Homomorphic Encryption for Large Integers from Nested Residue Number Systems

Dan Boneh and Jaehyung Kim

Stanford University

February 25, 2025

Abstract

Existing fully homomorphic encryption (FHE) schemes primarily support a plaintext space defined over a relatively small prime. However, in some important applications of FHE one needs arithmetic over a large prescribed prime. In this paper we construct a new FHE system that is specifically designed for this purpose. Our system composes three layers of residue systems to enable much better performance than was previously possible. Our experiments show that for arithmetic modulo a 256-bit integer, when compared to the TFHE-rs implementation of 256-bit arithmetic, our new system achieves a factor of a thousand better multiplication throughput and a factor of ten better latency. Moreover, for a 2048-bit prime modulus we achieve far better performance than was previously possible.

Contents

1 Introduction

Fully homomorphic encryption (FHE) [Gen09] schemes enable arithmetic operations over encrypted data. The supported arithmetic operations are typically defined modulo relatively small primes. However, in several important applications of FHE there is a need for arithmetic modulo a prescribed *large* integer. We review two examples informally:

- A **universal thresholdizer** [BGG⁺18, EY24] uses threshold FHE to compile any (deterministic) signature scheme[1] into a one-round threshold signature scheme. The high level idea is to first homomorphically run the (non-threshold) signing algorithm on the encrypted signing key and the cleartext message m to obtain an encrypted signature on m. Then have the threshold signing parties use FHE threshold decryption to release the cleartext signature on m. This gives a one-round threshold signature scheme where the final signature is a standard signature for the non-threshold scheme.

- A **universal blinder** uses FHE to compile any (deterministic) signature scheme into a one-round blind signature scheme [Cha82]. Here the client sends an encrypted message m to the signer. The signer runs the signing algorithm homomorphically on the cleartext secret key and the encrypted message m, and sends the resulting encrypted signature back to the client. The client decrypts and obtains the desired signature on m. This gives a one-round blind signature scheme where the final signature is a standard signature for the non-blind scheme. Note that the signer must send back a *verifiable* FHE ciphertext [KVMH24, ABPS24] to convince the client that it ran the signing algorithm correctly[2]. Otherwise, the signer may be able to link the request to a subsequent published signature.

In both cases the signing algorithm is run homomorphically, and we therefore refer to these constructions as *homomorphic signing.*

While these constructions may seem to be only of theoretical interest, they become quite interesting when applied to widely used signature schemes such as ECDSA and Schnorr [BS23, Ch. 19]. Current constructions for threshold and blind variants of Schnorr [KG20, Lin22, FW24] require two or more rounds. Threshold and blind variants of ECDSA are even more complicated [DKLS24, QCY21]. The universal thresholdizer and universal blinder are the only known constructions for a one-round threshold and a one-round blind variants of ECDSA and Schnorr signatures, where the final signature is backwards compatible with an ECDSA or Schnorr verifier. While a universal thresholdizer [AAB⁺24] and universal blinder [Fis06] can be built from general (succinct) non-interactive zero-knowledge, the resulting signatures are very different from a signature in the underlying scheme.

The ECDSA and Schnorr signature schemes are used in practice over an elliptic curve group defined over a 256-bit or a 384-bit prime. Consequently, running the signing algorithms inside an FHE requires support for arithmetic modulo a prescribed 256-bit or 384-bit large prime.

The universal thresholdizer and blinder are also useful when applied to other signature schemes. For example, Gennaro, Halevi, and Rabin [GHR99] designed a signature scheme in the standard model that operates in RSA-like groups. That scheme is not easily made threshold or blind because the signature is not a linear function of the secret key (as in the RSA signature scheme). However, the universal thresholdizer and blinder can nevertheless compile this scheme into a one-round threshold and blind scheme. Here,

[1] The signing algorithm of any signature scheme can be made deterministic by derandomizing it using a secure PRF applied to (pk, m). The PRF secret key becomes part of the secret signing key [BS23, Ex. 13.6].

[2] If a PRF is used to derandomize the signing algorithm, then the client will be given a certified commitment to the PRF secret key, to be used in FHE verification.

running the signing algorithm homomorphically requires arithmetic modulo a prescribed 2048-bit integer.

Current FHE schemes perform poorly when required to support arithmetic modulo such a large 256-bit, 384-bit, or 2048-bit prescribed integer. For example, for ECDSA homomorphic signing, the reported running time[3] for generating a single signature over Secp256k1 using Zama's TFHE is two days on a 64 cores machine.

1.1 Our results

We construct a new FHE system specifically designed for fast arithmetic operations modulo a prescribed large integer. The key innovation is the ability to very efficiently support a different plaintext modulus in every slot of a *single* RLWE ciphertext. This lets us use a Chinese Remainder (CRT) representation to support a very large *smooth* plaintext modulus in a single ciphertext. We then leverage this to support a large *prescribed* (non-smooth) plaintext modulus in a single ciphertext. Our scheme builds upon a recent work by Kim and Noh [KN24] on modular reductions in CKKS. We generalize that construction to support a different plaintext modulus in every slot.

Large Plaintext Modulus. A ciphertext in our FHE scheme is an RLWE vector of a certain dimension N. We use nested CRT and homomorphic base conversion, as explained below, to support a plaintext modulus as large as $O(N)$ bits. For example, using the standard $\log N = 16$ ring dimension we achieve up to $30,000$ bits of smooth plaintext modulus in a single ciphertext, with an efficiency that was not possible before. This enables us to support up to $10,000$ bits of a prescribed (non-smooth) modulus. We can similarly support multiple slots of a smaller modulus, such as a 2048-bit RSA modulus or the popular prime $2^{255} - 19$ used to define the elliptic curve 25519 [Ber06].

Flexible Choice of Primes. One of the difficulties in using BGV/BFV is that one needs to carefully choose the plaintext space for efficiency. This stems from the fact that the slot structure in BGV/BFV requires certain special properties to support the Number Theoretic Transform (NTT). For example, to support full slots (i.e. N slots for a ring of degree N), one must use a prime p such that $2N$ divides $p - 1$, or a product of such primes. However, N is typically as large as 2^{16}, and this severely limits the choice of prime modulus that defines the plaintext space. In our new FHE scheme, there is more flexibility in the choice of prime modulus because we rely on the Discrete Fourier Transform (DFT) in characteristic zero (over $\mathbb{C}$), instead of relying on NTT.

Universal Parameter Set. The FHE scheme enables a single ciphertext to support a vector of plaintexts, where each plaintext space is defined over a different large prescribed modulus.

Performance. We report on our performance experiments in Section 6. A summary is shown in Table 1, which shows the latency and throughput of our FHE for 255-bit, 384-bit, and 2048-bit moduli. These running times are all single threaded on a Mac. We expect that a multi-threaded implementation will be more than ten times faster.

It is also illustrative to compare the running time of this new FHE to the performance of Zama's TFHE library for 256-bit arithmetic. We show in Section 6 that the throughput is about a thousand times higher. More importantly, the latency for a single 256-bit multiplication is about ten times lower. For ECDSA homomorphic signing, this means that one can expect that the time to generate a single signature will be approximately

[3] https://github.com/zama-ai/bounty-ecdsa-signature/blob/master/tutorial.md

ten times lower than in previous experiments. When generating a batch of signatures, the amortized time per signature should be about a thousand times lower.

$\log(s)$	# slots	$\mathbb{Z}_s$ multiply time	
		latency	amortized time
255	32	254 sec	7.94 sec
384	32	259 sec	8.09 sec
2048	4	330 sec	82.5 sec

Table 1: Timing for a modular homomorphic multiplication over $\mathbb{Z}_s$. The number of slots refers to the number of $\mathbb{Z}_s$ elements that fit in a single ciphertext. The 255 bit prime used is the elliptic curve prime $2^{255} - 19$. The 384 bit prime used is the P-384 prime for elliptic curves. The 2048 bit row refers to a modulus of that size. The times were measured on a single threaded Mac with 32GB of RAM.

