ct}_t \leftarrow \text{CKKS.Mult}(\text{ct}_\pi, \text{ct}_x - \text{ct}_u)$ ;
6    $\text{ct}_u \leftarrow \text{ct}_u + \text{ct}_t$ 
7    $s \leftarrow 2s$ 
8  $\text{ct}_c \leftarrow \mu_{\text{out}} \odot (\text{ct}_u + \text{CKKS.Conjugate}(\text{ct}_u))$ 
9 return  $\text{ct}_c$ 

```

---

We summarize the two cases by the following expression for  $c_{k-1} \in \{0, 1\}$ . Let  $\beta_{k,t}$  denote the borrow input to the  $t$ -th bit. For  $t \geq 1$ , this is the borrow generated by the  $(t-1)$ -th bit and for  $t = 0$ , we set  $\beta_{k,0} = c_{k-1}$ ,

$$\beta_{k,t} = c_{k-1} \prod_{j=0}^{t-1} (1 - v_{k,j}), \quad 0 \leq t < w, \quad (7)$$

Therefore, the  $t$ -th bit of  $D_k$  is

$$d_{k,t} = v_{k,t} + \beta_{k,t}(1 - 2v_{k,t}) = v_{k,t} + c_{k-1}(1 - 2v_{k,t}) \prod_{j=0}^{t-1} (1 - v_{k,j}). \quad (8)$$

This expression corrects the previously extracted bits according to the incoming carry. In particular, leveled additions and multiplications suffice to recover the canonical bits of  $D_k$  without an additional functional bootstrapping.

#### 4.4 Homomorphic A2B

We present the complete A2B conversion algorithm in Algorithm 4 for  $\omega \mid n$ . It first applies  $\text{Refresh}_I$  from Algorithm 2 to obtain the special- $d_0$  coefficient representation in CKKS slots with refreshed integer overflow, denoted as  $\text{ct}_f$ <sup>4</sup>.

The first functional bootstrapping computes the remainders  $r_k = s_k \bmod D$ . Although  $s_k$  may take values in a large input domain, the remainder function

---

<sup>4</sup> Throughout the algorithm,  $\text{ct}_x$  denotes the ciphertext pair  $(\text{ct}_{x,0}, \text{ct}_{x,1})$ . Slotwise operations are applied separately to the two ciphertexts. Rotations are defined over all  $n$  triangle coefficients of each message and are implemented using ciphertext rotations and plaintext masks.

$x \mapsto x \bmod D$  is  $D$ -periodic, so it can be evaluated using an interpolation polynomial of degree  $D - 1$ . After shared overflow cancellation forms  $v_k$ , the second stage jointly evaluates the carry behaviors  $u_k$  and the bit functions  $v_{k,t} = \text{bits}_t(v_k \bmod D)$  using multi-value functional bootstrapping. It is note that We obtain the complex carry state by evaluating two real-valued LUTs for its real and imaginary components and then combining their outputs linearly. These lookup tables use a common period of  $2D$ . Finally, we perform parallel carry propagation and use each resulting bit carry to correct the corresponding candidate bit as in eq. (8).

---

**Algorithm 4:** Homomorphic A2B

---

**Input:** An arithmetic ciphertext  $\text{ct}_A$ ;  
parameters  $d, n, B, \omega$  with  $d \geq 2$  a power of two,  
 $D = d^\omega \geq B$  and  $\omega \mid n$ ;  
evaluation keys, including those for  $\text{Refresh}_I$

**Output:** Native Boolean ciphertexts  $\{\text{ct}_r\}_{0 \leq r < 2w}$ , where  $w = \omega \log_2 d$

- 1  $L \leftarrow n/\omega$ ,  $w \leftarrow \omega \log_2 d$ ;
- 2  $\text{ct}_f \leftarrow \text{Refresh}_I(\text{ct}_A)$  ;      // Refresh the overflow and expose coefficients in slots (Algorithm 2)
- 3  $\text{ct}_{f'} \leftarrow \text{ct}_f + B/2$  ;      // Shift the refreshed overflow
- 4  $\text{ct}_s \leftarrow \mu_{\text{out}} \odot \text{CKKS.Rotate}(\text{ct}_{f'}, \omega) - D\text{ct}_{f'}$ ;
- 5  $\text{ct}_r \leftarrow \text{FBT}(\text{ct}_s; \text{LUT}_{\text{mod}D})$  ;      // First bootstrap: compute remainders (Algorithm 1)
- 6  $\text{ct}_q \leftarrow (\text{ct}_r - \text{ct}_s)/D$  ;      // Recover the quotients
- 7  $\text{ct}_v \leftarrow \text{ct}_r - \mu_{\text{out}} \odot \text{CKKS.Rotate}(\text{ct}_q, \omega)$  ;      // Cancel the shared overflow
- 8  $(\text{ct}_u, \{\text{ct}_{v,t}\}_{0 \leq t < w}) \leftarrow \text{MV-FBT}(\text{ct}_v; \text{LUT}_u, \{\text{LUT}_{\text{bits}_t}\}_{0 \leq t < w})$  ;      // Second bootstrap:  
jointly compute carry behaviors and bits (Algorithm 1)
- 9  $\text{ct}_u \leftarrow \mu_{\text{out}} \odot \text{ct}_u$  ;      // Retain the complete-window states
- 10  $\text{ct}_c \leftarrow \text{ParallelCarry}(\text{ct}_u)$  ;      // Recover all carries using Algorithm 3
- 11  $\text{ct}_c^{\text{in}} \leftarrow \mu_{\text{out}} \odot \text{CKKS.Rotate}(\text{ct}_c, \omega)$ ;
- 12 **for**  $t = 0, \dots, w - 1$  **do**
- 13      $\text{ct}_{d,t} \leftarrow \text{ct}_{v,t} + \text{ct}_c^{\text{in}} \cdot (1 - 2\text{ct}_{v,t}) \prod_{j=0}^{t-1} (1 - \text{ct}_{v,j})$  ;      // Correct the  $t$ -th bit
- 14  $\text{ct}_{bw+t} \leftarrow \text{ct}_{d,t,b}$     ( $b \in \{0, 1\}, 0 \leq t < w$ );
- 15 **return**  $\{\text{ct}_r\}_{0 \leq r < 2w}$

---

*Remark (Optional output reordering).* Algorithm 4 output groups bits by their positions within each window. Bitwise operations can be evaluated directly when corresponding operand bits at same slot positions. Optionally, we can reording consecutive bits of each word into fewer ciphertexts, reducing storage and the number of ciphertexts processed by subsequent bitwise operations. This reordering requires plaintext masks and rotations without additional functional bootstrapping, see Appendix C.

*Correctness.* We analyze error propagation through the two functional bootstrappings and the subsequent leveled carry recovery and bit correction computations. For the functional bootstrapping, we use first-order trigonometric Hermite interpolation and apply the explicit approximation error bound of Alexandru et al. [1].

