Faster Logical Operations from Discrete CKKS

Jaehyung Kim

Stanford University, Stanford, USA
jaehk@cs.stanford.edu

Abstract. We study non-arithmetic operations for BFV with arbitrary plaintext moduli and for large-modulus GBFV. Existing approaches either rely on interpolation over large plaintext spaces or use digit representations that are efficient for logic but poorly matched to native arithmetic. We build conversions between BFV/GBFV and Discrete CKKS, so that radix-based integer computation in Discrete CKKS can be used as a reusable backend. For fixed radix base, comparison uses $O(\log p \log\log p)$ bootstrappings for BFV with plaintext modulus p and $O(\log\log p)$ bootstrappings for GBFV with associated modulus $p = b^\ell + 1$.

Keywords: GBFV · BFV · Discrete CKKS · Logical Operations

1 Introduction

Fully homomorphic encryption (FHE) schemes are optimized for different types of computation: BGV [11] and BFV [10, 22] for SIMD integer arithmetic, DM [21] and CGGI [19] for low-latency computation, and CKKS [17] for real-number arithmetic. BGV/BFV is therefore a natural choice for integer-heavy SIMD workloads such as private information retrieval (PIR) [34] and private set intersection (PSI) [16].

A persistent limitation of BGV/BFV is that its arithmetic representation is poorly suited to non-arithmetic operations. While addition and multiplication are natively supported, operations such as comparison, digit decomposition, and bitwise computation are often implemented through polynomial interpolation. For a prime modulus p, a univariate interpolation polynomial of degree $p-1$ can evaluate an arbitrary function over $\mathbb{F}_p$, but this approach quickly becomes prohibitive as p grows. A standard way to bypass this $O(p)$ barrier is to work with a digit-decomposed representation. For example, state-of-the-art BGV/BFV comparison methods [25, 36] compare integers represented as $x = \sum_{i=0}^{d-1} \rho^i \cdot x_i$, where each digit x_i lies in $[0, \rho)$ for a small base prime ρ. This digit representation supports logical operations efficiently, but it differs from the compact representation used for arithmetic. The conversion of [38] maps the compact representation $\mathbb{Z}_{\rho^d}$ to the digit representation $\mathbb{Z}_\rho^d$ with multiplicative depth $O(d)$ for fixed ρ, logarithmic in the plaintext modulus ρ^d.

This approach relies on the plaintext modulus being a power of a small prime. Many applications instead require a large or NTT-friendly prime plaintext

modulus, including homomorphic signing [5, 7], FHE-SNARKs [37], and SPDZ matrix-tuple generation [13]. In particular, an NTT-friendly prime modulus is essential for full SIMD support, since the ring $\mathcal{R}_p$ needs to split completely. The challenge is therefore to connect native BGV/BFV arithmetic modulo a large plaintext modulus with logical computation on small-base radix digits. This motivates the following question:

> *Can we support efficient non-arithmetic operations for BGV/BFV with arbitrary plaintext moduli?*

One recent approach to this question is the functional bootstrapping framework of Lee, Min, and Song [31], which evaluates functions during bootstrapping by reducing them to lookup-table evaluation over prime-power plaintext spaces. Their framework includes optimized procedures for functions such as sign and delta. We instead use Discrete CKKS [20, 3, 4, 1] to separate arithmetic from logical computation. Arithmetic remains in the native BFV or GBFV representation; when logical operations are needed, we convert ciphertexts to a radix-based Discrete-CKKS representation. An existing homomorphic integer computer can then perform comparison, digit decomposition, bitwise computation, and related operations on that representation, without a separate lookup table for each target function.

For BFV, the main obstacle is that modulus raising does not directly produce a clean encoding of the message m; instead, it gives a noisy Discrete-CKKS ciphertext encrypting $m + pI$ for a bounded integral polynomial I. Our conversion extracts radix digits and removes the extra pI term by homomorphic modular reduction, recovering a Discrete-CKKS representation of the original BFV message modulo p. For GBFV [23, 13], a generalization of BFV, the plaintext polynomial exposes a digit-decomposition structure more directly. Writing the GBFV plaintext polynomial as $t(X) = X^h - b$, we set $\ell = N/h$ and $p = b^\ell + 1$. We pack the exposed digits into a length-2ℓ representation in the integer computer of [12] and use modular reduction to obtain canonical digits.

1.1 Results and Contributions

Our main contribution is a bidirectional conversion framework that uses Discrete CKKS as a reusable backend for logical operations in BFV and GBFV. The following bootstrapping bounds assume a fixed radix base and a bounded lifting range.

Arbitrary-modulus BFV. For BFV with plaintext modulus p and a radix base coprime to p, we construct a conversion to the radix-based integer representation of [27] and a conversion back to BFV. The forward conversion and the resulting digit decomposition and comparison each use $O(\log p \log\log p)$ bootstrappings. To obtain this conversion, we introduce an arbitrary-modulus reduction routine built from the integer computer's arithmetic operations. The reverse conversion allows subsequent arithmetic to resume in BFV after logical operations have been evaluated in Discrete CKKS.

Large-modulus GBFV. For GBFV with plaintext polynomial $t(X) = X^h - b$ and associated modulus $p = b^\ell + 1$, where $\ell = N/h \leq N/8$, we give an exact conversion to the length-2ℓ packed integer representation of [12]. Conversion and GBFV comparison each use $O(\log \ell) = O(\log \log p)$ bootstrappings. The same interface supports digit decomposition and bitwise operations, as well as conversion back to GBFV.

Sparse packing. Sparse BFV ciphertexts reach the packed integer-computer interface using $O(\log \log p)$ bootstrappings, plus lightweight digit extraction. For sparse CKKS ciphertexts, comparison uses $O(\log \ell)$ rather than $O(\ell)$ bootstrappings in the radix digit length ℓ. These variants extend the same conversion principle to sparse coefficient layouts.

1.2 Technical Overview

We first describe the BFV conversion. Let N be a power-of-two integer and let q be a ciphertext modulus. Let $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ and $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$. Consider a coefficient-encoded BFV ciphertext encrypting $m \in \mathcal{R}_p$ with plaintext modulus p. Such a ciphertext can be written as

$$\mathsf{ct} = (c_0, c_1) = \left(-c_1 s + \left\lfloor \frac{q}{p} \cdot m \right\rceil + e, c_1 \right),$$

where $s \in \mathcal{R}$ is the secret key. Equivalently, as an RLWE ciphertext, it satisfies

$$\mathsf{ct} \cdot (1, s) = \left\lfloor \frac{q}{p} \cdot m \right\rceil + e.$$

The BFV-to-Discrete-CKKS conversion has three main steps.

1. **Modulus raising.** After modulus raising, we obtain

$$\mathsf{ModRaise}(\mathsf{ct}) \cdot (1, s) = \left\lfloor \frac{q}{p} \cdot m \right\rceil + e + qI$$

for some bounded integral polynomial $I \in \mathcal{R}$. Since $qI = (q/p) \cdot pI$, the right-hand side can be viewed as a noisy Discrete-CKKS encryption of $m + pI$ with scaling factor $\Delta = q/p$:

$$\mathsf{ModRaise}(\mathsf{ct}) \cdot (1, s) = \lfloor \Delta \cdot (m + pI) \rceil + e.$$

2. **Radix decomposition.** The iterative MSB bootstrapping of [29] decomposes a Discrete-CKKS ciphertext into radix digits. Because the lifting polynomial I may have signed coefficients, we first add a public coefficientwise shift $pK\mathbf{1}_\mathcal{R}$, where $\mathbf{1}_\mathcal{R} = 1 + X + \cdots + X^{N-1}$, so that every coefficient of $m + pI + pK\mathbf{1}_\mathcal{R}$ is nonnegative and representable by k base-b digits. Let

$$m + pI + pK\mathbf{1}_\mathcal{R} = \sum_{i=0}^{k-1} b^i \cdot m_i$$

be the resulting coefficientwise base-b digit decomposition. The output consists of k Discrete-CKKS ciphertext pairs $\mathsf{ct}_i = (\mathsf{ct}_i^R, \mathsf{ct}_i^I)$ for $0 \le i < k$, where the two ciphertexts in each pair jointly encrypt the i-th digit vector m_i. Since I has bounded coefficients, we have $k = \log_b p + O(1)$. This matches the radix representation used by the integer computer of [27].

3. **Homomorphic modular reduction.** The integer computer of [27] supports addition, subtraction, multiplication, comparison, and shifts, so we can implement homomorphic modular reduction by p inside this representation. We instantiate Montgomery reduction using these operations, following the same principle as in [12]. This reduces the radix representation of $m + pI + pK\mathbf{1}_{\mathcal{R}}$ to a radix representation of $m \bmod p$, since the additional terms are multiples of p. Homomorphic multiplication inside the integer computer dominates the bootstrapping count, giving $O(k \log k) = O(\log p \log\log p)$ bootstrappings.

The conversion itself gives a digit decomposition for arbitrary-modulus BFV, avoiding the $O(p)$ interpolation approach over $\mathbb{F}_p$. To compare two BFV ciphertexts, we convert both inputs, evaluate integer-computer comparison, and convert the resulting bit back to BFV. The reverse conversion cleans the radix digits if necessary, recombines them, applies the slot-to-coefficient transformation, and reduces the resulting coefficients modulo the BFV ciphertext modulus. For fixed radix base and bounded lifting range, this comparison uses $O(\log p \log\log p)$ bootstrappings. Other logical and bitwise operations use the same interface.

We next extend the idea to GBFV [23, 13]. Let h be a power-of-two divisor of N, let $b > 1$, let $t(X) = X^h - b$, and set $\ell = N/h$ and $p = b^\ell + 1$. A GBFV ciphertext encrypting $m \in \mathcal{R}_t = \mathcal{R}/t\mathcal{R}$ can be written as

$$\mathsf{ct} = (c_0, c_1) = (-c_1 s + \lfloor (q/t) \cdot m \rceil + e, c_1),$$

so that

$$\mathsf{ct} \cdot (1, s) = \lfloor (q/t) \cdot m \rceil + e.$$

After modulus raising, we have

$$\mathsf{ModRaise}(\mathsf{ct}) \cdot (1, s) = \lfloor (q/t) \cdot m \rceil + e + qI$$

for some bounded integral polynomial $I \in \mathcal{R}$. Multiplying by t gives a ciphertext ct' satisfying

$$\mathsf{ct}' \cdot (1, s) = q \cdot (m + tI) + e',$$

where e' remains small under the usual GBFV correctness conditions. The polynomial $m + tI$ can be interpreted as a bounded signed radix representation of the h plaintext coordinates of m modulo $p = b^\ell + 1$. RLWE-to-LWE extraction, permutation, and ring packing [2] place these digits in the Discrete-CKKS integer representation of [12]. Homomorphic modular reduction by p in a length-2ℓ representation then gives the exact base-b digits of the canonical representatives of m. This modular reduction dominates the bootstrapping count, and the conversion uses $O(\log \ell) = O(\log\log p)$ bootstrappings for fixed b and bounded lifting range. Consequently, logical operations for GBFV, including comparison, can be evaluated using $O(\log\log p)$ bootstrappings under the same assumptions.

1.3 Related Work

Exact and approximate arithmetic. Since the first construction of FHE [24], research has improved both its efficiency and its range of supported computations. The BFV scheme of Brakerski, Fan, and Vercauteren [10, 22] provides conceptually simple homomorphic integer arithmetic by placing the message in the most significant bits and separating it from the error accumulated in the least significant bits. The scheme of Chen et al. [15] instead chooses a polynomial plaintext modulus $t(X) = X - b$ and achieves high-precision computation with significantly less noise growth. That construction did not provide a bootstrapping procedure or BFV's SIMD capability, which limited its use. Recently, two independent works [23, 13] extended the plaintext space to a general ideal, so that $t(X)$ can be an arbitrary polynomial, and obtained a useful middle ground between BFV and CLPX. In particular, [23] gave the first bootstrapping procedure for this type of scheme.

The CKKS scheme of Cheon et al. [17] is widely used for privacy-preserving real-number computation and machine learning because it provides efficient SIMD approximate arithmetic. A recent line of work develops Discrete CKKS, which adapts CKKS-style techniques to discrete computation [20]. A central tool in this line is CKKS-style functional or programmable bootstrapping [3, 4, 1], which evaluates a function during bootstrapping. Building on these techniques, several works [27, 6, 28, 12] construct integer-computation units that support high-precision integer arithmetic and logical operations. Our work uses these integer computers as backends for logical operations in arbitrary-modulus BFV and large-modulus GBFV.

Scheme switching. Cross-scheme conversion combines the strengths of different FHE schemes. Chimera [9] and Pegasus [33], for example, provide scheme-switching frameworks that include conversions between LWE and RLWE ciphertexts. These frameworks, however, do not address efficient conversion between BFV and CKKS for logical operations over arbitrary BFV plaintext moduli. More closely related are CKKS-assisted (G)BFV bootstrapping methods [30, 26], which regard the (G)BFV error as a CKKS plaintext and use CKKS bootstrapping to refresh (G)BFV ciphertexts. In particular, [26] gives conversions between GBFV and CKKS. From the perspective of digit decomposition, its output is approximate or redundant: the components are bounded signed digits of size $O(b)$ rather than canonical digits in $[0, b)$. Our GBFV conversion uses modular reduction in the integer computer to obtain an exact canonical digit representation.