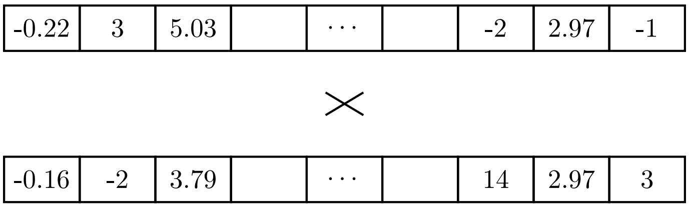
1.2 Technical Overview

Our FHE scheme is built on top of a homomorphic computer that supports two data types $\mathbb{Z}$ and $\mathbb{R}$. The computer supports addition and multiplication over each data type, possibly type casting between them as necessary. The computer also supports modular reduction by a small integer t. These computations can be managed in parallel in a single RLWE ciphertext of dimension N with $N/2$ slots. For example, in Figure 1 we are given two vectors filled with elements in either $\mathbb{Z}$ or $\mathbb{R}$, and we can perform element-wise operations on them. In Section 2 we explain how such a homomorphic computer can be instantiated using the modular reduction framework of Kim and Noh [KN24]. In this section we treat this homomorphic computer as a black box.

Our construction proceeds in four layers, where each layer builds on the one before it. We briefly describe each layer in turn. The details are provided in Sections 3 and 4.

Homomorphic Residue Number System (RNS). In Section 3.1 we generalize the modular reduction method from [KN24] by showing that modular reduction can be taken with respect to different moduli in different slots. That is, in one slot we can perform a mod p reduction, while in another slot we perform a mod p' reduction, for $p \neq p'$. This observation lets us allocate k slots to a CRT system where k moduli $p_1, \ldots, p_k$ are used to implement arithmetic in $\mathbb{Z}_p$ for $p := \prod_{i=0}^{k-1} p_i$. Since the vectors contain $N/2$ slots in total, this lets us operate on $(N/2)/k$ slots of $\mathbb{Z}_p$ elements in parallel.

Homomorphic Base Conversion. A large smooth plaintext modulus is insufficient for the applications described in the introduction. Instead, we need support for a large prescribed modulus, such as a 384-bit elliptic curve prime. To do so, first recall the fast base conversion from [HPS19] that is used to switch an element x represented in the



-0.22	3	5.03		···		-2	2.97	-1

$\times$

-0.16	-2	3.79		···		14	2.97	3

Figure 1: A visualization of the homomorphic computer operating on two vectors

CRT basis $\{p_i\}_{0\leq i<k}$ to an element $[x]_r$ in $\mathbb{Z}_r$. Naively, this can be done by lifting the given element x to $\mathbb{Z}$ and then reducing the result modulo r. However, that would be quite expensive as it would require a mod r operation inside the FHE. Instead [HPS19] converts x as follows

$$[x]_r := \left[\sum_{i=0}^{k-1} y_i \cdot [\hat{p}_i]_r - v \cdot [p]_r\right]_r \qquad \text{where} \qquad \begin{array}{l} p := p_0 \cdots p_{k-1}, \quad \hat{p}_i := p/p_i \\ y_i := \left[[x]_{p_i} \cdot \hat{p}_i^{-1}\right]_{p_i} \\ v := \sum_{i=0}^{k-1} \lfloor y_i/p_i \rceil \end{array}$$

This conversion can be used to implement a mod r reduction for an arbitrary integer r. In particular, converting from $\{p_i\}_{0\leq i<k}$ to $\mathbb{Z}_r$ and converting back can be expressed using only arithmetic over $\mathbb{Z}_{p_i}$, $\mathbb{Z}$, and $\mathbb{R}$ (with moderate precision). This lets us implement $\mathbb{Z}_r$ arithmetic without expensive mod r operations inside the FHE.

Nested Residue Number System (RNS). Although our homomorphic RNS system provides both large integer arithmetic and arbitrary modular reduction, it fails to support a very large modulus due to the lack of sufficiently many primes. As the homomorphic computer can only reduce modulo a small integer (up to 4 or 8 bits), there are not enough primes to construct a CRT system expressing a large smooth modulus. At most we can represent a 100-bit modulus. To overcome this, we make use of a secondary CRT layer on top of the first CRT system. That is, we may choose a larger CRT basis $\{r_j\}_{0\leq j<\ell}$, simulate each $\mathbb{Z}_{r_j}$ arithmetic with the arbitrary modular reduction in the previous paragraph, and combine them to support $\mathbb{Z}_r \simeq \prod_{j=0}^{\ell-1} \mathbb{Z}_{r_j}$ arithmetic. The end result is support for $N/(2k\ell)$ parallel slots, each one providing arithmetic over $\mathbb{Z}_r$, inside a single RLWE ciphertext.

Nested Base Conversion. In order to support arbitrary modular reduction, we need to simulate the homomorphic computer we initially had, which supports not only modular reduction but also arithmetic over $\mathbb{Z}$ and $\mathbb{R}$. Addition and multiplication over $\mathbb{Z}$ can be supported directly by embedding $\mathbb{Z}$ inside $\mathbb{Z}_p$, relying on $\mathbb{Z}_p$ operations. Arithmetic over $\mathbb{R}$ is more difficult since we can only use integer arithmetic. We follow the framework of CKKS [CKKS17] that “integer arithmetic plus rescaling implies fixed point arithmetic.” As in CKKS, we construct a new rescaling $\mathsf{Rescale}_{r_j} = x \mapsto (x - [x]_{r_j})/r_j$ from modular reduction by r_j for each j. Using these ingredients, we simulate the homomorphic base conversion again to modulo reduce by any large integer.

A visualization of the complete construction is shown in Figure 2.

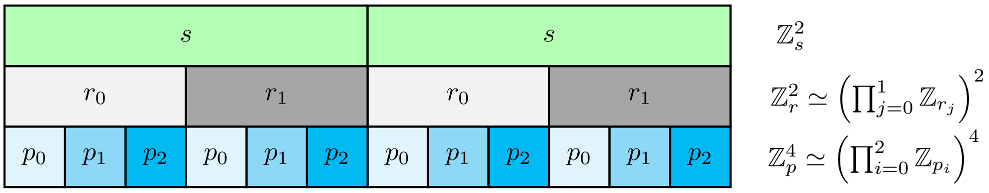

s
s
$\mathbb{Z}_s^2$
r_0
r_1
r_0
r_1
$\mathbb{Z}_r^2 \simeq \left(\prod_{j=0}^{1} \mathbb{Z}_{r_j}\right)^2$
p_0
p_1
p_2
p_0
p_1
p_2
p_0
p_1
p_2
p_0
p_1
p_2
$\mathbb{Z}_p^4 \simeq \left(\prod_{i=0}^{2} \mathbb{Z}_{p_i}\right)^4$

Figure 2: A visualization of the nested CRT system. Given $N/2 = 12$ slots, we first construct four copies of a CRT system $\mathbb{Z}_p \simeq \mathbb{Z}_{p_0} \times \mathbb{Z}_{p_1} \times \mathbb{Z}_{p_2}$. We then embed $\mathbb{Z}_{r_0}$ and $\mathbb{Z}_{r_1}$ inside $\mathbb{Z}_p$, to support two copies of a second CRT system $\mathbb{Z}_r \simeq \mathbb{Z}_{r_0} \times \mathbb{Z}_{r_1}$. Finally, we embed $\mathbb{Z}_s$ inside $\mathbb{Z}_r$ and obtain two slots for $\mathbb{Z}_s$ arithmetic.

1.3 Related Work

Since Gentry [Gen09] proposed the first fully homomorphic encryption (FHE) system, many others have been proposed. For integer computations, a standard choice is to use BGV [BGV12] or BFV [Bra12, FV12], as they naturally support $\mathbb{Z}_t$ arithmetic for $t \in \mathbb{Z}_{>0}$. For applications that require large plaintext precision, one can use BGV/BFV with a large plaintext modulus t. However, as the noise growth is proportional to t, supporting a large t requires large parameters (e.g. a large ciphertext modulus and RLWE ring dimension). As a result, supporting even a few hundred bits of plaintext modulus is nearly impractical. To tackle this problem, CLPX [CLPX18] suggests to replace the modulus t with the polynomial $t = X - b \in \mathbb{Z}[X]$, for some b. Such a choice significantly reduces the noise growth and efficiently handles large integers. However, bootstrapping for such a large modulus is unknown which means that it cannot support unlimited computations.