**Lemma 1 (Functional bootstrapping correctness [1]).** *Let  $f : \mathbb{Z}_p \rightarrow \mathbb{R}$ , where  $p \geq 2$ , and  $R_f$  is its first-order trigonometric Hermite interpolant constructed in [1], satisfying  $R_f(m/p) = f(m)$ ,  $R'_f(m/p) = 0$  for all  $m \in \mathbb{Z}_p$ . Let  $\varepsilon_{\text{in}}$  denote the input noise, then the functional bootstrapping output noise  $e_{\text{FBT},f}$  satisfies*

$$|e_{\text{FBT},f}| \leq \frac{C_f}{\Delta} |\varepsilon_{\text{in}}|^2 + |e_{\text{eval},f}|,$$

where  $A_f = \max_{m \in \mathbb{Z}_p} |f(m)|$ ,  $C_f = \frac{2\pi^2}{3} A_f (p-1)^2 (2p-1)$ , and  $e_{\text{eval},f}$  is the remaining approximation and homomorphic evaluation errors.

**Theorem 6 (Error of leveled carry recovery).** *Let  $e_{\text{in}}$  is the error of carry state. After Algorithm 3, then the output carry error  $e_{\text{carry}}$  satisfies*

$$|e_{\text{carry}}| \leq \frac{3\Delta}{2\sqrt{2}} \left[ \left( 1 + \frac{4\sqrt{2}}{3\Delta} |e_{\text{in}}| \right)^{3^{\lceil \log_2 L \rceil}} - 1 \right] + |e_{\text{eval}}|,$$

where  $|e_{\text{eval}}|$  bounds the total additional homomorphic evaluation error at the carry outputs.

**Theorem 7 (Error of leveled bit correction).** *Suppose the extracted bits  $v_{k,t}$  and the incoming carries have errors at most  $|e_{\text{in}}|$  and  $|e_{\text{carry}}|$ , respectively. For a window of  $w$  bits, the output bit error  $e_{\text{bit}}$  after leveled bit correction satisfies*

$$|e_{\text{bit}}| \leq |e_{\text{in}}| + \Delta \left[ \left( 1 + \frac{|e_{\text{carry}}|}{\Delta} \right) \left( 1 + \frac{2|e_{\text{in}}|}{\Delta} \right)^w - 1 \right] + |e_{\text{eval}}|.$$

where  $e_{\text{eval}}$  is the aggregate homomorphic-evaluation error introduced by leveled bit correction, measured at its output relative to exact evaluation on the same approximate inputs.

For readability, we defer the proofs of Theorems 6 and 7 to Appendix B. Combining Lemma 1 and Theorems 6 and 7 gives a bound on the aggregate A2B output error  $e_{\text{final}}$ . If  $\|e_{\text{final}}\|_{\infty} < \Delta/2$ , Algorithm 4 is exactly correct and recovers all canonical bits.

*Complexity.* We analyze the complexity of each stage separately.

- **Refresh<sub>I</sub>.** The Refresh<sub>I</sub> procedure includes the two linear transformations special- $d_0$ - $\mathcal{Z}$ -to- $\mathcal{R}$  and  $\mathcal{R}$ -to- $\mathbb{C}$ , which require  $L_{\mathcal{Z}\text{-to-}\mathcal{R}}$  levels before modulus raising and  $L_{\mathcal{R}\text{-to-}\mathbb{C}}$  levels after modulus raising, respectively. It requires no ciphertext multiplication.
- **Functional bootstrapping.** Algorithm 1 describes the functional bootstrapping procedure, which applies SlotsToCoeffs, raises the modulus, applies CoeffsToSlots, and then evaluates an exponential approximation followed by the interpolation polynomial representing the LUT. Therefore, its complete level budget is  $L_{\text{FBT}} = L_{\text{S2C}} + L_{\text{C2S}} + L_{\text{exp}} + L_{\text{poly}}$ , where  $L_{\text{exp}}$  and  $L_{\text{poly}}$  denote the depths

of the exponential approximation and interpolation polynomial evaluation, respectively. A  $p$ -entry LUT is represented by a polynomial of degree at most  $p - 1$ , which requires approximately  $\sqrt{2p}$  ciphertext multiplications and depth  $L_{\text{poly}}(p) = \lceil \log_2 p \rceil$ , using PS method [31].

The first functional bootstrap computes  $r_k = s_k \bmod D$  using a  $D$ -entry LUT, requiring approximately  $\sqrt{2D}$  ciphertext multiplications and polynomial depth  $L_{\text{poly}}(D) = w$ . The intervening computations of  $q_k = (r_k - s_k)/D$  and  $v_k = r_k - q_{k-1}$  use rotations and linear operations, adding no ciphertext multiplication and depth. The second functional bootstrapping jointly evaluates the carry state LUT and candidate bits using a  $2D$ -entry LUT, requires approximately  $(w + 1)\sqrt{D}$  ciphertext multiplications and polynomial depth  $L_{\text{poly}}(2D) = w + 1$ .

- *Leveled carry propagation and bit correction.* Parallel carry propagation over the  $L$  windows requires  $\lceil \log_2 L \rceil$  iterations, each of which uses one rotation, ciphertext multiplication and one multiplicative level. Hence, recovering all carry bits requires multiplicative depth  $L_{\text{carry}} = \lceil \log_2 L \rceil$  and  $\lceil \log_2 L \rceil$  ciphertext multiplications. To correct the candidate bits using the bit carry, we first evaluate the borrow eq. (7) using balanced binary trees. Across all  $w$  bit positions, this requires  $\sum_{t=0}^{w-1} t = w(w - 1)/2$  ciphertext multiplications and multiplicative depth  $\lceil \log_2 w \rceil$ . Then the canonical bits are recovered using eq. (8), requiring another  $w$  ciphertext multiplications in one additional level.

## 5 Implementation and Performance

We implement our scheme builds on GZ26 [23] and uses the OpenFHE library [2]. We evaluate the A2B conversion under general-base triangle encoding with  $d \in \{2, 4, 32\}$  at message precisions of 64, 128, and 256 bits. All experiments use single-threaded execution on a server equipped with two Intel Xeon Platinum 8592V processors and 1.0 TiB of RAM, running Ubuntu 22.04.5 LTS. The code was compiled with GCC 11.4.0 in Release mode with Intel HEXL enabled. Our source code is publicly available.<sup>5</sup> For completeness, we also report the running times of arithmetic and Boolean operations, together with the corresponding refreshing procedure within each mode in Appendix D.

### 5.1 Parameter Selection

We use ring dimension  $N = 2^{16}$  and 50-bit scaling factors for all parameter sets. We sample the main secret key from the uniform ternary distribution and use an ephemeral sparse ternary key with Hamming weight  $h = 32$  during modulus raising. The latter ensures that  $\|I\|_\infty < 16$ , except with negligible probability. We sample the encryption error from a discrete Gaussian distribution with standard deviation  $\sigma_{\text{err}} = 3.2$ . We assess the security of the resulting RLWE parameters using the LWE Estimator.<sup>6</sup> For  $N = 2^{16}$  and these distributions, the estimated

<sup>5</sup> <https://anonymous.4open.science/r/fhe-simd-alu>

<sup>6</sup> <https://github.com/malb/lattice-estimator>

upper bound on  $\log_2(QP)$  at the 128-bit security level is approximately 1746 bits. As shown in Table 2, the largest value of  $\log_2(QP)$  among our parameter sets is 1740 bits, which remains below this bound. Therefore, all parameter sets used in our experiments meet the targeted 128-bit security level.