Non-arithmetic operations. Non-arithmetic operations in BFV, especially comparison, have long been challenging. The straightforward approach evaluates the desired function by interpolation over $\mathbb{F}_p$, which costs $O(p)$ for a prime modulus p. Prior comparison methods therefore often assume that the input is already digit-decomposed over a small base [25, 36]. To connect such digit-decomposed

computation with ordinary arithmetic, [38] introduced an asymptotically efficient conversion from $\mathbb{Z}_{\rho^d}$ to $\mathbb{Z}_\rho^d$. This supports prime-power plaintext moduli with small base ρ, but does not directly address general large prime moduli.

Lee, Min, and Song [31] recently introduced a functional bootstrapping framework for BFV with plaintext-modulus switching. They start from a BFV ciphertext with plaintext modulus t, reinterpret it through modulus switching as a noisy encryption over an intermediate ring $\mathbb{Z}_{\rho^r}$, and evaluate a lookup table by functional bootstrapping. Their framework outputs a ciphertext over $\mathbb{Z}_{\rho^s}$ and efficiently evaluates several functions, including Heaviside/sign-style functions. Our approach is complementary: conversion produces a radix-based Discrete-CKKS representation that supports multiple logical operations before conversion back to BFV or GBFV.

Concurrent work. A concurrent and independent work [35] also uses bidirectional switching between BFV-style and CKKS-style ciphertexts. Both works use BFV-style representations for modular arithmetic and CKKS-style representations for comparison-oriented computation. The switching in [35] serves the bit and small-integer computations needed for threshold Dilithium: the BFV-to-CKKS direction is used with bit extraction for small bit-length values, and the CKKS-to-BFV direction converts encrypted bits for reconstruction and modular arithmetic. Our conversion instead produces a reusable radix representation for arbitrary BFV plaintext moduli, reduces homomorphically modulo p, and supports conversion back for arbitrary residues. We also treat large-modulus GBFV and sparse-packing variants for BFV and CKKS.

Organization. Section 2 introduces the schemes and integer-computer interfaces. Sections 3 and 4 present the BFV and GBFV conversions and their applications. Section 5 develops the sparse-packing variants, and Section 6 reports the experiments.

2 Preliminaries

Let $N \geq 2$ be a power of two, let $q > 0$ be a ciphertext modulus, and let $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ and $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$. For an integral polynomial $\mu(X)$, the notation $\alpha \leq \mu < \beta$ means that every coefficient of μ lies in $[\alpha, \beta)$. All plaintexts are coefficient-encoded unless a slot encoding is explicitly mentioned. For an integer modulus m, $[x]_m$ denotes the representative of x in $[0, m)$; we apply this notation coefficientwise or coordinatewise as appropriate. We write $\bar{x}$ for the canonical representative of a residue class x when an order comparison is involved. For equal-length vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, $\mathbf{1}[\boldsymbol{x} \geq \boldsymbol{y}]$ is the componentwise comparison bit vector obtained by comparing canonical representatives. For a radix-b vector $\mathbf{z} = (z_0, \ldots, z_{L-1})$, possibly with lazy digits outside $[0, b)$, write $\mathrm{val}_b(\mathbf{z}) = \sum_{i=0}^{L-1} b^i \cdot z_i$.

Efficiency measure. Our asymptotic bounds count bootstrapping invocations, including discrete functional bootstrappings and ordinary refresh bootstrappings, for the specified packing layout. The experiments measure execution time, which also includes linear transformations such as HomDFT, extraction, ring packing, and other homomorphic operations.

2.1 BFV and GBFV

BFV and GBFV are RLWE encryption schemes over $\mathcal{R}$ with ciphertexts in $\mathcal{R}_q^2$ and secret key $\mathsf{sk} = (1, s)$. The plaintext space is $\mathcal{R}_t = \mathcal{R}/t\mathcal{R}$. Given $m \in \mathcal{R}_t$, encryption samples a uniformly from $\mathcal{R}_q$ and a small error polynomial e, and outputs

$$\mathsf{GBFV.Enc}(\mathsf{sk}, m) = \left(-as + \left\lfloor \frac{q}{t} \cdot m \right\rceil + e, a\right) \in \mathcal{R}_q^2.$$

The resulting ciphertext $\mathsf{ct} = (c_0, c_1)$ satisfies

$$\mathsf{ct} \cdot \mathsf{sk} = c_0 + c_1 s = \left\lfloor \frac{q}{t} \cdot m \right\rceil + e \pmod{q}$$

for sufficiently small error e. Decryption computes $\lfloor (\mathsf{ct} \cdot \mathsf{sk})/(q/t) \rceil \bmod t$. For BFV, $t(X) = p$ and a plaintext $m = \sum_i m_i \cdot X^i \in \mathcal{R}_p$ is identified with the coefficient vector $(m_0, \ldots, m_{N-1}) \in \mathbb{Z}_p^N$. For GBFV, $t(X) = X^h - b$ with $h \mid N$, $\ell = N/h$, and $p = b^\ell + 1$. In this case

$$\frac{1}{t} = -\frac{b^{\ell-1} + b^{\ell-2} \cdot X^h + \cdots + X^{N-h}}{b^\ell + 1},$$

so expressions such as $\lfloor u/t \rceil$ mean multiplication by this rational polynomial followed by coefficientwise rounding. Writing

$$m = \sum_{a=0}^{h-1} \sum_{j=0}^{\ell-1} u_{a,j} \cdot X^{a+jh}$$

identifies the GBFV plaintext with $(m_0, \ldots, m_{h-1}) \in \mathbb{Z}_p^h$, where

$$m_a = \sum_{j=0}^{\ell-1} b^j \cdot u_{a,j} \pmod{p}$$

for $0 \le a < h$. Thus BFV comparison is coefficientwise comparison in $[0, p)^N$, and GBFV comparison is coordinatewise comparison in $[0, p)^h$. Our constructions use these coefficient encodings rather than the SIMD encoding obtained by factoring the plaintext ring.

Homomorphic operations. Addition and multiplication evaluate the corresponding operations in $\mathcal{R}_t$ as long as the accumulated error stays within the decryption threshold. Bootstrapping refreshes a ciphertext while preserving its plaintext,

reducing the error relative to the available ciphertext modulus. We use exact-scheme addition, multiplication, bootstrapping, modulus switching, and modulus raising as black boxes. Modulus raising has the following property: if $\mathsf{ct}\cdot\mathsf{sk} = u+e$ modulo q, then the coefficient-lifted ciphertext satisfies

$$\mathsf{ModRaise}(\mathsf{ct}) \cdot \mathsf{sk} = u + e + qI$$

over the integers for some integral lifting polynomial I.

2.2 CKKS

The CKKS scheme [17] encodes vectors of complex numbers using the discrete Fourier transform

$$\mathsf{DFT}(\alpha(X)) = (\alpha(\zeta^{5^i}))_{0 \le i < N/2},$$

where ζ is a primitive $2N$-th root of unity. For a scaling factor $\Delta > 0$, the slot-encoding and decoding maps are

$$\mathsf{CKKS.Ecd}(\boldsymbol{z}) = \lfloor \Delta \cdot \mathsf{DFT}^{-1}(\boldsymbol{z}) \rceil, \qquad \mathsf{CKKS.Dcd}(u) = \frac{1}{\Delta} \cdot \mathsf{DFT}(u).$$

Their composition is an approximate identity on $\mathbb{C}^{N/2}$. Coefficient encoding instead places scaled values directly in the polynomial coefficients.

Encryption and decryption. For a slot vector $\boldsymbol{z}$, encryption under $\mathsf{sk} = (1, s)$ outputs

$$\mathsf{CKKS.Enc}(\mathsf{sk}, \boldsymbol{z}) = (-as + \mathsf{CKKS.Ecd}(\boldsymbol{z}) + e, a) \in \mathcal{R}_q^2,$$

where a is uniform in $\mathcal{R}_q$ and e is a small error polynomial. Decryption returns $\mathsf{CKKS.Dcd}(\mathsf{ct} \cdot \mathsf{sk})$, an approximation to $\boldsymbol{z}$. We distinguish the underlying polynomial plaintext $\mathsf{pt} = \mathsf{ct} \cdot \mathsf{sk}$ from the decoded message $\mathsf{CKKS.Dcd}(\mathsf{pt})$.

Homomorphic operations. CKKS supports approximate componentwise addition and multiplication, cyclic slot rotation, and complex conjugation. For example, addition and multiplication of ciphertexts encrypting $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$ decode approximately to $\boldsymbol{z}_1 + \boldsymbol{z}_2$ and $\boldsymbol{z}_1 \odot \boldsymbol{z}_2$, where $\odot$ is the Hadamard product. Rotation by i maps $(z_0, \ldots, z_{N/2-1})$ to $(z_i, z_{i+1}, \ldots, z_{i-1})$, with indices taken modulo $N/2$, while conjugation maps it to $(\bar{z}_0, \ldots, \bar{z}_{N/2-1})$. CKKS treats error as approximation error in the decoded message. Multiplication and rescaling consume ciphertext modulus, and bootstrapping restores a larger modulus while approximately preserving the message.

2.3 Discrete CKKS and Functional Bootstrapping

Discrete CKKS [20] uses CKKS to compute on bounded integer messages. A Discrete-CKKS ciphertext decrypts to an intended integer vector $\boldsymbol{z}$ plus an approximation error $\boldsymbol{e}$; when $\|\boldsymbol{e}\|_\infty < 1/2$, rounding recovers $\boldsymbol{z}$ exactly. This rounding interpretation gives exact discrete semantics in either coefficient or slot encoding. Addition and multiplication are inherited from CKKS, with error controlled by cleaning and bootstrapping.

Functional bootstrapping. CKKS-style functional bootstrapping [3, 4, 1] evaluates a function while refreshing the ciphertext. For a function $f : \mathbb{Z}_m \to \mathbb{Z}_m$, we write $\mathsf{LUT}_f(\mathsf{ct})$ for the output encrypting $f(\boldsymbol{z})$ componentwise. The modular-reduction variant of [29], denoted IntMod_m, accepts bounded integer messages and outputs their residues $[\boldsymbol{z}]_m$. Both operations include bootstrapping, restoring the modulus and refreshing the approximation error. For the small LUT moduli used here, their cost is comparable to CKKS bootstrapping.

Cleaning. Cleaning preserves the intended discrete message while reducing its approximation error. For example, $h_1(x) = 3x^2 - 2x^3$ from [18] fixes 0 and 1 and has vanishing first derivative at both points, so it cleans approximate bits. A general $\mathbb{Z}_b$ cleaning operation Clean_b can use polynomial evaluation with vanishing derivatives [20], functional bootstrapping [4, 1], or both.

Definition 1. *A cleaning function* Clean_b *with cleaning factor* (B, B') *preserves the intended* $\mathbb{Z}_b$*-valued plaintext and maps any input within its supported range whose underlying error satisfies* $\|e\|_\infty < B$ *to an output with error* $\|e'\|_\infty < B'$.

Coefficient/slot interface. The BFV conversion uses $\mathsf{CoeffIntBoot}_b$, coefficient-to-slot identity functional bootstrapping modulo b. It takes a coefficient-encoded input with integral coefficients modulo b and outputs slot ciphertexts encrypting those residues. When both components of the complex CKKS slots are used, we write the output as real-part and imaginary-part ciphertexts and apply subsequent digit operations to each independently. The linear transformation StC returns slot data to coefficient encoding. All invocations of these CKKS-side primitives assume the range, scale, and precision conditions of the cited Discrete-CKKS constructions.

2.4 Extraction and Ring Packing

Ring packing [9, 14, 2] converts a list of LWE ciphertexts into an RLWE ciphertext. We use it as a layout conversion together with coefficient extraction. If an RLWE ciphertext encrypts $a(X) = \sum_i a_i \cdot X^i$, then $\mathsf{Extract}_i(\mathsf{ct})$ outputs an LWE ciphertext encrypting a_i, and $\mathsf{Extract}(\mathsf{ct})$ denotes the list of extracted coefficient ciphertexts needed by the current algorithm. Conversely, $\mathsf{RingPack}$ takes a prescribed list of LWE ciphertexts and packs their messages into the coefficient or slot positions required by the construction.

2.5 Integer Computers from Discrete CKKS

Recent works [27, 6, 28, 12] build integer arithmetic and logical operations on top of Discrete CKKS. We use two radix-based interfaces, which differ in how they distribute digits across ciphertexts and slots.

Digit-separated integer computer. The backend of [27], denoted IntCom, stores a vector of integers in $\mathbb{Z}_{b^k}$ as k Discrete-CKKS ciphertexts, one per radix digit. A tuple $(\mathsf{ct}_0, \ldots, \mathsf{ct}_{k-1})$ encrypts digit vectors $(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{k-1})$ and represents $\boldsymbol{x} = \sum_{i=0}^{k-1} b^i \cdot \boldsymbol{x}_i \bmod b^k$ slotwise. The comparison operation $\mathsf{IntCom.Compare}$ maps two such encodings of $\boldsymbol{x}$ and $\boldsymbol{y}$ to an encoding of $\mathbf{1}[\boldsymbol{x} \geq \boldsymbol{y}]$. The Montgomery-style reduction in Section 3 uses the arithmetic operations of the same interface. The operation $\mathsf{IntCom.Mult}$ multiplies two k-digit inputs modulo b^k, while $\mathsf{IntCom.MultExt}$ outputs the full $2k$-digit product. The operations $\mathsf{IntCom.Low}$ and $\mathsf{IntCom.High}$ return the lower and upper k digits of a $2k$-digit value, respectively, and $\mathsf{IntCom.CondSub}(y, p)$ returns $y - p$ if $y \geq p$ and y otherwise. We construct $\mathsf{IntCom.REDC}_p$ from these operations in Section 3. The backend also supports addition modulo b^k and bitwise operations. The bootstrapping counts used later are $O(k)$ for addition and comparison and $O(k \log k)$ for multiplication and extended multiplication.