Recently, an elegant generalization of BFV and CLPX, called Generalized BFV [GV24] (GBFV), was suggested. GBFV achieves both high precision arithmetic (including bootstrapping) and SIMD capability which has not been possible with earlier schemes. The key idea is to use a generalized $t(x) \in \mathbb{Z}[X]$ and choose $t(x)$ carefully to support bootstrapping via switching to BFV and appropriate slot structure for SIMD. In addition, they suggest several choices of $t(x)$ with different precision and number of slots. For instance, they instantiated $\log(N) = 14$ bootstrapping for $p = 2^{16} + 1$, which was nearly impossible with BFV. Despite the improvements, GBFV still cannot support very large precision bootstrapping as they use large precision BFV linear transformations as a subroutine. Furthermore, for the BGV/BFV family the efficiency is bound to the plaintext modulus t. In particular, the number of parallel slots is determined by how the ring $\mathbb{Z}_t[X]/\Phi_M(X)$ splits. If it splits completely then it enjoys maximum parallelism, but otherwise, parallelism is somewhat limited. As a result, the BGV/BFV family often uses a plaintext moduli such as $2^{16} + 1$ that split (almost) completely.

Other options for a high plaintext modulus include CGGI [CGGI16]/DM [DM15] and CKKS [CKKS17]. In CGGI/DM, computing over a large modulus directly is difficult as it significantly increases the FHE parameters, as in BGV/BFV. Instead, one can use radix decomposition [CLOT21], Chinese Remainder Theorem (CRT) [CKLM24a, CKLM24b], or a hybrid between the two [BBB$^+$23]. However, these approaches are not efficient as they require a significant number of bootstrappings. In CKKS, one can rely on the usual CKKS operations to support arithmetic over $\mathbb{C}$ (or $\mathbb{R}$) that contains $\mathbb{Z}$. A more favorable choice is to use the recent developments of the discrete variant of CKKS [DMPS24, CKKL24, BCKS24, BKSS24, KN24] which handles integers more efficiently than the original CKKS.

Recently, [Kim25] suggested a decomposition-based approach to handle large integers, successfully supporting power-of-two moduli like 32 or 64 bits. However, their method is not efficient for a high modulus, because one needs to bootstrap $O(p)$ times to support p-bit arithmetic.

Table 2 compares the existing approaches for supporting a large plaintext modulus with our new FHE.

2 Preliminaries

Let N be an integer that is a power-of-two. Let $Q > 0$ be an integer, $\mathcal{R} = \mathbb{Z}[X]/(X^N+1)$, and $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R}$. The vectors are denoted in bold lower case (e.g. $\boldsymbol{v}, \boldsymbol{w}$). Given a real number $r \in \mathbb{R}$, $\lfloor r \rceil \in \mathbb{Z}$ is the rounding of r sending to the closest integer. Unless stated otherwise, we identify $\mathbb{Z}_t$ with $[-t/2, t/2) \subset \mathbb{Z}$ for $t \in \mathbb{Z}_{>0}$.

		Arbitrary	Bootstrapping	Compactness	Arbitrary
		precision	Bootstrapping	Compactness	modulus
BGV/BFV	Classic	✗	✗	✓	✗
	CRT	✓	✓	✗	✗
CLPX [CLPX18]		✓	✗	✓	✗
GBFV [GV24]		✓	▲	✓	✗
[Kim25]		✓	✓	✗	✗
This paper		✓	✓	✓	✓

Table 2: Comparing the support for a large prescribed modulus in different systems. The arbitrary modulus column denotes whether the scheme supports an arbitrarily large plaintext modulus. The bootstrapping column denotes whether the scheme supports bootstrapping for such computations. The compactness column denotes whether it supports a high modulus using a single RLWE ciphertext (or equivalent). The arbitrary modulus column denotes whether it can choose an arbitrary integer as the plaintext modulus. The BGV/BFT CRT row refers to the use of multiple BGV/BFV ciphertexts with a different plaintext modulus to implement a high plaintext modulus. The ▲ mark for GBFV bootstrapping means that it supports large precision bootstrapping but it cannot support an arbitrarily large plaintext modulus.

2.1 CKKS Basics

In CKKS, complex messages in $\mathbb{C}^{N/2}$ are first *encoded* to a plaintext in $\mathcal{R}$ and then *encrypted* to a ciphertext in $\mathcal{R}_Q^2$. We first explain how encoding and encryption are defined.

Discrete Fourier Transform. In order to connect $\mathbb{C}^{N/2}$ with $\mathcal{R}$, the CKKS scheme uses the discrete Fourier transform (DFT) and its inverse (iDFT). The discrete Fourier transform $\mathsf{DFT} : \mathbb{R}[X]/(X^N+1) \to \mathbb{C}^{N/2}$ is defined as

$$r(X) \in \mathbb{R}[X]/(X^N+1) \mapsto (r(\zeta_i))_{0\le i<N/2} \in \mathbb{C}^{N/2}, \tag{1}$$

where $\zeta_i = \zeta^{5^i}$ for a primitive $2N$-th root of unity ζ. The reason for defining ζ_i this way is that the automorphism group of the ring $\mathbb{Z}[X]/(X^N+1)$ is generated by -1 and 5. With this setup, (1) defines a ring isomorphism, and in particular a (scaled) 2-norm isometry in the sense that

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

for all $r \in \mathbb{R}[X]/(X^N+1)$. The inverse DFT denoted as $\mathsf{iDFT} : \mathbb{C}^{N/2} \to \mathbb{R}[X]/(X^N+1)$ is defined as the inverse of DFT.

Encoding Structure. Let $\boldsymbol{z} \in \mathbb{C}^{N/2}$ be a complex vector. The encoding $\mathsf{Ecd} : \mathbb{C}^{N/2} \to \mathcal{R}$ is defined as

$$\boldsymbol{z} \in \mathbb{C}^{N/2} \mapsto \lfloor \Delta \cdot \mathsf{iDFT}(\boldsymbol{z}) \rceil \in \mathcal{R},$$

where $\Delta \in \mathbb{R}_{>0}$ is a scaling factor used to preserve the precision. The decoding $\mathsf{Dcd} : \mathcal{R} \to \mathbb{C}^{N/2}$ is its approximate inverse defined as

$$m(X) \in \mathcal{R} \mapsto \frac{1}{\Delta} \cdot \mathsf{DFT}(m) \in \mathbb{C}^{N/2}.$$

Encryption. The CKKS encryption is inherited from RLWE. In other words, given a plaintext $m \in \mathcal{R}$, a CKKS ciphertext encrypting m is a pair $(b, a) \in \mathcal{R}_Q^2$ such that

$$(b, a) \cdot \mathsf{sk} = m + e$$

for a secret key $\mathsf{sk} = (1, s) \in \mathcal{R}^2$ and a small error $e \in \mathcal{R}$. Unlike other schemes, the error in CKKS ciphertexts is not separated from the underlying plaintext. In this regard, CKKS encryption is an *approximate* encryption, allowing some errors in it. This sometimes leads to security issues (e.g. $\mathsf{IND\text{-}CPA}^D$ [LM21, LMSS22]).

Homomorphic Operations. Note that both encoding and encryption functions are (approximate) ring homomorphisms, meaning that we can directly use this property to enable addition and multiplication on ciphertexts. Given $\mathsf{ct}_1 = (b_1, a_1), \mathsf{ct}_2 = (b_2, a_2) \in \mathcal{R}_Q^2$, the addition of ct_1 and ct_2 is defined as

$$\mathsf{ct}_1 + \mathsf{ct}_2 = (b_1 + b_2, a_1 + a_2) \in \mathcal{R}_Q^2.$$

The multiplication of ct_1 and ct_2 is defined as

$$\mathsf{ct}_1 \otimes \mathsf{ct}_2 = (b_1 b_2, b_1 a_2 + a_1 b_2, a_1 a_2) \in \mathcal{R}_Q^3.$$