**Table 2.** Parameter sets for the A2B conversion.

| Message precision<br>$\ell$ (bits) | Arithmetic embedding<br>$\ell_{\text{emb}}$ (bits) | Encoding base<br>$\log_2 d$ (bits) | Packing capacity<br>$N/n$ | Window bit widths<br>$(\omega_0 \log_2 d, \omega \log_2 d)$ | Context depth<br>$L_{\text{ctx}}$ | Hybrid KS<br>digits | Modulus bits<br>$\log_2(QP)$ |
|------------------------------------|----------------------------------------------------|------------------------------------|---------------------------|-------------------------------------------------------------|-----------------------------------|---------------------|------------------------------|
| 64                                 | 64                                                 | 1                                  | 1024                      | (4, 5)                                                      | 26                                | 6                   | 1650                         |
|                                    | 64                                                 | 2                                  | 2048                      | (4, 6)                                                      | 27                                | 6                   | 1700                         |
|                                    | 80                                                 | 5                                  | 4096                      | (5, 5)                                                      | 26                                | 6                   | 1650                         |
| 128                                | 128                                                | 1                                  | 512                       | (3, 5)                                                      | 27                                | 7                   | 1640                         |
|                                    | 128                                                | 2                                  | 1024                      | (2, 6)                                                      | 28                                | 8                   | 1700                         |
|                                    | 160                                                | 5                                  | 2048                      | (5, 5)                                                      | 27                                | 7                   | 1640                         |
| 256                                | 256                                                | 1                                  | 256                       | (6, 5)                                                      | 28                                | 8                   | 1690                         |
|                                    | 256                                                | 2                                  | 512                       | (4, 6)                                                      | 29                                | 8                   | 1740                         |
|                                    | 320                                                | 5                                  | 1024                      | (5, 5)                                                      | 28                                | 8                   | 1690                         |

*Note.* For  $\log_2 d = 5$ , an  $\ell$ -bit word is represented in an  $\ell_{\text{emb}}$ -bit arithmetic embedding. The A2B conversion retains the least significant  $\ell$  bits as the Boolean output.

## 5.2 Performance and Comparison

We report the measured A2B latency, breakdown time and output error for all evaluated message precisions and encoding bases in Table 3.

Across all evaluated message precisions,  $d = 2$  achieves the lowest latency, ranging from 71.47 seconds at 64 bits to 104.02 seconds at 256 bits. In contrast,  $d = 32$  achieves the lowest amortized time, ranging from 29.84 ms per message at 64 bits to 138.51 ms per message at 256 bits. Although the larger encoding base increases the conversion latency, its higher packing capacity reduces the cost per message. Across all parameters, the maximum output error remains below  $2^{-8}$ .

**Table 3.** Single-thread A2B performance and output error at  $N = 2^{16}$ . Reported timings are averaged over five measurements after one warm-up run.

| Message precision<br>$\ell$ (bits) | Encoding base<br>$\log_2 d$ (bits) | Refresh <sub>I</sub><br>(s) | Stage 1<br>(s) | Stage 2<br>(s) | Carry recovery<br>(s) | Bit correction<br>(s) | Latency<br>(s) | Amortized time<br>(ms) | Output error<br>$-\log_2 e_{\text{max}}$ (bits) |
|------------------------------------|------------------------------------|-----------------------------|----------------|----------------|-----------------------|-----------------------|----------------|------------------------|-------------------------------------------------|
| 64                                 | 1                                  | 12.21                       | 28.54          | 28.37          | 1.07                  | 1.29                  | <b>71.47</b>   | 69.80                  | 11.79                                           |
|                                    | 2                                  | 11.45                       | 31.86          | 40.14          | 1.08                  | 1.76                  | 86.29          | 42.13                  | 10.54                                           |
|                                    | 5                                  | 10.87                       | 48.99          | 57.65          | 2.13                  | 2.60                  | 122.24         | <b>29.84</b>           | 11.94                                           |
| 128                                | 1                                  | 12.86                       | 34.82          | 30.50          | 1.42                  | 1.43                  | <b>81.02</b>   | 158.25                 | 10.53                                           |
|                                    | 2                                  | 14.96                       | 41.10          | 45.61          | 1.46                  | 1.97                  | 105.10         | 102.64                 | 9.46                                            |
|                                    | 5                                  | 11.22                       | 49.31          | 60.16          | 2.83                  | 2.86                  | 126.39         | <b>61.71</b>           | 11.01                                           |
| 256                                | 1                                  | 17.36                       | 50.34          | 32.96          | 1.80                  | 1.56                  | <b>104.02</b>  | 406.33                 | 9.59                                            |
|                                    | 2                                  | 13.70                       | 51.47          | 46.53          | 1.78                  | 2.09                  | 115.57         | 225.72                 | 8.93                                            |
|                                    | 5                                  | 13.76                       | 55.16          | 66.15          | 3.63                  | 3.14                  | 141.84         | <b>138.51</b>          | 10.43                                           |

The latency breakdown shows that the two functional bootstrappings dominate the computation, accounting for approximately 80% of the total A2B latency. Following Refresh<sub>I</sub>, Stage 1 computes the modular reduction used for shared overflow cancellation, while Stage 2 jointly evaluates the carry states and candidate bits. The subsequent carry recovery and bit correction require only

leveled homomorphic operations, with each taking approximately 1–4 seconds. The latency differences between encoding bases reflect the lookup-table sizes and the number of ciphertexts processed. For  $d = 4$ , each complete window contains six bits, compared with five for  $d = 2$ . Consequently, Stage 2 uses LUTs over a larger input domain and produces more candidate bits per window, increasing the functional bootstrapping cost. For  $d = 32$ , the additional cost comes from processing two ciphertexts instead of one. The special- $d_0$   $\mathcal{Z}$ -to- $\mathcal{C}$  transformation produces two ciphertexts containing the lower and upper halves of the modified triangle coefficient vectors. For  $d = 2$  and  $d = 4$ , only a subset of these coefficients is needed for the subsequent computation, allowing us to repack the required values into a single ciphertext before functional bootstrapping. However, For  $d = 32$ , however, we must retain all triangle coefficients therefore the Stage 2 latency for  $d = 32$  is approximately twice that for  $d = 2$ .

We compare the latency and amortized performance of our scheme with GZ26 [23] in Table 4. For  $d = 2$ , our scheme achieves latency speedups ranging from  $3.49\times$  at 64-bit precision to  $9.71\times$  at 256-bit precision. Due to both schemes have the same packing capacity, these gains stem from the efficiency of the A2B conversion itself. For  $d = 32$ , our scheme achieves amortized speedups of  $8.15\times$  and  $28.47\times$  at 64-bit and 256-bit precision, respectively. These gains stem from both the efficiency of our A2B procedure and our higher packing capacity. The speedup becomes more pronounced as the message precision increases. This is because GZ26 extracts windows sequentially, requiring more functional bootstrappings at higher precision. Our scheme retains two functional bootstrappings, while the multiplicative depth of leveled carry recovery grows only logarithmically with the message precision. This removes the sequential bootstrapping dependency across windows and leads to the slower latency growth observed in our experiments.