Packed integer computer. The backend of [12], denoted ParCom, packs radix digits into CKKS slots. When the number of represented integers exceeds one ciphertext's capacity, we use a tuple of packed ciphertexts. A length-L packed input with digit slots $(z_0, \ldots, z_{L-1})$ represents $\mathrm{val}_b(z_0, \ldots, z_{L-1})$ modulo the backend's supported range. The parameter L denotes the backend length, distinct from the GBFV plaintext-polynomial degree h and source digit length ℓ. Digits may be lazy: they can temporarily lie outside $[0, b)$ while still representing the same integer. The backend provides radix normalization to convert such lazy representations into canonical radix digits when required. Alongside packed addition and multiplication, we use radix normalization and the operation $\mathsf{ParCom.Compare}$, which outputs encrypted comparison bits. For fixed radix base and bounded input digits, these carry and comparison operations use $O(\log L)$ bootstrappings. We write $\mathsf{ParCom.REDC}_p$ for reduction to the ordinary canonical residue $[x]_p$, with the input range and instantiation specified in each construction. The generic Montgomery reduction of [12, Section 4.2] uses length $L = 2s$ with $R = b^s > p$, $\gcd(R, p) = 1$, and a canonical radix input $T \in [0, pR)$; it outputs $[TR^{-1}]_p$. For an ordinary input $x \in [0, R)$, we first multiply by $\rho = [R]_p$ and apply Montgomery reduction to $T = x\rho$, so the output is $[x]_p$ rather than a Montgomery residue. In this instantiation we choose $R > 2p$ so that the intermediate numerator is below $2pR < R^2$ and fits in $L = 2s$ digits; the algebra is the same as in Lemma 1, evaluated with packed operations. Thus its modulus restriction $p < b^{L/2}$ does not cover our choice $p = b^\ell + 1$ at length $L = 2\ell$. For GBFV, we instead instantiate $\mathsf{ParCom.REDC}_p$ by the specialized folding reduction of Lemma 2, built from the packed carry and signed-comparison primitives. It retains length $L = 2\ell$, placing the exposed ℓ digits in the low positions and reserving the remaining positions for reduction.

2.6 Prior GBFV/CKKS Conversion

The GBFV-to-CKKS conversion of [26], denoted $\mathsf{ConvGtC}$, exposes a bounded signed radix representation rather than canonical radix digits. For a message

$(m_0, \ldots, m_{h-1}) \in \mathbb{Z}_p^h$, its coefficient-encoded output approximates digits $m_{a,j}$ satisfying

$$m_a = \sum_{j=0}^{\ell-1} b^j \cdot m_{a,j} \pmod{p}.$$

In the parameter regime analyzed there, these digits satisfy $|m_{a,j}| \leq 3(b+1)/2$. In our conversion proofs, K denotes a public bound covering the relevant signed digits and lifting terms. The reverse direction uses $\mathsf{ConvCtG}$, the CKKS-to-GBFV conversion of [26]. Its input is a coefficient-encoded CKKS ciphertext containing, for each GBFV coordinate a, bounded signed digits $z_{a,0}, \ldots, z_{a,\ell-1}$. Its output is a GBFV ciphertext encrypting the vector whose a-th coordinate is

$$\sum_{j=0}^{\ell-1} b^j \cdot z_{a,j} \bmod p.$$

The same prior analysis shows that multiplying a modulus-raised GBFV ciphertext by $t(X) = X^h - b$ exposes bounded signed radix digits of this form; Section 4 makes this representation exact by packing those exposed digits and reducing them with $\mathsf{ParCom.REDC}_p$.

3 Conversion between BFV and Discrete CKKS

We construct conversions between BFV and the Discrete-CKKS integer representation of [27]. These conversions support BFV comparison and related logical operations with polylogarithmic bootstrapping counts.

Interface-level correctness. We use Discrete-CKKS functional bootstrapping, cleaning, slot/coefficient transformations, and the radix-integer operations of [27] as black boxes. Our correctness statements assume that each invocation meets the encoding, range, scale, and approximation-error requirements of these primitives. We construct the new routine $\mathsf{IntCom.REDC}_p$ from the arithmetic operations of [27] and prove the algebraic correctness of both the reduction and the surrounding conversions.

3.1 Discrete-CKKS Interface Primitives

We first recall the radix-decomposition interface and then construct arbitrary-modulus reduction in the integer computer.

Digit Decomposition. We adapt the iterative MSB bootstrapping of [29] so that it starts in coefficient encoding, ends in slot encoding, and outputs k ciphertext pairs rather than combining the final result into a single ciphertext. The resulting procedure, given in Algorithm 1, decomposes a Discrete-CKKS message encoded in the most significant bits.

Algorithm 1: $\mathsf{IntDecomp}_b^k$

Setting: $\Delta_0 = q_0/b^k$. $\mathsf{CoeffIntBoot}_b$ denotes an identity functional bootstrapping procedure from coefficient encoding to slot encoding for LUT modulus b. It outputs two ciphertexts, one for the real part and one for the imaginary part. On the subtraction path, StC includes public scale alignment to Δ_0 and reduction to modulus q_0.

Input : $\mathsf{ct} \in \mathcal{R}_{q_0}^2$, a coefficient-encoded CKKS ciphertext at scale Δ_0 encrypting a vector m of integers in $[0, b^k)$.

Output: $(\mathsf{ct}_0, \mathsf{ct}_1, \ldots, \mathsf{ct}_{k-1})$ encrypting the base-b digit decomposition of m.

$\mathsf{ct}' \leftarrow \mathsf{ct}$;
for $i \leftarrow 0$ **to** $k-1$ **do**
  $\mathsf{ct}_i = (\mathsf{ct}_i^R, \mathsf{ct}_i^I) \leftarrow \mathsf{CoeffIntBoot}_b(b^{k-1-i} \cdot \mathsf{ct}')$;
  if $i \neq k-1$ **then**
    $\mathsf{ct}' \leftarrow \mathsf{ct}' - b^i \cdot \mathsf{StC}(\mathsf{ct}_i^R + \sqrt{-1} \cdot \mathsf{ct}_i^I)$;
  end if
end for
return $(\mathsf{ct}_0, \mathsf{ct}_1, \ldots, \mathsf{ct}_{k-1})$

The digit ciphertexts retain the integer computer's output scale; only the copy returned to coefficient encoding is aligned to Δ_0 modulo q_0. This alignment acts on the ciphertext through public multiplication and modulus adjustment, and can be absorbed into StC. Consequently, before iteration i, the residual ciphertext represents $m - \sum_{j<i} b^j m_j$ at scale Δ_0, up to the accumulated approximation error. Under the black-box correctness of $\mathsf{CoeffIntBoot}_b$ and StC, Algorithm 1 maps a valid coefficient-encoded Discrete-CKKS encryption of $m \in [0, b^k)$ to valid encryptions of digits $m_0, \ldots, m_{k-1}$ satisfying $m = \sum_{i=0}^{k-1} b^i \cdot m_i$. The resulting ciphertext vector has the encoding structure of the integer computer from [27]. Each digit is represented by a pair $(\mathsf{ct}_i^R, \mathsf{ct}_i^I)$. We treat the real parts and imaginary parts as separate digit vectors and apply every subsequent integer-computer operation independently to each; the two evaluations can run in parallel.

General Modular Reduction. The integer computer of [27] provides the operations needed for Montgomery reduction: addition, multiplication, comparison, and bit shifts. We use them to construct homomorphic reduction modulo an arbitrary plaintext modulus p for inputs in base-b radix form. This follows the same high-level principle used by [12] in a different integer-computer framework. Algorithm 2 gives our construction in the interface of [27].

Multiplication by $\rho = [R]_p$ converts the ordinary input x into the Montgomery domain, while the subsequent REDC step multiplies by R^{-1} modulo p. The two effects cancel, so the output is $x \bmod p$ rather than a Montgomery residue. The following lemma establishes correctness and the bootstrapping count.

Algorithm 2: $\mathsf{IntCom.REDC}_p$

Setting: $R = b^k > 2p$ and $\gcd(p, b) = 1$. Let $\rho = [R]_p$ and $\eta = -p^{-1} \bmod R$. $\mathsf{IntCom.MultExt}$ denotes extended multiplication, outputting a $2k$-digit integer. $\mathsf{IntCom.Low}$ takes the low k digits modulo R, and $\mathsf{IntCom.High}$ takes the high k digits, i.e., division by R. $\mathsf{IntCom.CondSub}(\cdot, p)$ outputs $y - p$ if $y \geq p$ and y otherwise.

Input : A ciphertext vector $\mathbf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1, \ldots, \mathsf{ct}_{k-1})$ encrypting $x \in [0, R)$ according to the encoding from [27].

Output: A ciphertext vector $\mathbf{ct}_{\mathsf{out}} = (\mathsf{ct}'_0, \mathsf{ct}'_1, \ldots, \mathsf{ct}'_{k-1})$ encrypting $[x]_p$.

$\mathbf{ct}_T \leftarrow \mathsf{IntCom.MultExt}(\mathbf{ct}, \rho)$;
$\mathbf{ct}_n \leftarrow \mathsf{IntCom.Mult}(\mathsf{IntCom.Low}(\mathbf{ct}_T), \eta)$;
$\mathbf{ct}_y \leftarrow \mathsf{IntCom.High}(\mathbf{ct}_T + \mathsf{IntCom.MultExt}(\mathbf{ct}_n, p))$;
$\mathbf{ct}_{\mathsf{out}} \leftarrow \mathsf{IntCom.CondSub}(\mathbf{ct}_y, p)$;
return $\mathbf{ct}_{\mathsf{out}}$

Lemma 1 (General modular reduction). *Let $R = b^k > 2p$ and assume $\gcd(p, b) = 1$. Assume that the integer-computer operations in [27] are correct on the required input range and precision. If the input ciphertext vector is a valid radix encoding of $x \in [0, R)$, then Algorithm 2 outputs a valid radix encoding of $[x]_p$. The routine uses $O(k \log k)$ discrete bootstrappings.*

Proof. Let $T = x\rho$ and let $n = (T \bmod R)\eta \bmod R$, where $\rho = [R]_p$ and $\eta = -p^{-1} \bmod R$. Since $\eta p \equiv -1 \pmod R$, we have

$$T + np \equiv T - (T \bmod R) \equiv 0 \pmod R.$$

Thus $y = (T + np)/R$ is an integer, and this is exactly the value computed by the high-part operation in Algorithm 2. Moreover,

$$yR \equiv T \equiv xR \pmod p,$$

because $\rho \equiv R \pmod p$. Since R is invertible modulo p, it follows that $y \equiv x \pmod p$. Finally, $0 \leq T < Rp$ because $0 \leq x < R$ and $0 \leq \rho < p$, and $0 \leq n < R$, so $0 \leq y < 2p$. Therefore one conditional subtraction maps y to the canonical representative $[x]_p \in [0, p)$.

By the black-box correctness of the integer-computer operations, the homomorphic evaluation follows this plaintext computation as long as all intermediate values stay within the prescribed range and precision. The dominant operations are the extended multiplications, each costing $O(k \log k)$ discrete bootstrappings; the remaining low/high extraction, additions, and conditional subtraction are asymptotically cheaper. □

3.2 Conversions

BFV to Discrete CKKS. The MSB bootstrapping of [29] repeatedly applies identity functional bootstrapping to a coefficient-encoded Discrete-CKKS

ciphertext whose message occupies the most significant bits. This encoding coincides with BFV encoding, giving a direct conversion when the plaintext modulus is a radix power $p = b^k$. This procedure uses k executions of $\mathbb{Z}_b$ functional bootstrapping, in addition to the surrounding linear transformations.

For a general BFV plaintext modulus, the direct approach would require one iteration with LUT modulus p. Since functional bootstrapping with LUT modulus p costs $O(p)$, this is expensive for NTT-friendly primes such as $p = 2^{16} + 1$. We handle other moduli by combining modulus raising, radix decomposition, and homomorphic modular reduction in the integer computer of [27].