For multiplication, the ciphertext length has increased and we now need to inner product with $(1, s, s^2)$ instead of $(1, s)$ to decrypt it. In order to decrease the length of the ciphertext, we use *relinearization*, which is an RLWE key switching from the secret key s^2 to s. That is, given a ciphertext $\mathsf{ct} = (c_0, c_1, c_2) \in \mathcal{R}_Q^3$, the relinearization is defined as

$$\mathsf{Relin}(c_0, c_1, c_2) = (c_0, c_1) + \mathsf{KeySwitch}_{s^2 \to s}(0, c_2)$$

where $\mathsf{KeySwitch}_{s^2 \to s}$ switches the secret key from s^2 to s. In addition, if the scaling factors of ct_1 and ct_2 are Δ, the product $\mathsf{ct}_1 \otimes \mathsf{ct}_2$ has scaling factor of Δ^2. In order to decrease the size of the scaling factor, we use *rescaling*, which is an approximate division. Given $\mathsf{ct} = (b, a) \in \mathcal{R}_Q^2$ and $q \mid Q$, the rescaling of ct by q is defined as

$$\mathsf{RS}_q(\mathsf{ct}) = (\lfloor b/q \rceil, \lfloor a/q \rceil) \in \mathcal{R}_{Q/q}^2.$$

Bootstrapping. Unlike the exact schemes (BGV/BFV/CGGI/DM) whose bootstrapping cleans the noise of a ciphertext, the CKKS bootstrapping increases the ciphertext modulus while approximately preserving its content. As a black box, the CKKS bootstrapping is defined as follows:

Definition 1 (CKKS Bootstrapping)**.** Let q, Q be integers such that $q < Q$. Let $\mathsf{ct} = \mathsf{Enc} \circ \mathsf{Ecd}(\boldsymbol{z})$ be a CKKS ciphertext encrypting a vector $\boldsymbol{z}$. The CKKS bootstrapping $\mathsf{BTS} : \mathcal{R}_q^2 \to \mathcal{R}_Q^2$ outputs

$$\mathsf{BTS}(\mathsf{ct}) = \mathsf{Enc} \circ \mathsf{Ecd}(\boldsymbol{z}') \in \mathcal{R}_Q^2$$

where $\boldsymbol{z} \simeq \boldsymbol{z}'$.

A typical bootstrapping [CHK$^+$18] includes homomorphic linear transformations (i.e. CtS and StC) and homomorphic modular reduction (i.e. $\mathsf{EvalMod}$), and therefore is significantly more expensive than other homomorphic operations.

2.2 Discrete CKKS

The original CKKS [CKKS17] is defined over the complex plane $\mathbb{C}$. Since $\mathbb{Z} \subset \mathbb{C}$, it naturally supports any integer arithmetic. However, using CKKS for integers without adaptation is not so efficient as it wastes some of its capabilities. The recent approaches for discrete computations over CKKS (initiated in [DMPS24]) provide an efficient framework for handling integers through arithmetic operations and look-up tables.

Discrete Computation and Encoding. In [DMPS24], the authors suggested using CKKS for discrete data by restricting the message space. To elaborate, the message space of discrete CKKS is a discrete set (e.g. $\{0, 1\}, \mathbb{Z}_n$), and there is an additional encoding structure attached to it so that we can connect the discrete space with $\mathbb{C}$.

Definition 2 (Discrete Encoding)**.** Let $U \subseteq \mathbb{C}$ be a discrete set. A discrete CKKS ciphertext $\mathsf{ct} \in \mathcal{R}_Q^2$ is a CKKS ciphertext that encrypts a vector $\boldsymbol{z} \in U^{N/2}$. That is,

$$\mathsf{Dcd}(\mathsf{ct} \cdot (1, s)) = \boldsymbol{z} + \boldsymbol{e}$$

where $\boldsymbol{e} \in \mathbb{C}^{N/2}$ is a small complex vector. We denote $\boldsymbol{z}$ as the underlying message and $\boldsymbol{e}$ as the underlying error.

The most straightforward encoding is to use the identity encoding $\mathbb{Z} \hookrightarrow \mathbb{C}$, which is to regard an integer vector $\boldsymbol{z} \in \mathbb{Z}^{N/2}$ as a valid element of $\mathbb{C}^{N/2}$ and use the usual CKKS encoding $\mathsf{Ecd} : \mathbb{C}^{N/2} \to \mathcal{R}$. As $\mathbb{Z} \hookrightarrow \mathbb{C}$ is a ring homomorphism, homomorphic operations in CKKS can be directly inherited to the new encoding structure.

One of the key advantages of using discrete data rather than approximate data is that we can use interpolation rather than approximation. This allows us to efficiently evaluate discontinuous functions which is very difficult with the usual approximation-based methods in CKKS (e.g. [CKK20]). Another advantage is that one can distinguish the error from the message. When one decrypts and decodes a discrete CKKS ciphertext, one can identify the real message as the closest element of U, removing the error. Furthermore, by using proper polynomial evaluations, one can remove the error homomorphically. For instance, if the message space is restricted to $\{0, 1\}$, the function $h_1 : \mathbb{R} \to \mathbb{R}$ [DMPS24] defined as $h_1(x) = 3x^2 - 2x^3$ can be used as a *cleaning* polynomial that sends the points near 0 and 1 to closer to 0 and 1, respectively. Note that this cleaning corresponds to the bootstrapping framework in the exact schemes.

Definition 3 (Cleaning)**.** Let $U \subseteq \mathbb{C}$ be a discrete set used for discrete encoding. Let $\mathsf{ct} \in \mathcal{R}_Q^2$ be a discrete CKKS ciphertext whose underlying message and error are $\boldsymbol{z}$ and $\boldsymbol{e}$, respectively. A cleaning function $\mathsf{Clean} : \mathcal{R}_Q^2 \to \mathcal{R}_{Q'}^2$ maps ct to $\mathsf{ct}' \in \mathcal{R}_{Q'}^2$ such that the underlying message remains to be the same and the underlying error decreases significantly. That is,

$$\|\boldsymbol{e}'\| \ll \|\boldsymbol{e}\|$$

where $\boldsymbol{e'}$ is the underlying error of ct' and $\| \cdot \|$ is a norm over $\mathbb{C}^{N/2}$ (e.g. $\| \cdot \|_\infty$, $\| \cdot \|_2$).

In this regard, discrete CKKS has two bootstrappings, one for refreshing the error (i.e. Cleaning) and the other for raising the modulus (i.e. CKKS bootstrapping).

The following work [CKKL24] focused on improving the performance of interpolation (i.e. look-up tables) both in terms of efficiency and numerical stability. They mainly suggest to use the roots-of-unity encoding

$$\{\zeta_t^i \mid 0 \le i < t\} \hookrightarrow \mathbb{C}$$

where $\zeta_t = e^{2\pi i/t}$ is the primitive t-th root of unity. One of the biggest advantages of using roots-of-unity is that it supports numerically stable look-up table evaluation (i.e. polynomial interpolation) unlike identity encoding. In addition, it provides a very efficient cleaning function such as $x \mapsto ((t + 1)x - x^{t+1})/t$. For efficiency, [CKKL24] suggests using multivariate interpolation over the roots of unity, as it decreases the polynomial degree (hence decreasing the modulus consumption). Although they are not the first ones to use multivariate polynomials for better efficiency (e.g. [ADE$^+$23]), they thoroughly analyzed the precision and efficiency for the first time.

Discrete Bootstrapping. To bootstrap discrete data, one can indeed rely on the usual CKKS bootstrapping [CHK$^+$18] as the encoded data are still real numbers. For better efficiency, one may consider constructing a dedicated bootstrapping for discrete data, denoted as *discrete bootstrapping.*

Definition 4 (Discrete Bootstrapping)**.** Let q, Q be integers such that $q < Q$, and $U \subseteq \mathbb{C}$ be a discrete encoding set. Let $\mathsf{ct} \in \mathcal{R}_q^2$ be a discrete CKKS ciphertext over U. The discrete bootstrapping $\mathsf{DiBTS} : \mathcal{R}_q^2 \to \mathcal{R}_Q^2$ is defined as

$$\mathsf{DiBTS}(\mathsf{ct}) = \mathsf{ct}'$$

where ct' is a discrete ciphertext encrypting the same message as ct but in a larger modulus.

We observe that the discrete bootstrapping needs not to work for all message space over $\mathbb{C}$, but only over a discrete set U. Hence, one can construct a more efficient bootstrapping than the usual one (e.g. via interpolation rather than approximation).

Following [BKSS24], we provide a simplified overview of the state-of-the-art discrete bootstrapping instantiation.