**Table 4.** Comparison against the GZ26 baseline.

| Word size<br>(bits) | Method            | Latency<br>(s) | Latency<br>Speedup | Packed<br>words | Amortized<br>(ms/word) | Amortized<br>Speedup |
|---------------------|-------------------|----------------|--------------------|-----------------|------------------------|----------------------|
| 64                  | GZ26 [23]         | 249.10         | 1.00×              | 1024            | 243.26                 | 1.00×                |
|                     | Ours ( $d = 2$ )  | 71.47          | <b>3.49×</b>       | 1024            | 69.80                  | 3.49×                |
|                     | Ours ( $d = 32$ ) | 122.24         | 2.04×              | 4096            | 29.84                  | <b>8.15×</b>         |
| 128                 | GZ26 [23]         | 503.00         | 1.00×              | 512             | 982.42                 | 1.00×                |
|                     | Ours ( $d = 2$ )  | 81.02          | <b>6.21×</b>       | 512             | 158.25                 | 6.21×                |
|                     | Ours ( $d = 32$ ) | 126.39         | 3.98×              | 2048            | 61.71                  | <b>15.92×</b>        |
| 256                 | GZ26 [23]         | 1009.60        | 1.00×              | 256             | 3943.75                | 1.00×                |
|                     | Ours ( $d = 2$ )  | 104.02         | <b>9.71×</b>       | 256             | 406.33                 | 9.71×                |
|                     | Ours ( $d = 32$ ) | 141.84         | 7.12×              | 1024            | 138.51                 | <b>28.47×</b>        |

## 6 Conclusion

We present a CKKS-based homomorphic ALU using general-base triangle encoding for SIMD-packed exact arithmetic over  $\mathbb{Z}_{d^n}$ , where  $d \geq 2$ . After refreshing the overflow  $I$ , our A2B conversion recovers canonical bits via shared overflow cancellation, using two functional bootstrapping stages followed by leveled carry propagation and bit correction. The number of functional bootstrapping stages

is independent of message precision. Our OpenFHE implementation evaluate it at message precisions of 64, 128, and 256 bits Compared with GZ26,  $d = 2$  achieves  $3.49\times\text{--}9.71\times$  latency speedups at the same packing capacity, while  $d = 32$  achieves  $8.15\times\text{--}28.47\times$  amortized speedups.

## References

1. Alexandru, A., Kim, A., Polyakov, Y.: General functional bootstrapping using ckks. In: Annual International Cryptology Conference. pp. 304–337. Springer (2025)
2. Badawi, A.A., Alexandru, A., Bates, J., Bergamaschi, F., Cousins, D.B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., Kim, A., Lee, Y., Liu, Z., Micciancio, D., Pascoe, C., Polyakov, Y., Quah, I., R.V., S., Rohloff, K., Saylor, J., Suponitsky, D., Triplett, M., Vaikuntanathan, V., Zucca, V.: OpenFHE: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915 (2022), <https://eprint.iacr.org/2022/915>, <https://eprint.iacr.org/2022/915>
3. Bae, Y., Cheon, J.H., Cho, W., Kim, J., Kim, T.: Meta-bts: Bootstrapping precision beyond the limit. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 223–234 (2022)
4. Bae, Y., Cheon, J.H., Kim, J., Stehlé, D.: Bootstrapping bits with ckks. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 94–123. Springer (2024)
5. Bae, Y., Kim, J., Stehlé, D., Suvanto, E.: Bootstrapping small integers with ckks. In: international conference on the theory and application of cryptology and information security. pp. 330–360. Springer (2024)
6. Bian, S., Fu, Y., Shen, R., Pan, H., Wang, A., Guan, Z.: High-precision functional bootstrapping for ckks from fourier extension. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 274–303. Springer (2026)
7. Boneh, D., Kim, J.: Homomorphic encryption for large integers from nested residue number systems. In: Annual International Cryptology Conference. pp. 338–370. Springer (2025)
8. Bossuat, J.P., Mouchet, C., Troncoso-Pastoriza, J., Hubaux, J.P.: Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 587–617. Springer (2021)
9. Bossuat, J.P., Troncoso-Pastoriza, J., Hubaux, J.P.: Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation. In: International Conference on Applied Cryptography and Network Security. pp. 521–541. Springer (2022)
10. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical gapsvp. In: Annual cryptology conference. pp. 868–886. Springer (2012)
11. Brakerski, Z., Friedman, O., Golan, D., Gurny, A., Mutzari, D., Sheinfeld, O.: Refhe: Fully homomorphic alu. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 183–213. Springer (2026)
12. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT) **6**(3), 1–36 (2014)
13. Cha, G., Park, D., Lee, J.W.: Improved radix-based approximate homomorphic encryption for large integers via lightweight bootstrapped digit carry. In: Annual

- International Conference on the Theory and Applications of Cryptographic Techniques. pp. 243–273. Springer (2026)
14. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: Bootstrapping for approximate homomorphic encryption. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 360–384. Springer (2018)
  15. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A full rns variant of approximate homomorphic encryption. In: International conference on selected areas in cryptography. pp. 347–368. Springer (2018)
  16. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: International conference on the theory and application of cryptology and information security. pp. 409–437. Springer (2017)
  17. Chillotti, I., Gama, N., Georgieva, M., Izabachene, M.: Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In: international conference on the theory and application of cryptology and information security. pp. 3–33. Springer (2016)
  18. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Tfhe: Fast fully homomorphic encryption over the torus: I. chillotti et al. *Journal of Cryptology* **33**(1), 34–91 (2020)
  19. Drucker, N., Moshkovich, G., Pelleg, T., Shaul, H.: Bleach: Cleaning errors in discrete computations over ckks: N. drucker et al. *Journal of Cryptology* **37**(1), 3 (2024)
  20. Ducas, L., Micciancio, D.: Fhew: bootstrapping homomorphic encryption in less than a second. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 617–640. Springer (2015)
  21. Dumezy, J., Alexandru, A., Polyakov, Y., Clet, P.E., Chakraborty, O., Boudguiga, A.: Evaluating larger lookup tables using ckks. *Cryptology ePrint Archive* (2025)
  22. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive* (2012)
  23. Gao, M., Zheng, H.: Fhe for simd arithmetic logic units with amortized o (1) bootstrapping per ciphertext. In: Annual International Cryptology Conference. pp. 299–330. Springer (2026)
  24. Geelen, R., Vercauteren, F.: Fully homomorphic encryption for cyclotomic prime moduli. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 366–397. Springer (2025)
  25. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the forty-first annual ACM symposium on Theory of computing. pp. 169–178 (2009)
  26. Kim, J.: Bootstrapping gbvf with ckks. *Cryptology ePrint Archive* (2025)
  27. Kim, J.: Efficient homomorphic integer computer from ckks. *IACR Transactions on Cryptographic Hardware and Embedded Systems* **2025**(4), 873–898 (2025)
  28. Kim, J.: Faster homomorphic integer computer. *Cryptology ePrint Archive* (2025)
  29. Kim, J., Noh, T.: Modular reduction in ckks. *Cryptology ePrint Archive* (2024)
  30. Kim, J., Seo, J., Song, Y.: Simpler and faster bfv bootstrapping for arbitrary plaintext modulus from ckks. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. pp. 2535–2546 (2024)
  31. Paterson, M.S., Stockmeyer, L.J.: On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM Journal on Computing* **2**(1), 60–66 (1973)
  32. Wei, Y., Ma, K., Gao, M., Zheng, H.: FHE for arbitrary modulus ALU and application to one-round threshold ECDSA. *Cryptology ePrint Archive*, Paper 2026/1836 (2026), <https://eprint.iacr.org/2026/1836>

## A Bounds for the Canonical Embeddings

We prove  $\|\sigma_{\mathcal{R}}\| \leq N$  and  $\|\sigma_{\mathcal{Z}}^{-1}\| < 2$ . Throughout,  $F(Y) = Y^n - Y + d$ , where  $n \geq 2$  is even and  $d \geq 2$ .