Consider a coefficient-encoded BFV ciphertext $\mathsf{ct} \in \mathcal{R}_q^2$ with plaintext modulus p, not necessarily a power of the radix base. We choose the radix base b so that $\gcd(p, b) = 1$, as required by the Montgomery-style reduction in Algorithm 2. For some message $m \in \mathcal{R}_p$ and noise $e \in \mathcal{R}$, it satisfies

$$\mathsf{ct} \cdot \mathsf{sk} = \lfloor (q/p) \cdot m \rceil + e.$$

After modulus raising, or equivalently embedding the ciphertext into $\mathcal{R}$, we obtain

$$\mathsf{ModRaise}(\mathsf{ct}) \cdot \mathsf{sk} = \lfloor (q/p) \cdot m \rceil + e + q \cdot I$$

for some integral polynomial I. Let $\Delta = q/p$. Then the modulus-raised ciphertext can be interpreted as a noisy Discrete-CKKS ciphertext encrypting $m + pI$:

$$\mathsf{ModRaise}(\mathsf{ct}) \cdot \mathsf{sk} = \lfloor \Delta \cdot (m + pI) \rceil + e.$$

Assume that the lifting polynomial I satisfies $\|I\|_\infty \leq K$ with overwhelming probability, and write the coefficients of m in their canonical representatives in $[0, p)$. Let $\mathbf{1}_\mathcal{R} = 1 + X + \cdots + X^{N-1}$ denote the all-one polynomial. Then each coefficient of $m + pI + pK\mathbf{1}_\mathcal{R}$ lies in $[0, p(2K + 1))$ with overwhelming probability. We choose k so that $b^k > \max\{p(2K + 1), 2p\}$, and we add the encoded public polynomial $pK\mathbf{1}_\mathcal{R}$ to the ciphertext. We then adjust the CKKS ciphertext modulus to an integer modulus q' chosen so that the induced scale q'/b^k is sufficiently close to Δ; for notational simplicity, we write $q' = \Delta \cdot b^k$. With this convention, define

$$\mathsf{ct}' = [\mathsf{ModRaise}(\mathsf{ct}) + \lfloor \Delta \cdot pK\mathbf{1}_\mathcal{R} \rceil]_{q'},$$

where the public polynomial is added to the first ciphertext component. Then ct' satisfies

$$\mathsf{ct}' \cdot \mathsf{sk} = \left\lfloor q' \cdot \frac{m + pI + pK\mathbf{1}_\mathcal{R}}{b^k} \right\rceil + e,$$

up to the usual CKKS approximation and rounding error caused by replacing Δ with q'/b^k. Thus ct' is a valid input to the digit-decomposition interface whenever the approximation error is within the correctness tolerance of the black-box bootstrapping primitive. Applying $\mathsf{IntDecomp}_b^k$ gives ciphertexts $\mathsf{ct}_0 = (\mathsf{ct}_0^R, \mathsf{ct}_0^I), \ldots, \mathsf{ct}_{k-1} = (\mathsf{ct}_{k-1}^R, \mathsf{ct}_{k-1}^I)$ that encrypt the base-b digits of

$m + pI + pK\mathbf{1}_{\mathcal{R}}$, up to the reordering imposed by the coefficient-to-slot conversion. Equivalently, after decryption and rounding, the decoded digits $\widehat{m}_i$ satisfy

$$\sum_{i=0}^{k-1} b^i \cdot \widehat{m}_i = m + pI + pK\mathbf{1}_{\mathcal{R}}.$$

Here $\widehat{m}_i$ is obtained from the pair $(\mathsf{ct}_i^R, \mathsf{ct}_i^I)$. Finally, homomorphic modular reduction by p, applied independently to the real and imaginary digit vectors, maps this radix representation of $m + pI + pK\mathbf{1}_{\mathcal{R}}$ to a radix representation of $m \bmod p$. Algorithm 3 summarizes the conversion.

Algorithm 3: ConvBtD: Conversion from BFV to Discrete CKKS

Setting: $\Delta = q/p$. The radix base b satisfies $\gcd(p, b) = 1$. The modulus q' is an integer ciphertext modulus chosen so that q'/b^k is sufficiently close to Δ; for notation, we write $q' = \Delta \cdot b^k$. The bracket $[\cdot]_{q'}$ denotes the required modulus adjustment to this CKKS modulus. The parameter K is chosen so that the lifting polynomial I from modulus raising satisfies $\|I\|_\infty \le K$ with overwhelming probability, and k is chosen so that $b^k > \max\{p(2K+1), 2p\}$.

Input : $\mathsf{ct} = \mathsf{BFV.Enc}(m) \in \mathcal{R}_q^2$, a BFV ciphertext encrypting $m \in \mathcal{R}_p$.

Output: $\mathbf{ct}_{\mathsf{out}} \in ((\mathcal{R}_Q^2)^2)^k$ encrypting m with respect to the radix encoding of [27].

1 $\mathsf{ct}' \leftarrow [\mathsf{ModRaise}(\mathsf{ct}) + \lfloor \Delta \cdot pK\mathbf{1}_{\mathcal{R}} \rceil]_{q'}$;

2 $\mathbf{ct} \leftarrow \mathsf{IntDecomp}_b^k(\mathsf{ct}')$;

3 $\mathbf{ct}_{\mathsf{out}} \leftarrow \mathsf{IntCom.REDC}_p(\mathbf{ct})$; `// applied independently to the real and imaginary digit vectors`

4 **return** $\mathbf{ct}_{\mathsf{out}}$

Theorem 1 (BFV-to-Discrete-CKKS conversion). *Under the lifting, range, and black-box precision conditions above, Algorithm 3 correctly maps a BFV encryption of $m \in \mathcal{R}_p$ to a valid Discrete-CKKS integer-computer encryption of the coefficientwise canonical representative of m. It uses $O(k \log k)$ discrete bootstrappings, hence $O(\log p \log\log p)$ when the lifting bound K is constant.*

Proof. The BFV ciphertext satisfies $\mathsf{ct} \cdot \mathsf{sk} = \lfloor (q/p) \cdot m \rceil + e$. After modulus raising, it satisfies

$$\mathsf{ModRaise}(\mathsf{ct}) \cdot \mathsf{sk} = \lfloor \Delta \cdot (m + pI) \rceil + e$$

for an integral lifting polynomial I. By the choice of K and the canonical representative of m, the shifted plaintext $u = m + pI + pK\mathbf{1}_{\mathcal{R}}$ lies coefficientwise in $[0, b^k)$ except with negligible probability. Hence the shifted ciphertext ct' in Algorithm 3 is a valid coefficient-encoded Discrete-CKKS encryption of u for the k-digit MSB-bootstrapping interface, under the stated precision assumptions.

The black-box correctness of $\mathsf{IntDecomp}_b^k$ gives a valid radix encryption of $u = \sum_{i=0}^{k-1} b^i u_i$ in the integer-computer encoding of [27]. Since

$$u = m + p(I + K\mathbf{1}_{\mathcal{R}}) \equiv m \pmod{p},$$

Lemma 1 shows that $\mathsf{IntCom.REDC}_p$ outputs a valid radix encryption of the coefficientwise canonical representatives of m modulo p.

The bootstrapping count consists of k executions of modulo-b discrete bootstrapping in $\mathsf{IntDecomp}_b^k$, followed by $\mathsf{IntCom.REDC}_p$. The latter costs $O(k \log k)$ discrete bootstrappings and dominates the $O(k)$ digit-decomposition count. Thus the total number of bootstrappings is $O(k \log k)$, which is $O(\log p \log\log p)$ when K is bounded independently of p. □

Discrete CKKS to BFV. A sufficiently clean coefficient-encoded Discrete-CKKS ciphertext at scale q/p has the same form as a BFV ciphertext. To convert back, we clean and recombine the radix digits, normalize the scale to q/p if necessary, and apply the slot-to-coefficient transformation, as shown in Algorithm 4. Scale normalization multiplies the encoded phase by the ratio of the target and current scales, preserving the decoded plaintext under the new scale, and can be absorbed into the surrounding linear transformations. The cleaning must leave the recombined error below the BFV decryption threshold.

Algorithm 4: $\mathsf{ConvDtB}$: Conversion from Discrete CKKS to BFV

Setting: The cleaned digits are aligned to a common scale Δ. If $\Delta \neq q/p$, the recombined ciphertext is multiplied by the public constant $(q/p)/\Delta$ before the final BFV interpretation. The slot-to-coefficient transformation preserves this target scale.

Input : $\mathbf{ct} = (\mathsf{ct}_0, \ldots, \mathsf{ct}_{k-1}) \in ((\mathcal{R}_Q^2)^2)^k$, a Discrete-CKKS ciphertext vector encrypting radix digits $m_0, \ldots, m_{k-1}$ according to the encoding in [27].

Output: $\mathsf{ct}_{\mathsf{out}} \in \mathcal{R}_q^2$ encrypting the class of $m = \sum_{i=0}^{k-1} b^i \cdot m_i$ in $\mathcal{R}_p$.

$\mathsf{ct}_{\mathsf{sum}} \leftarrow \mathsf{Clean}_b(\mathsf{ct}_0)$;
for $i \leftarrow 1$ **to** $k-1$ **do**
  $\mathsf{ct}_{\mathsf{sum}} \leftarrow \mathsf{ct}_{\mathsf{sum}} + b^i \cdot \mathsf{Clean}_b(\mathsf{ct}_i)$;
end for
if $\Delta \neq q/p$ **then**
  $\mathsf{ct}_{\mathsf{sum}} \leftarrow ((q/p)/\Delta) \cdot \mathsf{ct}_{\mathsf{sum}}$;
end if
$\mathsf{ct}_{\mathsf{out}} \leftarrow [\mathsf{StC}(\mathsf{ct}_{\mathsf{sum}}^R + \sqrt{-1} \cdot \mathsf{ct}_{\mathsf{sum}}^I)]_q$;
return $\mathsf{ct}_{\mathsf{out}}$

Theorem 2 (Discrete-CKKS-to-BFV conversion). *Let the input of Algorithm 4 be a valid radix encryption of digits $m_0, \ldots, m_{k-1}$. If cleaning, scale*

normalization, and slot-to-coefficient conversion leave the recombined error below the BFV decryption threshold, then the algorithm outputs a BFV encryption of the class of $\sum_{i=0}^{k-1} b^i m_i$ *in* $\mathcal{R}_p$.

Proof. The plaintext recombined by the algorithm is

$$\sum_{i=0}^{k-1} b^i \cdot m_i = m.$$

It remains to check that the recombined error is small enough for BFV decryption. Let B_i denote the residual decoded-message error after cleaning the i-th digit. After multiplication by b^i, this digit contributes at most $b^i B_i$ to the final decoded-message error, so the error before slot-to-coefficient conversion is bounded by

$$\sum_{i=0}^{k-1} b^i \cdot B_i.$$

The sufficient condition $B_i \leq B^\star b^{-i}$ gives the simpler bound $kB^\star$. The optional public multiplication aligns the encoded phase to scale q/p while leaving the decoded plaintext equal to m, up to its rounding error. If StC adds only a negligible amount of error and the resulting coefficient-encoded ciphertext at scale q/p has error below the BFV correctness threshold, then the output ciphertext decrypts to the class of the recombined value in $\mathcal{R}_p$. When the recombined integer is in $[0, p)$ coefficientwise, this is the canonical BFV plaintext. □

Together, the conversion theorems establish round-trip correctness.

Corollary 1 (Round-trip correctness). *Under the hypotheses of Theorems 1 and 2, the composition*

$$\mathsf{ConvDtB} \circ \mathsf{ConvBtD}$$

is a homomorphic identity on BFV plaintexts, up to the fresh noise introduced by the black-box primitives.

3.3 Applications to BFV

The radix interface supports digit and bit decomposition, comparison, and bitwise operations.

Decompositions. The conversion outputs base-b digits in the encoding of [27]. Converting the digit ciphertexts back individually, rather than recombining them, therefore gives a BFV digit decomposition. The MSB-bootstrapping techniques of [29] further decompose these digits into bits.

Algorithm 5: CompBFV: BFV Comparison from Discrete CKKS

Setting: Δ is the scale after cleaning and slot-to-coefficient conversion. The final multiplication by $(q/p)/\Delta$ normalizes the comparison bit to BFV scale. Cleaning is with respect to $\mathbb{Z}_2$ and is applied componentwise to a real/imaginary pair; StC recombines that pair as $\mathsf{ct}^R + \sqrt{-1}\mathsf{ct}^I$.

Input : $\mathsf{ct}_i = \mathsf{BFV.Enc}(m_i) \in \mathcal{R}_q^2$, BFV ciphertexts for $i = 1, 2$.

Output: $\mathsf{ct}_{\mathsf{out}} \in \mathcal{R}_q^2$ encrypting a comparison-bit vector.

$\mathbf{ct}_i \leftarrow \mathsf{ConvBtD}(\mathsf{ct}_i)$ for $i = 1, 2$;

$\mathsf{ct}' \leftarrow \mathsf{IntCom.Compare}(\mathbf{ct}_1, \mathbf{ct}_2)$;

$\mathsf{ct}_{\mathsf{out}} \leftarrow [((q/p)/\Delta) \cdot \mathsf{StC}(\mathsf{Clean}_2(\mathsf{ct}'))]_q$;

return $\mathsf{ct}_{\mathsf{out}}$

Comparison. We compare two BFV ciphertexts by converting both to the radix encoding of [27], applying integer-computer comparison, and converting the cleaned output bit back to BFV. The comparison is coefficientwise on canonical representatives in $[0, p)$: the output is 1 if the first value is at least the second, and 0 otherwise. Algorithm 5 gives the complete procedure.