1. **Slots-to-Coefficients** (StC) : Given $\mathsf{ct} = \mathsf{Enc} \circ \mathsf{Ecd}(\boldsymbol{z}) \in \mathcal{R}_q^2$ encrypting a discrete message $\boldsymbol{z} \in \mathbb{Z}_t \subset \mathbb{Z}$, we output a ciphertext $\mathsf{ct}' = \mathsf{Enc}((q_0/t) \cdot z(X)) \in \mathcal{R}_{q_0}^2$ where $z(X)$ is a polynomial in $\mathcal{R}$ whose coefficients corresponds to the coordinates of $\boldsymbol{z}$. In other words, we put the slots into coefficients.

2. **Modulus Raising** ($\mathsf{ModRaise}$) : Given $\mathsf{ct} = \mathsf{Enc}((q_0/t) \cdot z(X)) \in \mathcal{R}_{q_0}^2$, we output a ciphertext $\mathsf{ct}' = \mathsf{Enc}((q_0/t) \cdot z(X) + q_0 \cdot I(X)) \in \mathcal{R}_{Q_{\mathsf{top}}}^2$ where $I \in \mathcal{R}$ is a small integer polynomial and $Q_{\mathsf{top}} > q_0$.

3. **Coefficients-to-Slots** (CtS) : Given $\mathsf{ct} = \mathsf{Enc}((q_0/t) \cdot z(X) + q_0 \cdot I(X)) \in \mathcal{R}_{Q_{\mathsf{top}}}^2$, output a ciphertext $\mathsf{ct}' = \mathsf{Enc} \circ \mathsf{Ecd}((1/t) \cdot \boldsymbol{z} + \boldsymbol{I})$. In other words, we put the coefficients into slots, while adjusting the scaling factors.

4. **Homomorphic Exponentiation**[4] ($\mathsf{EvalExp}$) : Given a ciphertext $\mathsf{ct} = \mathsf{Enc} \circ \mathsf{Ecd}((1/t) \cdot \boldsymbol{z} + \boldsymbol{I})$, output a ciphertext $\mathsf{ct}' = \mathsf{Enc} \circ \mathsf{Ecd}(e^{2\pi i \boldsymbol{z}/t})$ by homomorphically evaluating a complex exponential $x \mapsto e^{2\pi i x}$.

5. **Homomorphic Look-up Table**[5] (LUT) : Given a ciphertext $\mathsf{ct} = \mathsf{Enc} \circ \mathsf{Ecd}(e^{2\pi i \boldsymbol{z}/t})$ and a look-up table $\mathsf{LUT} : \mathbb{Z}_t \mapsto \mathbb{Z}_t$, output a ciphertext $\mathsf{ct} = \mathsf{Enc} \circ \mathsf{Ecd} \circ \mathsf{LUT}(\boldsymbol{z})$. This can be instantiated with a proper polynomial interpolation.

Note that the previous works [BKSS24, AKP24] further optimize this framework in terms of efficiency and better cleaning functionality. However, we only need discrete bootstrapping in a black-box manner, so we use the above framework for simplicity.

Modular Reduction. More recently, [KN24] introduced an efficient modular reduction for CKKS based on [BKSS24]. The idea is to perform RLWE modular reduction at the bottom modulus q_0 to a coefficients-encoded ciphertext and use discrete bootstrapping to bootstrap it. If one simply inserts a modular reduction step between StC and $\mathsf{ModRaise}$ of the discrete bootstrapping framework described in the previous paragraph and uses the identity look-up table, one achieves a modular reduction in addition to bootstrapping. This is a simplified (non-iterative, integer) version of the modular reduction in [KN24], and is detailed in Algorithm 1. Note that [KN24] extends the algorithm by iterative bootstrapping and discretization. However, we rely only on the most basic (degenerate) case in our paper.

[4] First introduced in [BGGJ20] to transform CGGI/DM ciphertexts into CKKS.

[5] This is based on the look-up table evaluation in [CKKL24].

```
Algorithm 1: IntMod_t [KN24]
  Setting: $\Delta_{\mathsf{StC}} = q_0/t$ where $\Delta_{\mathsf{StC}}$ denotes the scaling factor after StC.
  Input  : $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}} \circ \mathsf{Ecd}(\boldsymbol{z}) \in \mathcal{R}_q^2$ where $\boldsymbol{z} \in \mathbb{Z}^{N/2}$.
  Output: $\mathsf{ct}_{\mathsf{out}} = \mathsf{Enc} \circ \mathsf{Ecd}([\boldsymbol{z}]_t) \in \mathcal{R}_Q^2$.
  $\mathsf{ct}' \leftarrow [\mathsf{StC}(\mathsf{ct})]_{q_0}$;
  $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{LUT}_{\mathsf{id}} \circ \mathsf{EvalExp} \circ \mathsf{CtS} \circ \mathsf{ModRaise}(\mathsf{ct}')$ ;
  return $\mathsf{ct}_{\mathsf{out}}$
```

3 Single Layer RNS System

In this section, we propose an efficient homomorphic encryption scheme for evaluating over large integers. The basic idea is to use CRT as in [CKLM24a, CKLM24b], but we use CRT across slots rather than ciphertexts. To enable this, we generalize the modular reduction framework in [KN24] and evaluate different modular reductions across slots. As a result, we can put at most $N/2$ CRT moduli in a single ciphertext, supporting a large modulus within the ciphertext. Furthermore, we homomorphically evaluate the fast base conversion from [HPS19] to enable homomorphic modular reduction for a large modulus.

3.1 Asymmetric Modular Reduction

The modular reduction in [KN24] evaluates modulo p in a SIMD manner, as it is based on CKKS. We generalize the modular reduction so that the i-th slot evaluates modulo p_i for each $0 \leq i < N/2$. This relies on the fact that CKKS can evaluate different polynomials across the slots by using plaintext-ciphertext multiplication rather than constant-ciphertext multiplication.

Let $\{f_i\}_{0 \leq i < N/2}$ be a family of polynomials of degree at most d and let

$$f_i(x) = a_{i0} + a_{i1}x + \cdots + a_{id}x^d \in \mathbb{C}[x].$$

For each $0 \leq j \leq d$, let $\boldsymbol{a}_j = (a_{0j}, a_{1j}, \ldots, a_{(N/2-1)j}) \in \mathbb{C}^{N/2}$. Given a ciphertext $\mathsf{ct} = \mathsf{Enc} \circ \mathsf{Ecd}(\boldsymbol{z})$ encrypting a complex vector $\boldsymbol{z} \in \mathbb{C}^{N/2}$, evaluating $\{f_i\}$ for each slot i in parallel can be described as

$$\mathsf{Ecd}(\boldsymbol{a_0}) + \mathsf{Ecd}(\boldsymbol{a_1}) \cdot \mathsf{ct} + \cdots + \mathsf{Ecd}(\boldsymbol{a_d}) \cdot \mathsf{ct}^d$$

and can be evaluated with $\log_2(d+1)$ multiplicative depths. Its computational complexity is not very different from evaluating the same polynomial across the slots, as the only difference is that we use plaintext-ciphertext multiplication instead of constant-ciphertext multiplication, and the cost increase is negligible. This is rephrased from the parallel look-up table evaluation in [CKKL24, Section 4.2].

Based on the distinct polynomial evaluation, we propose a modular reduction that takes different modulo p_i across slots. We modify only two parts of the modular reduction algorithm in Algorithm 1. First of all, we add an additional scaling factor adjustment step right before taking modulo q_0. To be more specific, we ensure that taking modulo q_0 can be used to take modulo p_i homomorphically, and the output has elements of $\mathbb{Z}_{p_i}$ encoded in the most significant bits. Secondly, we modify the look-up table step LUT so that it evaluates the identity look-up table $\mathsf{id}_i : \mathbb{Z}_{p_i} \to \mathbb{Z}_{p_i}$ for each i in parallel. The detailed algorithm is described in Algorithm 2.