*Bound for  $\sigma_{\mathcal{R}}$ .* Let  $g(X) = \sum_{k=0}^{N-1} g_k X^k \in \mathcal{R}_{\mathbb{R}}$ . Every root  $\zeta$  used by  $\sigma_{\mathcal{R}}$  satisfies  $|\zeta| = 1$ . Hence  $|g(\zeta)| \leq \sum_{k=0}^{N-1} |g_k| \leq N\|g\|_{\infty}$ . Taking the maximum over the selected roots gives  $\|\sigma_{\mathcal{R}}(g)\|_{\infty} \leq N\|g\|_{\infty}$ , and therefore  $\|\sigma_{\mathcal{R}}\| \leq N$ .

*Bound for  $\sigma_{\mathcal{Z}}^{-1}$ .* First, every root  $\xi$  of  $F$  satisfies  $|\xi| > 1$ . Otherwise,  $2 \leq d = |\xi - \xi^n| \leq |\xi| + |\xi|^n \leq 2$ . Equality would force  $d = 2$ ,  $\xi = 1$ , and  $\xi^n = -1$ , which is impossible.

Let  $\xi_0, \dots, \xi_{n/2-1}$  be the roots defining  $\sigma_{\mathcal{Z}}$ . For each  $\xi_i$ , define the Lagrange polynomial  $L_i(Y) = F(Y)/((Y - \xi_i)F'(\xi_i)) = \sum_{k=0}^{n-1} \lambda_{i,k} Y^k$ . Polynomial division gives  $\lambda_{i,0} = (\xi_i^{n-1} - 1)/F'(\xi_i)$  and  $\lambda_{i,k} = \xi_i^{n-1-k}/F'(\xi_i)$  for  $1 \leq k < n$ .

Write  $r_i = |\xi_i| > 1$ . Since  $F'(\xi_i) = n\xi_i^{n-1} - 1$ , we have  $|F'(\xi_i)| \geq nr_i^{n-1} - 1 > n - 1$ . For the constant coefficient, the identity  $\lambda_{i,0} = 1/n - (n-1)/(nF'(\xi_i))$  gives  $|\lambda_{i,0}| \leq 1/n + (n-1)/(n|F'(\xi_i)|) < 2/n$ . For  $1 \leq k < n$ , we similarly obtain  $|\lambda_{i,k}| \leq r_i^{n-1-k}/(nr_i^{n-1} - 1) < 1/(n-1) \leq 2/n$ .

For any slot vector  $\mathbf{z} \in \mathbb{C}^{n/2}$ , interpolation using the conjugate root pairs gives  $(\sigma_{\mathcal{Z}}^{-1}(\mathbf{z}))_k = 2 \operatorname{Re} \sum_{i=0}^{n/2-1} \lambda_{i,k} z_i$ . Consequently,  $\|\sigma_{\mathcal{Z}}^{-1}\| \leq 2 \max_{0 \leq k < n} \sum_{i=0}^{n/2-1} |\lambda_{i,k}| < 2$ , because each sum contains  $n/2$  terms, each strictly smaller than  $2/n$ .

To quantify the effect of the encoding base more precisely, Table 5 reports the numerically evaluated bounds  $B_{\text{enc}} = N\|\sigma_{\mathcal{Z}}^{-1}\|/(2\Delta)$  and  $B_{\text{dec}} = N\|T_d\sigma_{\mathcal{Z}}^{-1}\|/(2\Delta)$  in bits, where  $T_d h = (Y - d)h \bmod F$ .

**Table 5.** Plaintext rounding-error bounds in bits at  $n = 32$ ,  $N = 65536$ , and  $\Delta = 2^{50}$ .  $B_{\text{enc}}$  and  $B_{\text{dec}}$  denote the numerically evaluated upper bounds on  $\|e_{\text{enc}}\|_{\infty}/\Delta$  and  $\|te_{\text{enc}}\|_{\infty}/\Delta$ , respectively. Values are rounded to two decimal places; larger values indicate smaller error bounds.

| Base $d - \log_2 B_{\text{enc}}$ (bits) $- \log_2 B_{\text{dec}}$ (bits) |       |       |
|--------------------------------------------------------------------------|-------|-------|
| 2                                                                        | 34.90 | 33.94 |
| 4                                                                        | 34.98 | 33.00 |
| 8                                                                        | 34.99 | 32.00 |
| 16                                                                       | 35.00 | 31.00 |
| 32                                                                       | 35.00 | 30.00 |

## B Proofs of the A2B Error Bounds

*Proof (Proof of Theorem 6).* Let  $H = \lceil \log_2 L \rceil$  and set  $E_0 = |e_{\text{in}}|/\Delta$ . We first analyze the leveled circuit under exact arithmetic on the perturbed input states;

the additional error introduced by homomorphic evaluation is accounted for at the end.