Corollary 2 (BFV comparison). *When the conversions, integer-computer comparison, and final cleaning/re-encoding steps are correct, Algorithm 5 outputs a BFV encryption of the coefficientwise comparison bit* $\mathbf{1}[\bar{m}_1 \geq \bar{m}_2]$ *for canonical representatives in* $[0, p)$*. For fixed radix base and bounded lifting range, it uses* $O(\log p \log\log p)$ *bootstrappings.*

Proof. Correctness follows by composing the BFV-to-Discrete-CKKS conversion, the black-box correctness of the integer-computer comparison, and the final conversion of the comparison bit back to BFV. Each execution of ConvBtD uses $O(k \log k) = O(\log p \log\log p)$ bootstrappings. The comparison operation of [27] uses $O(k) = O(\log p)$ bootstrappings. If final cleaning requires only a constant number of additional bootstrappings, this count is dominated by the conversions. If one instead requires enough precision to support a fixed number t of subsequent multiplicative depths, then cleaning to precision $t \log p$ can be achieved with $O(\log\log p)$ iterations of a base cleaning polynomial such as $h_1(x) = 3x^2 - 2x^3$. Evaluating each fixed-degree iteration with at most a constant number of refresh bootstrappings adds $O(\log\log p)$ bootstrappings. In either case, the total number of bootstrappings remains $O(\log p \log\log p)$. □

Bitwise Operations. After bit decomposition, we evaluate operations such as AND, OR, and NAND componentwise on the encrypted bits in Discrete-CKKS slots. The resulting bits can then be cleaned and converted back to coefficient-encoded BFV. These bit operations consume one multiplicative level and are inexpensive relative to the conversion. The overall number of bootstrappings is $O(\log p \log\log p)$, since the conversion dominates this count.

4 Conversion between GBFV and Discrete CKKS

We extend the conversion framework to GBFV using the packed integer computer of [12], referred to as CPL25 in the experiments. The conversion of [26] produces a bounded redundant digit representation; we make this representation canonical by applying homomorphic modular reduction.

As in the BFV case, we treat extraction, ring packing, slot/coefficient transformations, the CKKS-to-GBFV conversion of [26], and the packed integer-computer operations of [12] as black boxes under their layout, range, and precision conditions. Our proofs establish the algebraic correspondence: multiplication by t exposes bounded radix digits, ring packing places them in the backend layout, and modular reduction maps the publicly shifted lazy representation to the canonical residue modulo p.

4.1 Exact Conversion

GBFV to Discrete CKKS. Let h be a power-of-two divisor of N, let $t(X) = X^h - b$, and set $\ell = N/h$ and $p = b^\ell + 1$. A GBFV plaintext has h coordinates over $\mathbb{Z}_p$, while ℓ is its source radix length. Because $p = b^\ell + 1$ exceeds b^ℓ, not every canonical residue in $[0, p)$ fits in ℓ ordinary base-b digits. We use the packed integer-computer interface of [12] with backend length

$$L = 2\ell.$$

This representation accommodates every canonical residue modulo p. The packed layout reserves L digit slots and L padding slots per value, giving $N/(4L) = N/(8\ell)$ values per ciphertext. We therefore require $\ell \le N/8$ for this same-ring layout; eight ciphertexts then accommodate all $h = N/\ell$ GBFV coordinates. We specialize the folding approach of [12, Section 4.2] using $b^\ell \equiv -1 \pmod{p}$, rather than invoking generic Montgomery reduction.

Lemma 2 (Packed reduction modulo $b^\ell + 1$). *Let $b \ge 2$, $\ell \ge 1$, and $p = b^\ell + 1$. Suppose a length-2ℓ packed ciphertext encrypts a bounded nonnegative lazy radix representation of $u \in [0, b^{2\ell})$. Assume the packed carry and signed-comparison primitives satisfy their range and precision conditions for the inputs and intermediate values below. Then the following reduction, denoted $\mathsf{ParCom.REDC}_p$, outputs the canonical length-2ℓ radix representation of $[u]_p$. For fixed b and a fixed bound on the input digits, it uses $O(\log L)$ bootstrappings, where $L = 2\ell$.*

Proof. Set $B = b^\ell$. First normalize the lazy digits and split the result into its low and high halves:

$$u = u_{\mathrm{lo}} + B u_{\mathrm{hi}}, \qquad 0 \le u_{\mathrm{lo}}, u_{\mathrm{hi}} < B.$$

Fold by subtracting the high-half digits from the low-half digits, obtaining $z = u_{\mathrm{lo}} - u_{\mathrm{hi}}$. Then $z \equiv u \pmod{p}$ and $-(B-1) \le z \le B - 1$. Each folded digit

lies in $[-(b-1), b-1]$, the signed carry input range. The signed carry procedure $\mathsf{LCtoCneg}$ of [12, Section 4.1], with its marker in position ℓ, recovers the canonical low digits of $w = [z]_B$ and the indicator $\varepsilon = \mathbf{1}[z \geq 0]$. We extract this indicator, clear position ℓ and higher in the value ciphertext, and set $\sigma = 1 - \varepsilon = \mathbf{1}[z < 0]$. Since $w = z + B\sigma$, adding σ to the lowest digit gives

$$r = w + \sigma = z + (B+1)\sigma = z + p\sigma \in [0, p).$$

Thus $r = [u]_p$. A final carry propagation makes its digits canonical, retaining any carry into position ℓ. In particular, $z = -1$ gives $r = B = p - 1$, whose digit in position ℓ is 1; this fits in the length-2ℓ representation. The initial normalization, signed carry, and final normalization each use $O(\log L)$ bootstrappings under the stated bounds; splitting, folding, and adding the encrypted sign bit are linear operations. □

Multiplying a modulus-raised GBFV ciphertext by t exposes bounded signed radix digits in its coefficients. These digits represent values $m_i + pJ_i$ for bounded integers J_i. Rather than propagating carries before reduction, we add a public redundant radix representation of a multiple of p to make the low digits nonnegative. Under the bounds below, the resulting lazy digit vector is a valid input to the specialized reduction of Lemma 2. Modular reduction determines the asymptotic bootstrapping count: for fixed radix base b and bounded lifting range, the conversion uses $O(\log \ell) = O(\log \log p)$ bootstrappings.

Algorithm 6 gives the conversion. To specify its public offset, define

$$\mathbf{p}^\star = (b+1, b-1, \ldots, b-1, 0, \ldots, 0) \in \mathbb{Z}^{2\ell},$$

where the first ℓ entries are $(b+1, b-1, \ldots, b-1)$ and the last ℓ entries are zero. Then $\mathrm{val}_b(\mathbf{p}^\star) = p = b^\ell + 1$. The exposed low digits are bounded below by $-(b+1)K$, so we set

$$A = \left\lceil \frac{(b+1)K}{b-1} \right\rceil.$$

We write $\mathsf{ParCom.REDC}_p^{+A\mathbf{p}^\star}$ for this specialized length-2ℓ reduction applied after adding $A\mathbf{p}^\star$ digitwise to the input digits.

The digitwise addition of $A\mathbf{p}^\star$ changes the represented integer by Ap and makes the low digits nonnegative without a separate exact-addition or carry primitive. The digits need not be canonical at this stage: their normalization is part of the reduction in Lemma 2, not part of the public offset addition.

Theorem 3 (GBFV-to-Discrete-CKKS conversion). *Under the conditions above, Algorithm 6 correctly maps a GBFV encryption of $m \in \mathbb{Z}_p^h$ to a valid packed Discrete-CKKS integer-computer encryption of the canonical representatives of m. For fixed b and bounded lifting range, it uses $O(\log \ell) = O(\log \log p)$ bootstrappings.*

Proof. Let ct be a GBFV ciphertext encrypting $m \in \mathbb{Z}_p^h$. After modulus raising and multiplication by the plaintext polynomial t, the ciphertext $\mathsf{ct}' = t \cdot$

Algorithm 6: $\mathsf{ConvGtD}$: Conversion from GBFV to the encoding of [12]

Setting: $L = 2\ell$ and $\ell \le N/8$. The permutation π orders LWE ciphertexts so that the packed ciphertext follows the length-L encoding structure of [12]. $\mathsf{RingPack}^*$ is a variant of $\mathsf{RingPack}$ that outputs eight real-value-encoded ciphertexts in this length-L packed format. The digit-length parameter satisfies $b^L > p(2A+1)$.

Input : $\mathsf{ct} = \mathsf{GBFV.Enc}(m)$ for $m \in \mathbb{Z}_p^h$.

Output: $\mathbf{ct}_{\mathsf{out}} \in (\mathcal{R}_Q^2)^8$, a tuple of [12] ciphertexts encrypting the exact length-2ℓ radix representation of the canonical representatives of m in their slots.

$\mathsf{ct}' \leftarrow t \cdot \mathsf{ModRaise}(\mathsf{ct})$;

$(\mathsf{ct}_0, \mathsf{ct}_1, \ldots, \mathsf{ct}_{N-1}) \leftarrow \pi \circ \mathsf{Extract}(\mathsf{ct}')$;

$\mathbf{ct}_{\mathsf{out}} \leftarrow \mathsf{RingPack}^*(\mathsf{ct}_0, \ldots, \mathsf{ct}_{N-1})$;

$\mathbf{ct}_{\mathsf{out}} \leftarrow \mathsf{ParCom.REDC}_p^{+A\mathbf{p}^\star}(\mathbf{ct}_{\mathsf{out}})$;

return $\mathbf{ct}_{\mathsf{out}}$

$\mathsf{ModRaise}(\mathsf{ct})$ can be interpreted as a Discrete-CKKS ciphertext whose coefficient plaintext is $m + tI$, up to the small error inherited from GBFV decryption and multiplication by t. This is precisely the bounded signed radix representation exposed by the GBFV-to-CKKS conversion of [26].

Write $m = (m_0, \ldots, m_{h-1})$. For each m_i, the exposed coefficients give bounded integers $m_{i0}, \ldots, m_{i(\ell-1)}$ satisfying

$$m_i = \sum_{j=0}^{\ell-1} b^j m_{ij} \pmod{p}.$$

Equivalently, there exists an integer J_i such that

$$m_i + pJ_i = \sum_{j=0}^{\ell-1} b^j m_{ij}.$$

By the assumed bound on the exposed plaintext digits, each m_{ij} satisfies

$$|m_{ij}| \le (b+1)K.$$

Since $0 \le m_i < p = b^\ell + 1$, the triangle inequality and the geometric sum give

$$|J_i| \le \frac{(b+1)K \sum_{j=0}^{\ell-1} b^j + m_i}{p} < 1 + \frac{(b+1)K}{b-1}.$$

Since $|J_i|$ is an integer, this strict bound implies

$$|J_i| \le \left\lceil \frac{(b+1)K}{b-1} \right\rceil = A.$$

The public vector

$$\mathbf{p}^\star = (b+1, b-1, \ldots, b-1, 0, \ldots, 0) \in \mathbb{Z}^{2\ell}$$

has value

$$\mathrm{val}_b(\mathbf{p}^\star) = (b+1) + \sum_{j=1}^{\ell-1}(b-1)b^j = b^\ell + 1 = p.$$

Adding $A\mathbf{p}^\star$ digitwise increases digit 0 by $A(b+1)$ and digits $1,\dots,\ell-1$ by $A(b-1)$. Thus all exposed low digits become nonnegative: the digit-0 offset is at least $(b+1)K$, and the other low-digit offsets are at least $(b+1)K$ by the definition of A. At the same time, this offset changes the represented integer by exactly Ap:

$$\mathrm{val}_b(\mathbf{m}_i + A\mathbf{p}^\star) = m_i + p(J_i + A),$$

where $\mathbf{m}_i = (m_{i0},\dots,m_{i(\ell-1)},0,\dots,0)$ is the packed low-norm digit vector before reduction.

Since $|J_i| \le A$, the represented integer is nonnegative; since $m_i \in [0,p)$ and $J_i \le A$, it lies below $p(2A+1)$. The condition $b^{2\ell} > p(2A+1)$ therefore puts the shifted value in $[0, b^{2\ell})$, as required by Lemma 2. The public offset represents a multiple of p, so it does not change the residue modulo p. By that lemma, under the stated carry and precision conditions, reducing $m_i + p(J_i + A)$ modulo p outputs the exact length-2ℓ radix representation of the canonical representative of m_i. This proves correctness for every packed coordinate.

For the bootstrapping count, multiplication by t, extraction, permutation, and the public offset require no bootstrapping. Coefficient-to-slot refreshing in the ring-packing interface contributes at most a constant number of bootstrappings for the fixed eight-ciphertext output layout. The specialized length-L reduction of Lemma 2 uses $O(\log L)$ bootstrappings, since its input digits are bounded in terms of the fixed b and K. Since $L = 2\ell$ and $\ell = \log_b(p-1)$, the total is $O(\log L) = O(\log \ell) = O(\log\log p)$ bootstrappings for fixed b. □

Discrete CKKS to GBFV. The CKKS-to-GBFV conversion $\mathsf{ConvCtG}$ uses public polynomial multiplication followed by rescaling [26, Theorem 7]. It takes a coefficient-encoded CKKS ciphertext containing a signed length-ℓ radix representation of each GBFV coordinate. For output modulus q, its input is supplied at modulus qq' with $q' \approx q$ and scale q. Writing z for the bounded signed radix polynomial, the input phase is $qz + e$, and the conversion is

$$\mathsf{ConvCtG}(\mathsf{ct}) = \mathsf{RS}_{q'}\big(\lfloor q'/t \rceil \cdot \mathsf{ct}\big) \in \mathcal{R}_q^2,$$

where $\mathsf{RS}_{q'}$ rescales by q'. The resulting phase is approximately $(q/t)z$, the GBFV encoding of the same residue modulo t. We include the public scale alignment in the preceding linear transformations and retain the modulus and precision required by this conversion.

Our forward conversion outputs a nonnegative canonical length-2ℓ representation, so we fold its upper half into its lower half using

$$b^\ell \equiv -1 \pmod{p}.$$

If $(d_0,\dots,d_{2\ell-1})$ represents a canonical residue modulo p, the signed vector $(d_0 - d_\ell,\dots,d_{\ell-1} - d_{2\ell-1})$ represents the same residue and has digits in $[-(b-1), b-1]$.