Note that the scaling factor adjustment can be included into StC and $\mathsf{LUT}_{\{\mathsf{id}_i\}}$ is just as efficient as evaluating the same look-up table across slots, which means that

```
Algorithm 2: $\mathsf{IntMod}_{\{p_i\}_{0\le i<N/2}}$
  Setting: $\Delta_{\mathsf{StC}}$ denotes the scaling factor after $\mathsf{StC}$.
  Input  : $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}} \circ \mathsf{Ecd}(\boldsymbol{z}) \in \mathcal{R}_q^2$ where $\boldsymbol{z} \in \mathbb{Z}^{N/2}$
  Output: $\mathsf{ct}_{\mathsf{out}} = \mathsf{Enc} \circ \mathsf{Ecd}([z_i]_{p_i})_{0\le i<N/2} \in \mathcal{R}_Q^2$.
  $\mathsf{ct}' \leftarrow [\mathsf{Ecd}((\frac{q_0}{\Delta_{\mathsf{StC}}\cdot p_i})_{0\le i<N/2}) \cdot \mathsf{StC}(\mathsf{ct})]_{q_0}$;
  $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{LUT}_{\{\mathsf{id}_i\}} \circ \mathsf{EvalExp} \circ \mathsf{CtS} \circ \mathsf{ModRaise}(\mathsf{ct}')$ ;
  return $\mathsf{ct}_{\mathsf{out}}$
```

the computational complexity is almost the same as the usual modular reduction in Algorithm 1. The correctness is checked by the correctness of the parallel polynomial evaluation, which directly comes from the correctness of CKKS.

3.2 The RNS-based Encryption Scheme

Based on the generalized modular reduction in Section 3.1, we build an efficient FHE scheme for larger integers. In this subsection, we describe the scheme in detail, focusing on how to enable large-precision integer arithmetic. Let $k \mid N/2$ be a (power-of-two) divisor of $N/2$, $\{p_i\}_{0\le i<k}$ be mutually coprime integers, and $p = \prod_{0\le i<k} p_i$. We define an FHE scheme over $\mathbb{Z}_p$ as follows.

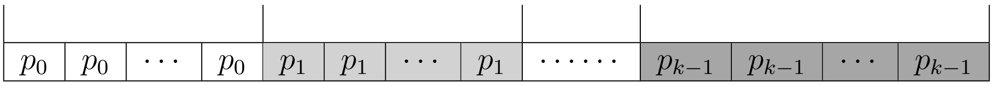
Encoding Structure. Let $\mathsf{CRT} : \prod_{0\le i<k} \mathbb{Z}_{p_i} \to \mathbb{Z}_p$ be a CRT isomorphism and iCRT be its inverse. We extend CRT and iCRT to maps between $\prod_{0\le i<k} \mathbb{Z}_{p_i}^{\frac{N}{2k}}$ and $\mathbb{Z}_p^{\frac{N}{2k}}$. Let $\iota : \prod_{0\le i<k} \mathbb{Z}_{p_i}^{\frac{N}{2k}} \to \mathbb{Z}^{N/2}$ be an inclusion extended from $\mathbb{Z}_{p_i} \mapsto [-p_i/2, p_i/2)$. The encoding $\mathsf{IEcd} : \mathbb{Z}_p^{\frac{N}{2k}} \to \mathcal{R}$ is defined as

$$\mathsf{IEcd}(\boldsymbol{m}) = \mathsf{Ecd} \circ \iota \circ \mathsf{iCRT}(\boldsymbol{m})$$

and the decoding $\mathsf{IDcd} : \mathcal{R} \to \mathbb{Z}_p^{\frac{N}{2k}}$ is defined as its approximate inverse,

$$\mathsf{IDcd}(p(X)) = \mathsf{CRT} \circ \iota^{-1} \circ \mathsf{Dcd}(p(X)).$$

To clarify, the modulus for each slot is distributed as in Figure 3.



p_0	p_0	$\cdots$	p_0	p_1	p_1	$\cdots$	p_1	$\cdots\cdots$	p_{k-1}	p_{k-1}	$\cdots$	p_{k-1}

Figure 3: The assigned modulus for each slot $0 \le i < N/2$.

Modular Reduction. Encryption, decryption, addition, and multiplication are directly inherited from CKKS. Note that addition and multiplication give only addition and multiplication over $\mathbb{Z}$, which means that it does not immediately give arithmetic over $\mathbb{Z}_p$. To complete the homomorphic operations, we need to define homomorphic modular reduction, which we rely on the discussions in the previous subsection. In other words, the modular reduction $\mathsf{Mod}_p : \mathcal{R}_q^2 \to \mathcal{R}_Q^2$ is defined as

$$\mathsf{Mod}_p(\mathsf{ct}) = \mathsf{IntMod}_{\{p_i\}_{0\le i<k}}(\mathsf{ct})$$

where we duplicate $\{p_i\}_{0\le i<k}$ $\frac{N}{2k}$ times to get the desired modular reduction. The straightforward way of using modular reduction is to modular-reduce every time we

perform addition or multiplication, but a more efficient approach is to delay the modular reduction until the precision budget is full. Such strategy was briefly mentioned in [KN24, Section 5.1] and will be used throughout the following sections.

Theorem 1 (Scheme Correctness)**.** *Let* $\mathsf{ct}_1, \mathsf{ct}_2 \in \mathcal{R}_Q^2$ *be encrypting* $\boldsymbol{m_1}, \boldsymbol{m_2} \in \mathbb{Z}_p^{\frac{N}{2k}}$*, respectively. Then* $\mathsf{Mod}_p(\mathsf{ct}_1 + \mathsf{ct}_2)$ *and* $\mathsf{Mod}_p(\mathsf{ct}_1 \cdot \mathsf{ct}_2)$ *encrypt* $\boldsymbol{m_1} + \boldsymbol{m_2}$ *and* $\boldsymbol{m_1} \odot \boldsymbol{m_2}$*, respectively.*

Proof. Let $\boldsymbol{n_1} = \iota \circ \mathsf{iCRT}(\boldsymbol{m_1})$ and $\boldsymbol{n_2} = \iota \circ \mathsf{iCRT}(\boldsymbol{m_2})$ as in the definition of IEcd. As the natural inclusion $\mathbb{Z} \hookrightarrow \mathbb{C}$ is a ring homomorphism, $\mathsf{ct}_1 + \mathsf{ct}_2$ and $\mathsf{ct}_1 \cdot \mathsf{ct}_2$ encrypt vectors $\boldsymbol{n_1} + \boldsymbol{n_2}$ and $\boldsymbol{n_1} \odot \boldsymbol{n_2}$, respectively. At this point, the value in slot i may not belong to $\mathbb{Z}_{p_i}$ as we performed addition and multiplication over integers. The modular reduction function Mod_p performs $[\cdot]_{p_i}$ for each slot i, enabling addition and multiplication over $\mathbb{Z}_{p_i}$ instead of $\mathbb{Z}$. Finally, as CRT is an isomorphism, we see that we achieved addition and multiplication over $\mathbb{Z}_p$. □ □

Automorphisms. Let $\mathsf{leftRot}_i$ and $\mathsf{rightRot}_i$ be usual CKKS rotations for index $0 \leq i < N/2$. Automorphisms are inherited from CKKS rotations, which can be categorized into two types. The first type is of rotation index $0 \leq i < \frac{N}{2k}$. This type of rotation is an analogue of the usual CKKS rotation as the vector in $\mathbb{Z}_p^{\frac{N}{2k}}$ is rotated as usual, except that we need some masking. That is, in order to left rotate by index α, one first needs to mask both the α slots on the left and the $\frac{N}{2k} - \alpha$ slots on the right, right rotate the former by α, left rotate the latter by α, and add them together.

Definition 5 (Outer Rotation)**.** Let $\mathsf{ct} \in \mathcal{R}_Q^2$ be encrypting $\boldsymbol{m} \in \mathbb{Z}_p^{\frac{N}{2k}}$. The outer rotation by index $0 \leq \alpha < \frac{N}{2k}$ is defined as

$$\mathsf{OutLeftRot}_\alpha(\mathsf{ct}) = \mathsf{leftRot}_\alpha(\boldsymbol{y} \cdot \mathsf{ct}) + \mathsf{rightRot}_{\frac{N}{2k} - \alpha}((1 - \boldsymbol{y}) \cdot \mathsf{ct})$$

where $\boldsymbol{y}$ is the mask vector that has α 0's and $\frac{N}{2k} - \alpha$ 1's.

The second type is of rotation index $\frac{N}{2k}\beta$ for $0 \leq \beta < k$. This type of rotation rotates inside the CRT structure, meaning that the order $(p_0, p_1, \ldots, p_{k-1})$ is shifted to $(p_\beta, p_{\beta+1}, \ldots, p_{k-1}, p_0, \ldots, p_{\beta-1})$.