Consider one composition gate

$$F(x, y) = y + \pi(y)(x - y), \quad \pi(y) = 1 - 2(\operatorname{Re}(y) + \operatorname{Im}(y)),$$

where  $x, y \in \mathcal{S} = \{0, \frac{1}{2}, \frac{i}{2}\}$  are exact carry states. For every  $y \in \mathcal{S}$ ,  $\pi(y) \in \{0, 1\}$ , and for all  $x, y \in \mathcal{S}$ ,  $|x - y| \leq 1/\sqrt{2}$ . Suppose that the inputs to a gate at level  $h$  are  $x + e_x$  and  $y + e_y$ , with  $|e_x|, |e_y| \leq E_h$ . Define

$$\eta(e_y) = -2(\operatorname{Re}(e_y) + \operatorname{Im}(e_y)),$$

so that  $\pi(y + e_y) = \pi(y) + \eta(e_y)$ . Expanding the perturbed gate gives

$$\begin{aligned} F(x + e_x, y + e_y) - F(x, y) &= (1 - \pi(y))e_y + \pi(y)e_x \\ &\quad + \eta(e_y)(x - y) + \eta(e_y)(e_x - e_y). \end{aligned}$$

Because  $\pi(y) \in \{0, 1\}$ , the first two terms together have magnitude at most  $E_h$ . Moreover,

$$|\eta(e_y)| \leq 2(|\operatorname{Re}(e_y)| + |\operatorname{Im}(e_y)|) \leq 2\sqrt{2} E_h.$$

It follows that

$$|\eta(e_y)(x - y)| \leq 2E_h, \quad |\eta(e_y)(e_x - e_y)| \leq 4\sqrt{2} E_h^2,$$

and hence

$$E_{h+1} \leq 3E_h + 4\sqrt{2} E_h^2.$$

Set  $a = 4\sqrt{2}/3$ . Since  $E_h \geq 0$ , the preceding recurrence implies

$$1 + aE_{h+1} \leq 1 + 3aE_h + 3a^2 E_h^2 \leq (1 + aE_h)^3.$$

Iterating this inequality over the  $H$  levels yields

$$E_H \leq \frac{1}{a} \left[ (1 + aE_0)^{3^H} - 1 \right] = \frac{3}{4\sqrt{2}} \left[ \left( 1 + \frac{4\sqrt{2}}{3\Delta} |e_{\text{in}}| \right)^{3^H} - 1 \right].$$

Finally, carry extraction applies  $\chi(u) = u + \bar{u}$ . This map is 2-Lipschitz, since

$$|\chi(u + e) - \chi(u)| = |e + \bar{e}| \leq 2|e|.$$

Therefore, the error attributable to the perturbed input states is at most  $2\Delta E_H$ . Adding the aggregate homomorphic-evaluation error at the carry outputs gives

$$|e_{\text{carry}}| \leq 2\Delta E_H + |e_{\text{eval}}|.$$

Substituting the bound on  $E_H$  and  $H = \lceil \log_2 L \rceil$  proves the result.  $\square$

*Proof (Proof of Theorem 7).* Set  $\varepsilon = |e_{\text{in}}|/\Delta$  and  $\gamma = |e_{\text{carry}}|/\Delta$ . Fix a window index  $k$  and a bit position  $0 \leq t < w$ . Write the approximate inputs as

$$\tilde{v}_{k,j} = v_{k,j} + \delta_j, \quad \tilde{c}_{k-1} = c_{k-1} + \eta,$$

where  $|\delta_j| \leq \varepsilon$  and  $|\eta| \leq \gamma$ . Define

$$P_{k,t} = (1 - 2v_{k,t}) \prod_{j=0}^{t-1} (1 - v_{k,j}), \quad \tilde{P}_{k,t} = (1 - 2\tilde{v}_{k,t}) \prod_{j=0}^{t-1} (1 - \tilde{v}_{k,j}).$$

The exact corrected bit and the value obtained by evaluating the correction polynomial on the approximate inputs are therefore

$$d_{k,t} = v_{k,t} + c_{k-1}P_{k,t}, \quad \tilde{d}_{k,t} = \tilde{v}_{k,t} + \tilde{c}_{k-1}\tilde{P}_{k,t}.$$

If the exact factors in a product have magnitude at most one and their perturbations are bounded by  $\rho_1, \dots, \rho_s$ , then telescoping the difference between the perturbed and exact products gives the bound

$$\prod_{i=1}^s (1 + \rho_i) - 1.$$

Because  $v_{k,j}, c_{k-1} \in \{0, 1\}$ , the exact factors

$$c_{k-1}, \quad 1 - 2v_{k,t}, \quad \{1 - v_{k,j}\}_{j=0}^{t-1}$$

all have magnitude at most one, while their perturbations have magnitudes at most  $\gamma$ ,  $2\varepsilon$ , and  $\varepsilon$ , respectively. Consequently,

$$\begin{aligned} \left| \tilde{c}_{k-1}\tilde{P}_{k,t} - c_{k-1}P_{k,t} \right| &\leq (1 + \gamma)(1 + 2\varepsilon)(1 + \varepsilon)^t - 1 \\ &\leq (1 + \gamma)(1 + 2\varepsilon)^w - 1, \end{aligned}$$

where the second inequality follows from  $t + 1 \leq w$ . Hence,

$$|\tilde{d}_{k,t} - d_{k,t}| \leq \varepsilon + (1 + \gamma)(1 + 2\varepsilon)^w - 1.$$

The uncorrected bits in the initial window have error at most  $\varepsilon$  and therefore satisfy the same uniform bound. Finally, adding the aggregate homomorphic-evaluation error relative to exact evaluation on the approximate inputs gives

$$|e_{\text{bit}}| \leq |e_{\text{in}}| + \Delta \left[ \left( 1 + \frac{|e_{\text{carry}}|}{\Delta} \right) \left( 1 + \frac{2|e_{\text{in}}|}{\Delta} \right)^w - 1 \right] + |e_{\text{eval}}|,$$

as claimed. □

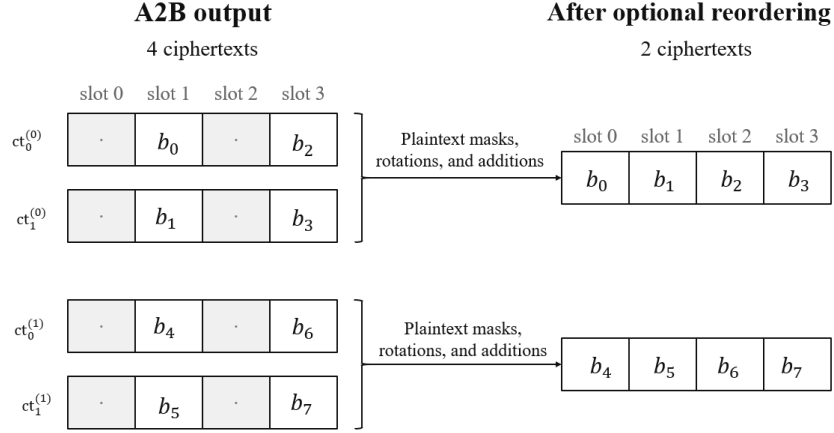


**Fig. 4.** Optional reordering of the A2B output. In the illustrated example, plaintext masks, rotations, and additions pack the four ciphertexts in the native output layout into two ciphertexts containing consecutive word bits. Dots denote slots unused in the illustrated packing.

## C Optional Reordering of A2B Outputs

Figure 4 illustrates the optional reordering of the output of Algorithm 4. In this example, the original layout stores  $b_0, b_2$  and  $b_1, b_3$  in two separate ciphertexts, and similarly stores  $b_4, b_6$  and  $b_5, b_7$  in another two ciphertexts. Plaintext masks select the relevant slots, rotations move the selected bits to consecutive positions, and additions combine the resulting ciphertexts. Consequently,  $b_0, \dots, b_3$  and  $b_4, \dots, b_7$  are packed into two ciphertexts, respectively. This transformation changes only the output layout and requires no additional functional bootstrapping.

## D Other ALU Components

This appendix describes the arithmetic refreshing procedures and Boolean operations used in our ALU, following the construction of Gao and Zheng [23].

### D.1 Arithmetic Mode

This subsection describes arithmetic refreshing and then reports the performance of arithmetic operations and refreshing.