After StC, we extract the relevant low and high digit positions and subtract each high-half ciphertext from its low-half counterpart. We then pack the folded digits in the coefficient order expected by $\mathsf{ConvCtG}$. We denote this restricted low/high extraction by $\mathsf{Extract}_{\mathsf{rel}}$. Algorithm 7 gives the reverse conversion.

Algorithm 7: $\mathsf{ConvDtG}$: Conversion from the encoding of [12] to GBFV

Setting: $\mathsf{Extract}_{\mathsf{rel}}$ extracts the populated low and high digit positions after StC. For each plaintext coordinate a and digit index $0 \le j < \ell$, it outputs LWE ciphertexts $\mathsf{ct}^{\mathrm{lo}}_{a,j}$ and $\mathsf{ct}^{\mathrm{hi}}_{a,j}$ encrypting the digits in positions j and $j+\ell$, respectively. ψ orders the folded signed digits into the coefficient layout expected by $\mathsf{ConvCtG}$. $\mathsf{BaseRingPack}$ is the cheaper coefficient-to-coefficient variant of ring packing [2]; its output is aligned to modulus qq' and scale q, with $q' \approx q$, for $\mathsf{ConvCtG}$.

Input : $\mathbf{ct} \in (\mathcal{R}_Q^2)^8$, a tuple of [12] ciphertexts encrypting the exact length-2ℓ radix representation of $m \in \mathbb{Z}_p^h \subseteq \mathbb{Z}_{b^{2\ell}}^h$ in their slots.

Output: $\mathsf{ct}_{\mathsf{out}} \in \mathcal{R}_q^2$, a GBFV ciphertext encrypting m.

$\mathbf{ct}' \leftarrow \mathsf{StC}(\mathbf{ct})$;

$(\{\mathsf{ct}^{\mathrm{lo}}_{a,j}, \mathsf{ct}^{\mathrm{hi}}_{a,j}\}_{0\le a<h,\, 0\le j<\ell}) \leftarrow \mathsf{Extract}_{\mathsf{rel}}(\mathbf{ct}')$;

$\widetilde{\mathsf{ct}}_{a,j} \leftarrow \mathsf{ct}^{\mathrm{lo}}_{a,j} - \mathsf{ct}^{\mathrm{hi}}_{a,j}$ for all $0 \le a < h$ and $0 \le j < \ell$;

$\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{ConvCtG}(\mathsf{BaseRingPack}(\psi(\{\widetilde{\mathsf{ct}}_{a,j}\}_{a,j})))$;

return $\mathsf{ct}_{\mathsf{out}}$

Theorem 4 (Discrete-CKKS-to-GBFV conversion). *Assume the scale, modulus, precision, and digit-range conditions above for the black-box steps in Algorithm 7. Then the algorithm outputs a GBFV encryption of the same message m. It uses no bootstrapping under the stated black-box precision assumptions.*

Proof. Applying StC moves the digit values from slots to coefficients while preserving the encrypted message under the assumed precision condition. The populated coefficient positions contain length-2ℓ radix digits for the h GBFV plaintext coordinates, together with padding introduced by the packed integer-computer layout. The restricted extraction $\mathsf{Extract}_{\mathsf{rel}}$ extracts the low and high halves of these populated positions. If the extracted plaintext digits for one coordinate are $(d_0, \ldots, d_{2\ell-1})$, then

$$\sum_{j=0}^{2\ell-1} b^j d_j \equiv \sum_{j=0}^{\ell-1} b^j (d_j - d_{j+\ell}) \pmod{p},$$

because $b^\ell \equiv -1 \pmod{p}$. Hence the folded ciphertexts $\widetilde{\mathsf{ct}}_{a,j} = \mathsf{ct}^{\mathrm{lo}}_{a,j} - \mathsf{ct}^{\mathrm{hi}}_{a,j}$ encrypt a bounded signed length-ℓ radix representation of the same GBFV message coordinate. The permutation ψ reorders these folded LWE ciphertexts into the coefficient order expected by the CKKS-to-GBFV conversion of [26]. Since

the input to ConvCtG is coefficient-encoded, BaseRingPack assembles the LWE ciphertexts into a coefficient-encoded CKKS ciphertext at modulus qq' and scale q. By the correctness of ConvCtG on bounded signed radix inputs, the final ciphertext is a GBFV encryption of the same message m.

The slot-to-coefficient transformation, extraction, digit folding, base ring packing, and the public multiplication/rescaling step require no bootstrapping under the stated precision assumptions. □

Corollary 3 (Round-trip correctness). *Under the assumptions of Theorems 3 and 4, the composition*

$$\mathsf{ConvDtG} \circ \mathsf{ConvGtD}$$

is a homomorphic identity map for GBFV ciphertexts, up to the ordinary noise and modulus changes introduced by the black-box conversion procedures.

4.2 Applications

As in BFV, conversion exposes radix digits and makes the integer computer's logical operations available to GBFV.

Decompositions. The conversion places each base-b digit in a CKKS slot. Converting these slots back individually therefore gives a GBFV digit decomposition. The digits can further be decomposed into bits using the procedure of [29]. Because the encoding of [12] packs different digits into one ciphertext, bit decomposition can be performed in SIMD. The resulting order differs from the standard one; when needed, a reordering step can use public masks.

Logical Operations. Once two GBFV ciphertexts are converted into the encoding of [12], we can use its comparison operation, which requires $O(\log \ell)$ bootstrappings. After comparison, the comparison-bit ciphertext can be converted back to GBFV using ConvDtG. Algorithm 8 combines these steps.

Algorithm 8: CompGBFV: GBFV Comparison from Discrete CKKS

Setting: The conversions use the scaling and packing conventions of Algorithms 6 and 7.
Input : $\mathsf{ct}_i = \mathsf{GBFV.Enc}(m_i) \in \mathcal{R}_q^2$, GBFV ciphertexts for $i = 1, 2$.
Output: $\mathsf{ct}_{\mathsf{out}} \in \mathcal{R}_q^2$ encrypting a comparison-bit vector.

$\mathbf{ct}_i \leftarrow \mathsf{ConvGtD}(\mathsf{ct}_i)$ for $i = 1, 2$;
$\mathbf{ct} \leftarrow \mathsf{ParCom.Compare}(\mathbf{ct}_1, \mathbf{ct}_2)$;
$\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{ConvDtG}(\mathbf{ct})$;
return $\mathsf{ct}_{\mathsf{out}}$

Corollary 4 (GBFV comparison). *Under the correctness assumptions of the conversions and backend comparison, Algorithm 8 outputs a GBFV encryption of* $\mathbf{1}[\bar{m}_1 \geq \bar{m}_2]$ *for canonical representatives in* $[0, p)^h$*. For fixed* b *and bounded lifting range, it uses* $O(\log \ell) = O(\log \log p)$ *bootstrappings.*

Proof. Correctness follows by composing the two forward conversions, the backend comparison, and the reverse conversion. Each forward conversion uses $O(\log \ell)$ bootstrappings, the backend comparison uses $O(\log \ell)$, and the reverse conversion adds none. Thus the total number of bootstrappings is $O(\log \ell) = O(\log \log p)$. □

Bitwise Operations. Rather than applying GBFV operations separately to each of the $O(\log p)$ extracted bits, we evaluate bitwise operations directly on the Discrete-CKKS slots containing those bits. Each radix-b digit yields $\log_2 b = O(1)$ bit ciphertexts, and decomposition runs in SIMD across packed digits. Including conversion and homomorphic bit decomposition, the procedure uses $O(\log \ell) = O(\log \log p)$ bootstrappings for fixed b and bounded lifting range.

Fixed-Point Arithmetic. GBFV also offers a possible route to high-precision fixed-point arithmetic. One can encode a real number z as a scaled-and-rounded integer $m = \lfloor \Delta \cdot z \rceil \in \mathbb{Z}_p$ for a large modulus p. Multiplying two values at scale Δ produces scale Δ^2; at multiplicative depth d, the scale can grow to Δ^{2^d}. In BGV, BFV, and CKKS, such exponential growth is prohibitive, which is why CKKS rescales by Δ after each multiplication. In GBFV, however, the modulus can be as large as approximately b^N, allowing a few multiplicative levels even with exponential scaling-factor growth. For instance, if $p \approx \Delta^{2^d}$ for some d, then one can support roughly d multiplicative levels before the integer message exceeds the plaintext range.

This supports leveled computation, but GBFV alone does not reduce the scaling factor back to Δ. Our conversion suggests a rescaling-like operation: convert to the Discrete-CKKS encoding of [12], discard s low-order digits, shift the retained digits down by s positions, and convert back to GBFV. For a nonnegative integer representative, this computes $\lfloor m/b^s \rfloor$ and reduces the fixed-point scale by b^s. Public masks discard the low digits, while a slot permutation reindexes the retained digits to their new radix positions.

This application sketch illustrates how the digit interface could adjust the scale of an integer-encoded GBFV message; we do not analyze it as a formal primitive here. A full instantiation would need rounding conventions, scale management, and the exact error bound after digit removal. Compared with CKKS, it would sacrifice amortized throughput because many slots are used to store very large integers. It may be useful when exact integer semantics are important, including settings where approximate CKKS noise calls for measures such as noise flooding for IND-CPA-D [32] or threshold FHE [5].

5 Sparse Packing

Sparse coefficient packing further reduces the number of bootstrappings needed by the conversion framework. We present the balanced sparse setting: the payload positions are degrees $0, \ell, 2\ell, \ldots, (h-1)\ell$, where $h = N/\ell$. Intermediate non-payload coefficients may contain carry or lifting values; we extract only the payload positions before the final packing. This balanced choice yields $h\ell = N$ extracted LWE ciphertexts across all decomposition steps. A different stride s can be handled similarly, with $(N/s)\ell$ extracted ciphertexts and a correspondingly different packing density.

As before, correctness assumes the black-box guarantees of modulus switching and raising, coefficient extraction, ring packing, and the packed integer computer of [12]. The backend interprets a low-norm digit vector $(z_0, \ldots, z_{\ell-1})$ as a lazy residue $\sum_i b^i \cdot z_i$ modulo b^ℓ. The backend's radix normalization/carry reduction converts bounded nonnegative lazy digits into canonical radix digits; we use a public offset below to handle signed inputs. For sparse BFV, we use the generic Montgomery-based instantiation of $\mathsf{ParCom.REDC}_p$ from Section 2, including the premultiplication that makes its output an ordinary residue, in a length-2ℓ packed backend.

5.1 Sparse BFV Conversion

Let $\mathsf{ct} \in \mathcal{R}_q^2$ encrypt a sparse BFV plaintext $m \in \mathbb{Z}_p[X^\ell]/(X^N+1)$. Choose $q' = pr \approx q$, switch to modulus q', and modulus-raise. The lifted ciphertext satisfies $\mathsf{ct}' \cdot \mathsf{sk} = r(m+pI)+e$ for a bounded integral polynomial I. Choose b coprime to p and a public integer bound $K \geq 1$ with $\|I\|_\infty \leq K$ and $R = b^\ell > p(2K+1)$. After modulus raising, add the public polynomial $rpK\mathbf{1}_\mathcal{R}$ to the first component of ct', where $\mathbf{1}_\mathcal{R} = 1 + X + \cdots + X^{N-1}$. This replaces I by $I + K\mathbf{1}_\mathcal{R}$; reusing I for the shifted polynomial, we now have $0 \leq I \leq 2K$ coefficientwise. The public shift preserves every payload residue modulo p and requires no bootstrapping. Each payload coefficient of $m + pI$ then lies in $[0, b^\ell)$ and has an ℓ-digit base-b expansion; write $m + pI = \sum_{i=0}^{\ell-1} b^i \cdot m_i$ on the payload positions.

The decomposition repeatedly reduces modulo b. Let $\mathsf{ct}_0 = \mathsf{ModRaise}([\mathsf{ct}']_{br})$; it decrypts to $r(m_0 + bI_0) + e$ for a bounded I_0. Since $\mathsf{ct}' - \mathsf{ct}_0$ is divisible by b, division by b and iteration give ciphertexts encrypting

$$m_i' = m_i - I_{i-1} + bI_i, \qquad 0 \leq i < \ell,$$

with $I_{-1} = 0$ and $\|I_i\|_\infty \leq K$ for every decomposition lifting term. We extract only the sparse payload coefficients from each ct_i and ring-pack the resulting $h\ell = N$ LWE ciphertexts as the low ℓ digits of a length-2ℓ packed layout, with the high ℓ digit positions set to public zeros. Before normalization, we add the public vector $C\mathbf{r}^\star$ digitwise to the low ℓ positions, where

$$\mathbf{r}^\star = (b, b-1, \ldots, b-1) \in \mathbb{Z}^\ell, \qquad C = \left\lceil \frac{(b+1)K}{b-1} \right\rceil.$$

Since $\mathrm{val}_b(\mathbf{r}^\star) = b^\ell$ and $m'_i \geq -(b+1)K$, this offset makes every digit nonnegative while preserving the represented residue modulo b^ℓ. We normalize modulo $b^{2\ell}$, clear the high ℓ digit positions with a public plaintext mask, and invoke $\mathsf{ParCom.REDC}_p$ in the length-2ℓ backend.