Definition 6 (Inner Rotation)**.** Let $\mathsf{ct} \in \mathcal{R}_Q^2$ be encrypting $\boldsymbol{m} \in \mathbb{Z}_p^{\frac{N}{2k}}$. The inner rotation by index $0 \leq \beta < k$ is defined as

$$\mathsf{InLeftRot}_\beta(\mathsf{ct}) = \mathsf{leftRot}_{\frac{N\beta}{2k}}(\mathsf{ct}).$$

The first type of rotation can be used for multivariate evaluation (e.g. matrix-vector multiplication) as in the original CKKS whereas the second type of rotation can be used to enable computations that need to be defined over different modulo arithmetic (e.g. look-up tables).

Cleaning. As the scheme relies on the discrete variant of CKKS, we need to clean the errors as in [DMPS24]. For efficiency, we prefer cleaning in the roots-of-unity state as described in [CKKL24]. Therefore, cleaning is performed inside of bootstrapping rather than outside, which means that we may put an additional cleaning step during Mod_p. As illustrated in [BKSS24, AKP24], there is an alternative option to find an appropriate polynomial interpolation with vanishing derivatives, which improves the efficiency greatly. In this paper, we only need cleaning in a black box manner so one may choose any options they want.

3.3 Homomorphic Base Conversion

Let $r < \sqrt{p}$ be a large modulus for which we take a modular reduction. We keep the CRT structure in the previous subsection, meaning that the encrypted data is stored modulo p_i for $0 \leq i < k$. The key observation is that we can perform base conversion from $\{p_i\}$ to $\{r\}$ and vice versa, without having an intermediate ciphertext encrypting a large integer modulo r. We homomorphically evaluate the fast base conversion in [HPS19] that uses floating point arithmetic. Using this technique allows us to compute base conversion with look-up tables of size $O(p_i)$ rather than $O(r)$.

We first recall the fast base conversion in [HPS19]. For simplicity, we discuss the case where the input moduli are $\{p_i\}_{0\leq i<k}$ and the output modulus is r. The fast base conversion of $x \in \mathbb{Z}_p$ (for $p = \prod_{i=0}^{k-1} p_i$) to r can be described as

$$[x]_r = \left[\sum_{i=0}^{k-1} y_i \cdot [\hat{p_i}]_r - v \cdot [p]_r\right]_r$$

where $\hat{p_i} = p/p_i \in \mathbb{Z}$, $y_i = [[x]_{p_i} \cdot \hat{p_i}^{-1}]_{p_i}$, and

$$v = \left\lfloor \sum_{i=0}^{k-1} \frac{y_i}{p_i} \right\rceil .$$

The key observation is that y_i and $[\hat{p_i}]_r$ depend only on the modulus p_i, and v can be computed with simple (real number) additions. Hence, when homomorphically evaluating the base conversion, we do not need to handle a large modulus $p = \prod_{i=0}^{k-1} p_i$ directly but needs k small moduli p_i and one small precision real number computer.

For efficiency, we do not compute the last $[\cdot]_r$, resulting in

$$\sum_{i=0}^{k-1} y_i \cdot [\hat{p_i}]_r - v \cdot [p]_r = [x]_r + re \tag{2}$$

for some small error $e \in \mathbb{Z}$. For modulo p_i, we need to compute

$$\left[\sum_{j=0}^{k-1} y_j \cdot [[\hat{p_j}]_r]_{p_i} - v \cdot [[p]_r]_{p_i}\right]_{p_i} . \tag{3}$$

Here $[[\hat{p_j}]_r]_{p_i}$ and $[[p]_r]_{p_i}$ can be precomputed so we only need to compute y_j for each j and compute v. We prove the correctness property as follows.

Theorem 2 (Modulo Arithmetic over CRT Basis). *Let $p = \prod_{i=0}^{k-1} p_i$ be a CRT system and r be a target modulus. Given $x \in \mathbb{Z}$ such that $|x| < p/2$, let*

$$x' = \sum_{i=0}^{k-1} y_i \cdot [\hat{p_i}]_r - v \cdot [p]_r$$

where $\hat{p_i} = p/p_i$, $y_i = [[x]_{p_i} \cdot \hat{p_i}^{-1}]_{p_i}$, and $v = \left\lfloor \sum_{i=0}^{k-1} \frac{y_i}{p_i} \right\rceil$. Then we have $x \equiv x'$ modulo r and

$$|x'| \leq \frac{1}{4}\left(k + \sum_{i=0}^{k-1} p_i\right) \cdot r.$$

Proof. We first check the property $x \equiv x'$ modulo r. It suffices to show that

$$\sum_{i=0}^{k-1} y_i \cdot \hat{p_i} - v \cdot p \equiv x \pmod{r}.$$

Since

$$\sum_{i=0}^{k-1} y_i \cdot \hat{p_i} - v \cdot p \equiv x_i \cdot \hat{p_i}^{-1} \cdot \hat{p_i} \equiv x_i = [x]_{p_i} \pmod{p_i}$$

for each j, we have that

$$\sum_{i=0}^{k-1} y_i \cdot \hat{p_i} - v \cdot p \equiv x \pmod{p_j}.$$

In addition, as

$$\sum_{i=0}^{k-1} y_i \cdot \hat{p_i} - v \cdot p = p \cdot \left(\sum_{i=0}^{k-1} \frac{y_i}{p_i} - \left\lfloor \sum_{i=0}^{k-1} \frac{y_i}{p_i} \right\rceil \right) \in [-p/2, p/2)$$

we have that $\sum_{i=0}^{k-1} y_i \cdot \hat{p_i} - v \cdot p = x$ which implies that they are equal modulo r.

Second, we check the upper bound of $|x'|$. By triangle inequality,

$$|x'| \leq \sum_{i=0}^{k-1} |y_i| \cdot (r/2) + v \cdot (r/2) \leq \left(\sum_{i=0}^{k-1} (p_i/2) + (k/2) \right) \cdot (r/2).$$

Hence we have the desired upper bound. □ □

Next, we discuss how to evaluate Equation (3) homomorphically in parallel.

1. **Compute y_i and v**: Since we start with a CRT system storing $[x]_{p_i}$ for each i, we can compute y_i in parallel across slots. We end with modular reduction and have fresh y_i's in the largest modulus. Next, v can be approximately computed as rotation-sum and constant multiplication (at most one multiplicative depth). In other words, we compute $\sum_{i=0}^{k-1} y_i/p_i$ and regard it as a valid discrete encoding of $v = \lfloor \sum_{i=0}^{k-1} y_i/p_i \rceil$.

2. **Computing the rest**: By a proper rotation-sum, the i-th slot computes

$$\sum_{j=0}^{k-1} y_j \cdot [[\hat{p_j}]_r]_{p_i}$$

in a parallel manner. Next, by using a ciphertext having v across the slots (in the previous step), compute

$$v \cdot [[p]_r]_{p_i}$$

in parallel. Finally, we subtract and modular reduce.

The total cost should be roughly 2-3 times the bootstrapping cost. The detailed algorithm is described in Algorithm 3. The correctness of the algorithm is checked by the individual components in Section 3.2 and the following lemma.