**Arithmetic Refreshing** A complete arithmetic refresh first applies  $A2A_I$  and then  $A2A_e$ .

*Integer overflow refreshing.* Algorithm 5 refreshes the integer overflow while preserving the encoded message. It first removes the existing overflow by truncation and then uses sparse secret encapsulation to control the overflow introduced by modulus raising. After  $\mathcal{Z}$ -to- $\mathcal{R}$ , the RLWE plaintext has the form  $\Delta I_R + \Delta\mu + e_R$ , where  $I_R$  is an integer polynomial and  $\mu$  is the centered fractional message term with  $\|\mu\|_\infty < 1/2$ . The truncation is implemented by combining scalar multiplication with rescaling [23]:

$$\text{ct}_0 = \text{CKKS.Rescale}_{Q'_\ell \rightarrow q'_0} \left( \text{CKKS.MultScalar} \left( \text{ct}_R, \frac{Q'_\ell}{\Delta} \right) \right).$$

Scalar multiplication maps the accumulated overflow to  $Q'_\ell I_R$ , which vanishes modulo  $Q'_\ell$ . The subsequent rescaling retains  $\mu$  at scale  $q'_0$ , up to the rescaled approximation error and the additional rounding error.

---

**Algorithm 5:** Arithmetic integer overflow refreshing ( $\text{A2A}_I$ )

---

**Input:** An arithmetic-mode ciphertext  $\text{ct}_A = \text{Enc}_s(\Delta I + \frac{\Delta}{t}[m])$  at modulus  $Q'_\ell$ ; Evaluation keys  $\text{ksk}_{s \rightarrow \tilde{s}}$  and  $\text{ksk}_{\tilde{s} \rightarrow s}$ .

**Output:** An arithmetic-mode ciphertext  $\text{ct}'_A = \text{Enc}_s(\Delta \tilde{I} + \frac{\Delta}{t}[m])$  with refreshed overflow  $\tilde{I}$

```

1  $\text{ct}_R \leftarrow \mathcal{Z}\text{-to-}\mathcal{R}(\text{ct}_A)$ 
2  $\text{ct}_0 \leftarrow \text{CKKS.Truncate}_{Q'_\ell \rightarrow q'_0}(\text{ct}_R)$ 
3  $\text{ct}_1 \leftarrow \text{KeySwitch}_{s \rightarrow \tilde{s}}(\text{ct}_0; \text{ksk}_{s \rightarrow \tilde{s}})$ 
4  $\text{ct}_2 \leftarrow \text{CKKS.ModRaise}_{q'_0 \rightarrow Q'_L}(\text{ct}_1)$ 
5  $\text{ct}_3 \leftarrow \text{KeySwitch}_{\tilde{s} \rightarrow s}(\text{ct}_2; \text{ksk}_{\tilde{s} \rightarrow s})$ 
6  $\text{ct}'_A \leftarrow \mathcal{R}\text{-to-}\mathcal{Z}(\text{ct}_3)$ 
7 return  $\text{ct}'_A$ 
```

---

The truncated ciphertext is switched to the sparse encapsulated key  $\tilde{s}$  before its modulus is raised to  $Q'_L$ . Modulus raising introduces a new term  $q'_0 I'$  whose magnitude is governed by the sparse secret key. Switching back to  $s$  and applying  $\mathcal{R}$ -to- $\mathcal{Z}$  returns an arithmetic ciphertext with refreshed overflow  $\tilde{I}$  and the same encoded message.

*Arithmetic error refreshing.* This procedure is inspired by META-BTS and CKKS-based BFV/GBFV bootstrapping [3,30,26], and its CKKS.Sine step follows [8]. For each packed word, write

$$f = \Delta I + \Delta t^{-1}[m] + e, \quad \varepsilon = \frac{e}{\Delta}, \quad \eta = t\varepsilon.$$

Multiplication by  $t$  gives  $tf = \Delta(tI + [m] + \eta)$ . Thus,  $tI + [m]$  is the integer component, whereas  $\eta = t\varepsilon$  is the normalized error to be refreshed. Algorithm 6 removes the integer component, estimates  $\eta$  using CKKS.Sine, maps the estimate

back through multiplication by  $t^{-1}$ , and subtracts it from the input ciphertext. The procedure applies whenever the transformed error lies within the certified approximation range of CKKS.Sine.

---

**Algorithm 6:** Arithmetic error refreshing ( $A2A_e$ )

---

**Input:** Arithmetic ciphertext  $ct_A$  at modulus  $Q'_\ell$  and scale  $\Delta$ , with normalized word representation  $I + t^{-1}[m] + \varepsilon$ ; evaluation keys, including  $ksk_{s \rightarrow \tilde{s}}$  and  $ksk_{\tilde{s} \rightarrow s}$ .  
**Output:** Arithmetic ciphertext  $ct'_A$  representing  $I + t^{-1}[m] + \varepsilon'$ , with refreshed error  $\varepsilon'$ .

```

1  $ct_R \leftarrow \mathcal{Z}\text{-to-}\mathcal{R}^{(t)}(ct_A)$ ;
2  $ct_0 \leftarrow \text{CKKS.Truncate}(ct_R; q'_0)$ ;
3  $ct_1 \leftarrow \text{KeySwitch}_{s \rightarrow \tilde{s}}(ct_0; ksk_{s \rightarrow \tilde{s}})$ ;
4  $ct_2 \leftarrow \text{CKKS.ModRaise}_{q'_0 \rightarrow Q'_L}(ct_1)$ ;
5  $ct_3 \leftarrow \text{KeySwitch}_{\tilde{s} \rightarrow s}(ct_2; ksk_{\tilde{s} \rightarrow s})$ ;
6  $(ct_{C,0}, ct_{C,1}) \leftarrow \mathcal{R}\text{-to-}\mathcal{C}(ct_3)$ ;
7 foreach  $b \in \{0, 1\}$  do
8    $ct_{E,b} \leftarrow \text{CKKS.Sine}(ct_{C,b})$ ;
9  $ct_\varepsilon \leftarrow \mathcal{C}\text{-to-}\mathcal{Z}^{(t^{-1})}(ct_{E,0}, ct_{E,1})$ ;
10  $(ct_{\text{keep}}, ct_\varepsilon) \leftarrow \text{Align}(ct_A, ct_\varepsilon)$  ;           // Match moduli and scales before
    subtraction
11  $ct'_A \leftarrow ct_{\text{keep}} - ct_\varepsilon$ ;
12 return  $ct'_A$ 
```

---

*Error analysis.* After truncation, modulus raising, and the  $\mathcal{R}$ -to- $\mathcal{C}$  transformation, each input to CKKS.Sine has the form  $J_i + \eta_i$  up to newly introduced CKKS error, where  $J_i \in \mathbb{Z}$ . Let  $\hat{\eta}$  denote the error estimate returned by this branch. Since the normalized sine  $\phi(x) = \frac{\sin(2\pi x)}{2\pi}$  is period one and satisfies  $\phi(x) = x + O(x^3)$  around every integer, we have  $\|\hat{\eta} - \eta\|_\infty \leq C_{\text{sin}}\|\eta\|_\infty^3 + E_{\text{boot}}$ , where  $E_{\text{boot}}$  collects the errors introduced by the forward transformations, truncation, key switching, modulus raising, and the Sine approximation and evaluation.