Lemma 3 (Sparse radix decomposition). *Under the decomposition conditions above,*

$$\sum_{i=0}^{\ell-1} b^i \cdot m'_i \equiv m + pI \pmod{b^\ell}$$

on the payload positions. If $m + pI \in [0, b^\ell)$ coefficientwise, then after the public offset, backend radix normalization, and high-digit masking in the length-2ℓ layout, the represented payload values are exactly $m + pI$.

Proof. Using $m'_i = m_i - I_{i-1} + bI_i$ and $I_{-1} = 0$,

$$\sum_{i=0}^{\ell-1} b^i \cdot m'_i = \sum_{i=0}^{\ell-1} b^i \cdot m_i - \sum_{i=1}^{\ell-1} b^i \cdot I_{i-1} + \sum_{i=0}^{\ell-1} b^{i+1} \cdot I_i = m + pI + b^\ell I_{\ell-1}.$$

Reducing modulo b^ℓ gives the first claim. The public offset adds Cb^ℓ and makes the digits nonnegative, so normalization modulo $b^{2\ell}$ preserves this congruence. Since $m + pI \in [0, b^\ell)$, the low ℓ canonical digits after radix normalization are the digits of $m + pI$. Masking the high ℓ positions therefore leaves exactly the length-2ℓ representation of $m + pI$ with zero high digits. □

Theorem 5 (Sparse BFV conversion). *Under the conditions of Lemma 3, and assuming the black-box correctness of the packed backend, the sparse BFV protocol outputs packed Discrete-CKKS encryptions of the payload coefficients modulo p. For fixed b and bounded K, it uses $O(\log \ell)$ bootstrappings, in addition to $O(\ell)$ lightweight decomposition steps.*

Proof. The lemma shows that the public offset, backend normalization, and high-digit masking yield the exact value $m_u + pI_u$ for every payload position u. Write $x = m_u + pI_u \in [0, R)$, where $R = b^\ell > p(2K+1) > 2p$ and $\gcd(R, p) = 1$. The generic reduction first forms $T = x[R]_p < pR$ and then applies Montgomery reduction, returning $[TR^{-1}]_p = [x]_p = m_u$. Thus the length-$2\ell$ backend is used within its stated range and outputs the ordinary residue, not a Montgomery residue. The public offset adds no bootstrapping and leaves the digits bounded in terms of the fixed b and K. Packed radix normalization and arbitrary-modulus reduction each use a number of bootstrappings logarithmic in the packed length 2ℓ. The stated lightweight decomposition steps do not add bootstrappings. □

5.2 Sparse CKKS Comparison

The same decomposition applies to coefficient-encoded CKKS ciphertexts with sparse, cleaned integral plaintexts, without the BFV plaintext modulus. Let the

payload coefficients satisfy $0 \le m < b^\ell$ and write $m = \sum_{i=0}^{\ell-1} b^i \cdot m_i$. Repeated coefficient reduction modulo b and exact division give ciphertexts encrypting $m'_i = m_i - I_{i-1} + bI_i$ with $I_{-1} = 0$, so

$$\sum_{i=0}^{\ell-1} b^i \cdot m'_i \equiv m \pmod{b^\ell}.$$

Under the same bound $\|I_i\|_\infty \le K$, we add the same public offset $C\mathbf{r}^\star$ after sparse extraction and ring packing, normalize modulo b^ℓ, and compare the two resulting canonical inputs with ParCom.Compare. The two sparse constructions thus share the telescoping decomposition but use different backend lengths. Sparse CKKS comparison works directly modulo b^ℓ in a length-ℓ layout. Sparse BFV conversion instead uses a length-2ℓ layout: normalization moves any telescoping overflow into the high positions, which are cleared before reduction modulo p.

Theorem 6 (Sparse CKKS comparison). *For two sparse integral CKKS payloads m, m' with coefficients in $[0, b^\ell)$, the sparse comparison protocol outputs an encryption of the payload-position comparison vector $\mathbf{1}[m \ge m']$. It uses $O(\log \ell)$ bootstrappings, in addition to $O(\ell)$ lightweight decomposition steps.*

Proof. The telescoping identity gives each backend input as a low-norm lazy residue modulo b^ℓ, and the public offset makes its digits nonnegative without changing that residue. Because the payload representatives lie in $[0, b^\ell)$, backend radix normalization recovers the intended integers before comparison. Correctness follows from ParCom.Compare. Backend normalization and comparison use $O(\log \ell)$ bootstrappings, while coefficient-to-slot refreshing in the packing interface adds at most a constant number. The lightweight decomposition overhead is separate from this count. □

6 Experimental Evaluation

We evaluate proof-of-concept implementations of the conversions and their logical-operation applications. The implementations use radix base $b = 16$ and build on Lattigo's CKKS bootstrapping examples. Our experiments ran single-threaded on an Apple M4 Max processor with 128 GB of RAM running macOS Sequoia 15.7.9. The computation and encapsulation secrets are fixed-weight ternary with Hamming weights 192 and 32, respectively, and the error standard deviation is 3.2. All full-size parameter sets meet the 128-bit classical security target under the lattice estimator's default MATZOV cost model.[1] In the tables, $\log QP$ denotes the combined ciphertext and key-switching modulus size used for the experiment and its security estimate.

[1] https://github.com/malb/lattice-estimator, commit `53da5982597709ba0fdf94ea37a84d822310fd84`.

The implementations start from coefficient-encoded ciphertexts with the BFV or GBFV representation, and logical-operation experiments return coefficient-encoded ciphertexts in the same scheme. The BFV experiments use $N/2 = 32768$ message coefficients and leave the other half zero; the Discrete-CKKS backend operates on real-valued slots. Modulus raising uses sparse-secret encapsulation [8]; the Goldilocks input retains a two-prime base modulus for precision. Reported logical-operation times include both forward conversions where applicable, the backend operation, and the final return conversion, excluding key generation and decryption checks.

6.1 BFV-to-Discrete-CKKS Evaluation

Table 1 reports conversion and BFV comparison for two prime plaintext moduli: the NTT-friendly prime $p = 2^{16} + 1$ and the Goldilocks prime $p = 2^{64} - 2^{32} + 1$. The conversion rows measure the cost of obtaining a Discrete-CKKS radix representation; the comparison rows measure the end-to-end comparison of two BFV ciphertexts. For $p = 2^{16}+1$, conversion and comparison take 232 s and 555 s, respectively, over 32768 packed slots. For the 64-bit Goldilocks prime, they take 745 s and 1980 s. The increase in latency reflects the larger radix length and the corresponding increase in bootstrapping calls.

Table 1. Parameters and performance of the BFV-to-Discrete-CKKS implementation for radix base $b = 16$. Comparison is end-to-end over two BFV ciphertexts.

Plaintext modulus p	N	$\log QP$	Operation	Latency (s)	Bootstrappings
$2^{16} + 1$	2^{16}	1470	Conversion	232	32
			Comparison	555	77
$2^{64} - 2^{32} + 1$	2^{16}	1550	Conversion	745	92
			Comparison	1980	221

The comparison-bit return conversion takes 0.485 s for $p = 2^{16}+1$ and 0.580 s for the Goldilocks prime, with no additional bootstrapping.

Sign evaluation. Our sign-style operation returns the complement of a selected top bit of the canonical representative $m \in [0, p)$:

$$s_w(m) = 1 - \left(\lfloor m/2^{w-1} \rfloor \bmod 2\right).$$

The 13-bit experiment uses $p = 8179$ and $w = 13$, so its output is 1 exactly when $m < 4096$. For $p = 2^{16} + 1$, we use $w = 16$: the output is 1 when $m < 32768$ or $m = 65536$, and 0 otherwise. For the Goldilocks prime, we use $w = 64$, so the output is 1 exactly when $m < 2^{63}$. Table 2 compares these top-bit evaluations with the sign-evaluation results reported by Lee, Min, and Song [31] (LMS24). LMS24 uses BFV functional bootstrapping over an intermediate prime-power

plaintext space, whereas we convert to Discrete-CKKS radix form. In the power-of-two cyclotomic ring $\mathbb{Z}[X]/(X^N+1)$, SIMD packing over a small odd prime-power plaintext modulus depends on the factorization of X^N+1 modulo the base prime; equivalently, the slot count is determined by the order of that prime modulo $2N$. The LMS24 sign-evaluation parameters reported in [31] provide 16 active slots. Our implementation instead uses the CKKS slot capacity after conversion. Although our implementation has higher total latency, it supports larger batches and achieves lower amortized latency in this high-throughput setting.

Table 2. LMS24 sign evaluation [31] and our top-bit evaluation. Our 13-bit setting is compared with their largest reported setting; w denotes our top-bit window width.

Method	Setting	# slots	Latency (s)	Amortized latency (ms)
LMS24 [31]	9 bits	16	47.3	2960
	13 bits	16	172	10800
Ours	$w=13$, $p=8179$	32768	452	13.8
	$w=16$, $p=2^{16}+1$	32768	254	7.76
	$w=64$, Goldilocks	32768	762	23.3

6.2 GBFV-to-CPL25 Evaluation

We next evaluate the GBFV-to-CPL25 conversion and comparison algorithms for plaintext moduli $p=2^\lambda+1$, where $\lambda\in\{16,64,256,1024\}$. For base $b=16$, the GBFV digit parameter is $\ell=\lambda/4$, and the backend length used by the conversion is $L=2\ell=\lambda/2$. A fully packed conversion produces eight CPL25 ciphertexts. The "Batch/output" column gives the number of active values in each output ciphertext. The public redundant offset uses $A=19$ for $b=K=16$. Long signed-carry chains include leveled cubic symbol cleaning, which adds no bootstrapping. Forward ring packing uses degree 2^{12} at level 0; the comparison-bit return path uses degree 2^{13} at level 1 to retain its coefficient scale before the final GBFV modulus switch. The smaller-ring switching keys use uniform ternary secrets. Table 3 gives the parameter sets, and Table 4 summarizes the performance results.

Table 3. Parameter sets for the GBFV-to-CPL25 experiments [12]. For modulus $p=2^\lambda+1$ and base $b=16$, the conversion uses backend length $L=2\ell$. The fully packed conversion outputs eight CPL25 ciphertexts.

λ	ℓ	Batch/output	# CPL25 ct.	N	$\log QP$
16	4	2048	8	2^{16}	1520
64	16	512	8		1538
256	64	128	8		1548
1024	256	32	8		1544

Table 4. GBFV-to-CPL25 implementation performance for $p = 2^\lambda + 1$ and $b = 16$. Comparison includes two forward conversions, packed comparison, and conversion of the output back to GBFV. The last two columns count CKKS bootstrappings per packed output.

λ	ℓ	Batch/output	Conversion (s)	Comparison (s)	Conv. btp.	Comp. btp.
16	4	2048	573	1430	7	16
64	16	512	760	1780	8	18
256	64	128	878	2030	8	18
1024	256	32	1070	2400	10	22

As λ increases from 16 to 1024, conversion latency rises from 573 s to 1070 s, and comparison latency rises from 1430 s to 2400 s. The comparison pipeline is dominated by its two forward conversions. Packed comparison and the return path each add one bootstrapping per packed output.

Conversion breakdown. Table 5 breaks down Algorithm 6 into the input/packing interface, refresh, public-offset addition, and CPL25-side modular reduction. The input/packing column includes modulus raising, multiplication by the plaintext polynomial, scale alignment, extraction, and ring packing. Here REDC denotes the folding reduction of Lemma 2, including its carry-normalization steps. Modular reduction dominates the latency in every parameter set; the public-offset cost is negligible.

Table 5. Time breakdown for Algorithm 6, the GBFV-to-CPL25 conversion. The output is the clean CPL25-style packed radix representation of the GBFV plaintext. All timings are in seconds and rounded to three significant figures.

λ	Input/packing	Refresh	Public offset	REDC	Total	Bootstrap/output
16	29.1	53.6	0.241	490	573	7
64	29.2	62.9	0.277	667	760	8
256	29.2	70.8	0.367	778	878	8
1024	29.3	67.4	0.292	977	1070	10

For $\lambda = 16$, the conversion performs 7 bootstrappings per packed CPL25 output, corresponding to 56 sequential bootstrapping calls across the eight output ciphertexts. For $\lambda = 64$ and $\lambda = 256$, it performs 8 bootstrappings per output, or 64 sequential calls; for $\lambda = 1024$, it performs 10 per output, or 80 sequential calls. All four conversion experiments produced exact canonical CPL25 digits on the active support, with zero measured integer error.

Comparison breakdown. Table 6 reports the end-to-end GBFV comparison breakdown for Algorithm 8. The "ConvGtD, two inputs" column sums the conversion times for the two inputs. The remaining columns separate packed

comparison from the bit-specialized return conversion ConvDtG. The return-conversion timing includes one level-restoring bootstrap per packed output (eight in total), followed by bit cleaning, slot-to-coefficient conversion, coefficient repacking, and the final GBFV modulus switch.

Table 6. End-to-end GBFV comparison, including the return conversion ConvDtG. All timings are in seconds and rounded to three significant figures.

λ	ConvGtD, two inputs	CPL25 compare	ConvDtG	Total	Bootstrap/output
16	1170	84.5	176	1430	16
64	1550	96.9	137	1780	18
256	1780	115	133	2030	18
1024	2180	108	120	2400	22

The total is 16 bootstrappings per output for $\lambda = 16$, 18 for $\lambda \in \{64, 256\}$, and 22 for $\lambda = 1024$, corresponding to 128, 144, and 176 sequential evaluator calls across the eight outputs, respectively. All reported comparison experiments produced exact packed comparison bits and exact final GBFV outputs on the active support, with zero messages outside that support.