After the back-transformation, alignment, and final subtraction, the original error  $\varepsilon$  cancels. Hence, for an accumulated post-processing error  $\delta_{\text{post}}$  satisfying  $\|\delta_{\text{post}}\|_\infty \leq E_{\text{post}}$ , the refreshed error satisfies  $t\varepsilon' = -(\hat{\eta} - \eta) + \delta_{\text{post}}$ , and therefore  $\|t\varepsilon'\|_\infty \leq C_{\text{sin}}\|t\varepsilon\|_\infty^3 + E_{\text{boot}} + E_{\text{post}}$ . Consequently, decoding recovers the original message whenever the right-hand side is smaller than  $1/2$ .

**Performance** Table 6 reports addition, multiplication, integer overflow refreshing, and arithmetic error refreshing.

**Table 6.** Single-thread arithmetic operation and refreshing performance at  $N = 2^{16}$ . Addition and multiplication results are averaged over 100 measurements, and refreshing results over five measurements, each after one warm-up run. Here  $\ell$  is the message precision.  $A2A_e$  is measured only for  $d = 2$ ; “–” denotes configurations not measured.

| $\ell$<br>(bits) | $\log_2 d$<br>(bits) | Addition        |                 | Multiplication  |                 | $A2A_I$        |                 | $A2A_e$        |                 |
|------------------|----------------------|-----------------|-----------------|-----------------|-----------------|----------------|-----------------|----------------|-----------------|
|                  |                      | Latency<br>(ms) | Error<br>(bits) | Latency<br>(ms) | Error<br>(bits) | Latency<br>(s) | Error<br>(bits) | Latency<br>(s) | Error<br>(bits) |
| 64               | 1                    | 0.66            | 32.63           | 38.28           | 24.03           | 12.21          | 18.63           | 26.01          | 20.72           |
|                  | 2                    | 0.63            | 32.13           | 144.10          | 18.53           | 11.45          | 16.25           | –              | –               |
|                  | 5                    | 0.61            | 31.53           | 142.80          | 18.07           | 10.87          | 12.68           | –              | –               |
| 128              | 1                    | 0.63            | 32.96           | 39.43           | 19.08           | 12.86          | 16.80           | 31.14          | 20.49           |
|                  | 2                    | 0.67            | 32.70           | 147.03          | 17.56           | 14.96          | 16.27           | –              | –               |
|                  | 5                    | 0.75            | 32.11           | 146.46          | 12.99           | 11.22          | 10.71           | –              | –               |
| 256              | 1                    | 0.65            | 33.61           | 41.51           | 18.12           | 17.36          | 16.83           | 42.22          | 20.78           |
|                  | 2                    | 0.69            | 33.07           | 149.58          | 16.47           | 13.70          | 13.96           | –              | –               |
|                  | 5                    | 0.62            | 32.72           | 146.39          | 11.95           | 13.76          | 10.66           | –              | –               |

## D.2 Boolean Mode

The A2B conversion returns the canonical bit decomposition of each packed word. Let

$$m = \sum_{j=0}^{\ell-1} b_j 2^j, \quad b_j \in \{0, 1\}, \quad \mathbf{b} = (b_0, \dots, b_{\ell-1}).$$

At CKKS scale  $\Delta$ , one Boolean word is represented by

$$\text{BoolEnc}_{\Delta}(m) = \Delta \mathbf{b} + \mathbf{e} \quad \text{in } \mathbb{C}^{\ell}. \quad (9)$$

The bits are distributed across the Boolean output ciphertexts in the native A2B layout; optional reordering places consecutive blocks of bits from each packed word into fewer ciphertexts. Bitwise operations act componentwise on the encoded bits and introduce no Triangle-overflow polynomial  $I$ .

*Bitwise operations and Boolean refreshing.* For ideal scaled bit vectors  $x = \Delta \mathbf{b}$  and  $y = \Delta \mathbf{b}'$ , the standard multilinear polynomials give

$$\begin{aligned} \text{AND}(x, y) &= \frac{1}{\Delta} xy, & \text{OR}(x, y) &= x + y - \frac{1}{\Delta} xy, \\ \text{XOR}(x, y) &= x + y - \frac{2}{\Delta} xy, & \text{NOT}(x) &= \Delta \mathbf{1} - x. \end{aligned} \quad (10)$$

Products are componentwise and are evaluated using CKKS multiplication, relinearization, and rescaling. For Boolean vectors  $\mathbf{b}$  and  $\mathbf{b}'$ , these formulas return  $\Delta$  times the corresponding componentwise Boolean result. NAND, NOR, and XNOR follow by composition. After several leveled operations, Boolean-to-Boolean refreshing applies discrete CKKS functional bootstrapping to return each coordinate to a noisy encoding of either 0 or  $\Delta$  [23].

*Word-local shifts and rotations.* For a public shift or rotation amount, word-local shifts and rotations are implemented using plaintext masks, slot rotations, and ciphertext additions. The masks select the bits that contribute to each destination block, and the selected bits are rotated and accumulated into the corresponding Boolean ciphertext. Logical shifts discard bits outside the word and set the vacated positions to zero, whereas word rotations wrap these bits to the opposite end. These are linear operations and require no ciphertext multiplication.

*Sign access and comparison.* For a two’s-complement word, the sign bit is  $b_{\ell-1}$ . It is selected by a plaintext mask and rotated to the designated output positions. Signed comparison combines the operand sign bits with the sign bit of their modular difference: operands with different signs are ordered by their sign bits, whereas operands with the same sign are ordered by the sign of their difference. The difference is computed in arithmetic mode and its sign bit is obtained by A2B conversion. Equality is evaluated by combining the bitwise XNOR results with an AND tree.

*Performance.* Table 7 reports Boolean refreshing and bitwise operations with the base-2 parameter configuration. The timings cover the complete Boolean batch of two ciphertexts.

**Table 7.** Single-thread Boolean refreshing and bitwise operation performance at  $N = 2^{16}$ . Boolean refreshing and bitwise operation results are averaged over five and 100 measurements, respectively, after one warm-up run.

| Operation | 64-bit         |                        | 128-bit        |                        | 256-bit        |                        |
|-----------|----------------|------------------------|----------------|------------------------|----------------|------------------------|
|           | Latency<br>(s) | Output error<br>(bits) | Latency<br>(s) | Output error<br>(bits) | Latency<br>(s) | Output error<br>(bits) |
| B2B       | 21.59          | 19.23                  | 22.80          | 19.23                  | 24.16          | 19.23                  |
| AND       | 0.62           | 28.68                  | 0.64           | 28.81                  | 0.69           | 28.88                  |
| OR        | 0.74           | 29.48                  | 0.76           | 29.58                  | 0.85           | 29.38                  |
| XOR       | 0.73           | 28.15                  | 0.77           | 28.24                  | 0.87           | 28.28                  |
| NOT       | 0.02           | 29.61                  | 0.02           | 30.00                  | 0.03           | 29.95                  |