6.3 Partial-Output GBFV Conversion

Finally, we evaluate partial-output conversion at the largest GBFV parameter, $\lambda = 1024$. When an application occupies only part of the packing capacity, the implementation skips unused output blocks rather than generating all eight CPL25 ciphertexts. This optimization applies to GBFV inputs and is distinct from the sparse BFV/CKKS constructions in Section 5. Table 7 shows that the conversion time scales almost linearly with the number of generated CPL25 ciphertexts: active-output densities 1/8, 1/4, and 1/2 reduce latency by 8.02×, 4.00×, and 2.00×, respectively.

Table 7. Partial-output GBFV-to-CPL25 conversion latency for $\lambda = 1024$.

Active-output density	CPL25 ciphertexts	Conversion time (s)	Latency reduction
1/8	1	134	8.02×
1/4	2	268	4.00×
1/2	4	538	2.00×
1	8	1070	1.00×

Acknowledgements

This work was partially supported by a grant from UBRI, and the author was supported by the Stanford Graduate Fellowship.

AI tool disclosure. The author used OpenAI ChatGPT and OpenAI Codex for writing, editing, technical analysis, and implementation assistance; details appear in Appendix A.

References

1. Alexandru, A., Kim, A., Polyakov, Y.: General functional bootstrapping using CKKS. In: Kalai, Y.T., Kamara, S.F. (eds.) CRYPTO 2025, Part III. LNCS, vol. 16002, pp. 304–337. Springer, Cham (Aug 2025). https://doi.org/10.1007/978-3-032-01881-6_10
2. Bae, Y., Cheon, J.H., Kim, J., Park, J.H., Stehlé, D.: HERMES: Efficient ring packing using MLWE ciphertexts and application to transciphering. In: Handschuh, H., Lysyanskaya, A. (eds.) CRYPTO 2023, Part IV. LNCS, vol. 14084, pp. 37–69. Springer, Cham (Aug 2023). https://doi.org/10.1007/978-3-031-38551-3_2
3. Bae, Y., Cheon, J.H., Kim, J., Stehlé, D.: Bootstrapping bits with CKKS. In: Joye, M., Leander, G. (eds.) EUROCRYPT 2024, Part II. LNCS, vol. 14652, pp. 94–123. Springer, Cham (May 2024). https://doi.org/10.1007/978-3-031-58723-8_4
4. Bae, Y., Kim, J., Stehlé, D., Suvanto, E.: Bootstrapping small integers with CKKS. In: Chung, K.M., Sasaki, Y. (eds.) ASIACRYPT 2024, Part I. LNCS, vol. 15484, pp. 330–360. Springer, Singapore (Dec 2024). https://doi.org/10.1007/978-981-96-0875-1_11
5. Boneh, D., Gennaro, R., Goldfeder, S., Jain, A., Kim, S., Rasmussen, P.M.R., Sahai, A.: Threshold cryptosystems from threshold fully homomorphic encryption. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 565–596. Springer, Cham (Aug 2018). https://doi.org/10.1007/978-3-319-96884-1_19
6. Boneh, D., Kim, J.: Homomorphic encryption for large integers from nested residue number systems. In: Kalai, Y.T., Kamara, S.F. (eds.) CRYPTO 2025, Part III. LNCS, vol. 16002, pp. 338–370. Springer, Cham (Aug 2025). https://doi.org/10.1007/978-3-032-01881-6_11
7. Boneh, D., Kim, J.: A universal blinder: One-round blind signatures from FHE. Cryptology ePrint Archive, Paper 2026/574 (2026), https://eprint.iacr.org/2026/574
8. Bossuat, J.P., Troncoso-Pastoriza, J.R., Hubaux, J.P.: Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation. In: Ateniese, G., Venturi, D. (eds.) ACNS 2022. LNCS, vol. 13269, pp. 521–541. Springer, Cham (Jun 2022). https://doi.org/10.1007/978-3-031-09234-3_26
9. Boura, C., Gama, N., Georgieva, M., Jetchev, D.: Chimera: Combining ring-lwe-based fully homomorphic encryption schemes. Journal of Mathematical Cryptology **14**(1), 316–338 (2020). https://doi.org/10.1515/jmc-2019-0026
10. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical GapSVP. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 868–886. Springer, Berlin, Heidelberg (Aug 2012). https://doi.org/10.1007/978-3-642-32009-5_50
11. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed.) ITCS 2012. pp. 309–325. ACM (Jan 2012). https://doi.org/10.1145/2090236.2090262

12. Cha, G., Park, D., Lee, J.W.: Improved radix-based approximate homomorphic encryption for large integers via lightweight bootstrapped digit carry. In: Daemen, J., Thomé, E. (eds.) EUROCRYPT 2026, Part IV. LNCS, vol. 16545, pp. 243–273. Springer, Cham (May 2026). https://doi.org/10.1007/978-3-032-25327-9_9
13. Cha, H., Hwang, I., Min, S., Seo, J., Song, Y.: MatriGear: Accelerating authenticated matrix triple generation with scalable prime fields via optimized HE packing. In: Blanton, M., Enck, W., Nita-Rotaru, C. (eds.) 2025 IEEE Symposium on Security and Privacy. pp. 2453–2471. IEEE Computer Society Press (May 2025). https://doi.org/10.1109/SP61157.2025.00174
14. Chen, H., Dai, W., Kim, M., Song, Y.: Efficient homomorphic conversion between (ring) LWE ciphertexts. In: Sako, K., Tippenhauer, N.O. (eds.) ACNS 2021, Part I. LNCS, vol. 12726, pp. 460–479. Springer, Cham (Jun 2021). https://doi.org/10.1007/978-3-030-78372-3_18
15. Chen, H., Laine, K., Player, R., Xia, Y.: High-precision arithmetic in homomorphic encryption. In: Smart, N.P. (ed.) CT-RSA 2018. LNCS, vol. 10808, pp. 116–136. Springer, Cham (Apr 2018). https://doi.org/10.1007/978-3-319-76953-0_7
16. Chen, H., Laine, K., Rindal, P.: Fast private set intersection from homomorphic encryption. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 1243–1255. ACM Press (Oct / Nov 2017). https://doi.org/10.1145/3133956.3134061
17. Cheon, J.H., Kim, A., Kim, M., Song, Y.S.: Homomorphic encryption for arithmetic of approximate numbers. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017, Part I. LNCS, vol. 10624, pp. 409–437. Springer, Cham (Dec 2017). https://doi.org/10.1007/978-3-319-70694-8_15
18. Cheon, J.H., Kim, D., Kim, D.: Efficient homomorphic comparison methods with optimal complexity. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020, Part II. LNCS, vol. 12492, pp. 221–256. Springer, Cham (Dec 2020). https://doi.org/10.1007/978-3-030-64834-3_8
19. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016, Part I. LNCS, vol. 10031, pp. 3–33. Springer, Berlin, Heidelberg (Dec 2016). https://doi.org/10.1007/978-3-662-53887-6_1
20. Drucker, N., Moshkowich, G., Pelleg, T., Shaul, H.: BLEACH: Cleaning errors in discrete computations over CKKS. Journal of Cryptology **37**(1), 3 (Jan 2024). https://doi.org/10.1007/s00145-023-09483-1
21. Ducas, L., Micciancio, D.: FHEW: Bootstrapping homomorphic encryption in less than a second. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part I. LNCS, vol. 9056, pp. 617–640. Springer, Berlin, Heidelberg (Apr 2015). https://doi.org/10.1007/978-3-662-46800-5_24
22. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144 (2012), https://eprint.iacr.org/2012/144
23. Geelen, R., Vercauteren, F.: Fully homomorphic encryption for cyclotomic prime moduli. In: Fehr, S., Fouque, P.A. (eds.) EUROCRYPT 2025, Part III. LNCS, vol. 15603, pp. 366–397. Springer, Cham (May 2025). https://doi.org/10.1007/978-3-031-91131-6_13
24. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) 41st ACM STOC. pp. 169–178. ACM Press (May / Jun 2009). https://doi.org/10.1145/1536414.1536440
25. Iliashenko, I., Zucca, V.: Faster homomorphic comparison operations for BGV and BFV. PoPETs **2021**(3), 246–264 (Jul 2021). https://doi.org/10.2478/popets-2021-0046

26. Kim, J.: Bootstrapping GBFV with CKKS. Cryptology ePrint Archive, Report 2025/888 (2025), https://eprint.iacr.org/2025/888
27. Kim, J.: Efficient homomorphic integer computer from CKKS. IACR TCHES **2025**(4), 873–898 (2025). https://doi.org/10.46586/tches.v2025.i4.873-898
28. Kim, J.: Faster homomorphic integer computer. Cryptology ePrint Archive, Report 2025/1440 (2025), https://eprint.iacr.org/2025/1440
29. Kim, J., Noh, T.: Modular reduction in CKKS. CiC **2**(2), 17 (2025). https://doi.org/10.62056/aevur-iuc
30. Kim, J., Seo, J., Song, Y.: Simpler and faster BFV bootstrapping for arbitrary plaintext modulus from CKKS. In: Luo, B., Liao, X., Xu, J., Kirda, E., Lie, D. (eds.) ACM CCS 2024. pp. 2535–2546. ACM Press (Oct 2024). https://doi.org/10.1145/3658644.3670302
31. Lee, D., Min, S., Song, Y.: Functional bootstrapping for packed ciphertexts via homomorphic LUT evaluation. Cryptology ePrint Archive, Report 2024/181 (2024), https://eprint.iacr.org/2024/181
32. Li, B., Micciancio, D.: On the security of homomorphic encryption on approximate numbers. In: Canteaut, A., Standaert, F.X. (eds.) EUROCRYPT 2021, Part I. LNCS, vol. 12696, pp. 648–677. Springer, Cham (Oct 2021). https://doi.org/10.1007/978-3-030-77870-5_23
33. Lu, W.j., Huang, Z., Hong, C., Ma, Y., Qu, H.: PEGASUS: Bridging polynomial and non-polynomial evaluations in homomorphic encryption. In: 2021 IEEE Symposium on Security and Privacy. pp. 1057–1073. IEEE Computer Society Press (May 2021). https://doi.org/10.1109/SP40001.2021.00043
34. Mughees, M.H., Chen, H., Ren, L.: OnionPIR: Response efficient single-server PIR. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021. pp. 2292–2306. ACM Press (Nov 2021). https://doi.org/10.1145/3460120.3485381
35. Park, J.H., Passelègue, A., Stehlé, D.: THED: Threshold dilithium from FHE. Cryptology ePrint Archive, Paper 2026/638 (2026), https://eprint.iacr.org/2026/638
36. Tan, B.H.M., Lee, H.T., Wang, H., Ren, S., Aung, K.M.M.: Efficient private comparison queries over encrypted databases using fully homomorphic encryption with finite fields. IEEE Transactions on Dependable and Secure Computing **18**(6), 2861–2874 (2021). https://doi.org/10.1109/TDSC.2020.2967740
37. Zhang, X., Wang, R., Liu, Z., Xiang, B., Deng, Y., Fisch, B., Lu, X.: Phalanx: An FHE-friendly SNARK for verifiable computation on encrypted data. In: Huang, C.Y., Chen, J.C., Shieh, S.P., Lie, D., Cortier, V. (eds.) ACM CCS 2025. pp. 4035–4048. ACM Press (Oct 2025). https://doi.org/10.1145/3719027.3765226
38. Zhang, Y., Chen, X., Lou, Q.: Hebridge: Connecting arithmetic and logic operations in fv-style he schemes. In: Proceedings of the 12th Workshop on Encrypted Computing & Applied Homomorphic Cryptography. p. 23–35. WAHC '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3689945.3694807

A Disclosure of Generative AI Assistance

The author used OpenAI ChatGPT and OpenAI Codex for writing, editing, technical analysis, and implementation assistance in preparing this manuscript and its accompanying artifact. The work was developed from the author's research ideas, constructions, manuscript text, and guidance.

Writing and editorial assistance included grammar and style editing, improvements to clarity and organization, refinement of contribution and related-work discussions, and polishing of theorem and proof presentation. The tools also helped draft or revise experimental descriptions, disclosure text, and artifact documentation; check notation and cross-references; and prepare the public ePrint revision, including compilation and layout checks.

Technical assistance included cross-checking the cited literature and implementation, identifying and analyzing potential correctness and consistency issues, and helping revise technical explanations and arguments under the author's direction. This covered conversion interfaces, complexity statements and bootstrapping counts, numerical precision and level management, security-estimation assumptions and cost models, and interpretation of experimental results.

Implementation assistance included drafting and editing proof-of-concept code for REDC, ConvBtD, ConvGtD, and return conversions; refactoring and debugging; checking correspondence with the paper's algorithms; organizing experimental workflows; designing and running validation tests; and checking cryptographic parameters and evaluation-key configurations. The tools also helped run resource-bounded experiments and lattice-estimator analyses and organize the resulting measurements and documentation.

Reported measurements come from executions of the implementation and security-estimation scripts under the author's direction. The author reviews and edits AI-assisted text and code, rewriting where needed, and checks the results before inclusion. The author is responsible for the correctness, appropriateness, originality, and integrity of all material in this work and its accompanying artifact.