Lemma 1. *Let p, p_i, r, x, y_i, v, x' be as in Theorem 2. If $|x| \leq \epsilon p$, then*

$$\left| v - \sum_{i=0}^{k-1} \frac{y_i}{p_i} \right| \leq \epsilon.$$

Algorithm 3: IMod_r

```
Setting: $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$, $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R}$, $k \mid N/2$.
Input  : $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}} \circ \mathsf{IEcd}(\boldsymbol{z}) \in \mathcal{R}_Q^2$ where $\boldsymbol{z} \in \mathbb{Z}_p^{\frac{N}{2k}}$ is a small vector.
Output: $\mathsf{ct}_{\mathsf{out}} = \mathsf{Enc} \circ \mathsf{IEcd}([\boldsymbol{z}]_r + r\boldsymbol{e}) \in \mathcal{R}_Q^2$.
$\mathsf{ct}' \leftarrow \mathsf{ct} \cdot ([\hat{p_i}^{-1}]_{p_i})_{0 \le i < k}$;
$\mathsf{ct}_y \leftarrow \mathsf{Mod}_p(\mathsf{ct}')$;
$\mathsf{ct}_v \leftarrow \mathsf{ct}_y \cdot (1/p_i)_{0 \le i < k}$;
for $i = 0$ to $\log_2(k) - 1$ do
    $\mathsf{ct}_v \leftarrow \mathsf{ct}_v + \mathsf{InLeftRot}_{k/2^{i+1}}(\mathsf{ct}_v)$;
end for
$\mathsf{ct}_{\mathsf{out}} \leftarrow 0$;
for $i = 0$ to $k - 1$ do
    $\mathsf{ct}' \leftarrow \mathsf{ct}_y \cdot ([[\hat{p_j}]_r]_{p_{j-i}})_{0 \le j < k}$;
    $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{ct}_{\mathsf{out}} + \mathsf{InLeftRot}_i(\mathsf{ct}')$;
end for
$\mathsf{ct}_v \leftarrow \mathsf{ct}_v \cdot ([[p]_r]_{p_i})_{0 \le i < k}$;
$\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{ct}_{\mathsf{out}} - \mathsf{ct}_v$;
$\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{Mod}_p(\mathsf{ct}_{\mathsf{out}})$;
return $\mathsf{ct}_{\mathsf{out}}$
```

Proof. In the proof of Theorem 2, we checked that

$$\sum_{i=0}^{k-1} y_i \cdot \hat{p}_i - v \cdot p = x.$$

After dividing by p, we get

$$\sum_{i=0}^{k-1} y_i/p_i - v = x/p.$$

Hence,

$$\left| v - \sum_{i=0}^{k-1} \frac{y_i}{p_i} \right| = |x/p| \le \epsilon.$$

This finishes the proof. □ □

Theorem 3 (Correctness of IMod_r). *Let* $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}} \circ \mathsf{IEcd}(\boldsymbol{z})$ *be a ciphertext encrypting a vector* $\boldsymbol{z} = (z_j)_{0 \le j < \frac{N}{2k}} \in \mathbb{Z}_p^{\frac{N}{2k}}$ *such that* $|z_j| \ll \min_i(\hat{p_i})$ *for each* $0 \le j < \frac{N}{2k}$. *Then Algorithm 3 evaluates a correct (lazy) modular reduction.*

Proof. We assume that the components in Section 3.2 are correct. We observe that Algorithm 3 homomorphically evaluates all the procedures to evaluate x' in Theorem 2, except that it approximately evaluates v by choosing $\sum_{i=0}^{k-1} y_i/p_i$ as a representative of v. Since $|z_j| \ll \min_i(\hat{p_i})$, Lemma 1 tells us that the error

$$\sum_{i=0}^{k-1} \frac{y_i}{p_i} - v = \frac{x}{p}$$

is sufficiently smaller than $1/p_i$ for every $0 \le i < k$, which is enough to be regarded as a valid encoding of an integer (in the CRT system) after multiplying $[[p]_r]_{p_i}$. Finally, Theorem 2 finishes the proof. □ □

Next, we discuss the base conversion within the set $\{p_i\}_{0\leq i<k}$. In particular, we consider the base conversion from $P_I = \{p_i \mid i \in I\} \subseteq \{p_i\}_{0\leq i<k}$ to $P_{[k]\setminus I} = \{p_i \mid i \notin I\}$. As in the previous case, we use the fast base conversion in [HPS19] but this time the base conversion is exact. For each $i \in [k] \setminus I$, the base conversion can be described as

$$\{x_i\}_{i\in I} \quad \mapsto \quad \left[\sum_{i\in I} y_i \cdot [\hat{p_i}]_{p_j} - v \cdot [p]_{p_j}\right]_{0\leq j<k}$$

where $\hat{p_i} = \left(\prod_{u\in I} p_u\right)/p_i$, $y_i = [[x]_{p_i} \cdot \hat{p_i}^{-1}]_{p_i}$, and

$$v = \left\lfloor \sum_{i\in I} \frac{y_i}{p_i} \right\rceil .$$

Unlike in the previous case, we can take the final $[\cdot]_{p_j}$ as its representation is in the moduli chain. We assume that the input of the base conversion is sufficiently small so that we can apply an analogue of Lemma 1. The evaluation is almost identical to Algorithm 3, and we leave it as an exercise. We denote the base conversion defined here as $\mathsf{BaseConv}_I$. This type of base conversion is not very useful at this point, but will be used as an ingredient in the next section.

Theorem 4 (Correctness of $\mathsf{BaseConv}_I$)**.** *Let* $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}} \circ \mathsf{IEcd}(\boldsymbol{z})$ *be a ciphertext encrypting a vector* $\boldsymbol{z} = (z_j)_{0\leq j<\frac{N}{2k}} \in \mathbb{Z}_p^{\frac{N}{2k}}$ *such that* $|z_j| \ll \min_{i\in I}(\hat{p_i})$ *for each* $0 \leq j < \frac{N}{2k}$*. Then* $\mathsf{BaseConv}_I$ *evaluates a correct modular reduction by* $\prod_{i\in I} p_i$*.*

Proof. It suffices to show that $\sum_{i\in I} y_i/p_i$ is a good representation of v. Similarly as in Theorem 3, we have

$$\sum_{i\in I} \frac{y_i}{p_i} - v = \frac{[x]_{p_I}}{p_I}$$

where $p_I = \prod_{i\in I} p_i$. Therefore, the representation error is much less than $1/p_j$ for each $0 \leq j < \frac{N}{2k}$, leading to sufficient precision. □ □

4 Double Layer RNS System

Although the scheme in Section 3 provides efficient homomorphic computations over large integers, its practical performance is somewhat limited. The main problem is that there are not enough number of mutually coprime moduli. Recall that the (non-iterative) modular reduction in [KN24] can only handle moduli of moderate size (e.g. 4 to 8 bits). For instance, for $\log N = 16$, the maximum modulus is a scale of 10 bits (see [BKSS24]), and the number of primes less than 1024 is 172 which is way smaller than $N/2 = 32768$. To tackle this issue, we observe that we can build an additional CRT layer upon the CRT layer so that we can enable an even larger modulus.

4.1 Homomorphic Rescaling

Given two different types of base conversions in the previous section, we can naturally define homomorphic rescaling which mimics the rescaling in CKKS. That is, given $x \in \mathbb{Z}_p$, we can either take $[x]_r + re$ for a large modulus r or $[x]_{\prod_{i\in I} p_i}$, and define rescaling by subtracting it from the original ciphertext and multiply the modular inverse of the modulus. For the latter, one loses the data for $\{p_i\}_{i\in I}$ while division, so it should be recovered using an extra base conversion. To be explicit, we can briefly illustrate the algorithms as follows.

Definition 7 (Rescale by r)**.** Let $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}} \circ \mathsf{IEcd}(\boldsymbol{z})$ be a ciphertext encrypting a vector $\boldsymbol{z} = (z_j)_{0 \le j < \frac{N}{2k}} \in \mathbb{Z}_p^{\frac{N}{2k}}$ such that $|z_j| \ll \min_i(\hat{p_i})$ for each $0 \le j < \frac{N}{2k}$. The rescaling of ct by r (that is coprime with $\prod_{i=0}^{k-1} p_i$) is defined as

$$\mathsf{Rescale}_r(\mathsf{ct}) = r^{-1} \cdot (\mathsf{ct} - \mathsf{IMod}_r).$$

Definition 8 (Rescale by $\prod_{i \in I} p_i$)**.** Let $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}} \circ \mathsf{IEcd}(\boldsymbol{z})$ be a ciphertext encrypting a vector $\boldsymbol{z} = (z_j)_{0 \le j < \frac{N}{2k}} \in \mathbb{Z}_p^{\frac{N}{2k}}$ such that

- $|[z_j]_{p_I}| \ll \prod_{i \in I} p_i / \max_{i \in I}(p_i)$ and
- $|z_j| \ll \prod_{0 \le i < k} p_i / \max_{i \notin I} p_i$

for each $0 \le j < \frac{N}{2k}$. The rescaling of ct by $\prod_{i \in I} p_i$ is defined as

$$\mathsf{Rescale}_{p_I} = \mathsf{BaseConv}_{[k] \setminus I} \left(\left(\prod_{i \in I} p_i \right)^{-1} \cdot (\mathsf{ct} - \mathsf{BaseConv}_I(\mathsf{ct})) \right).$$

Note that $\mathsf{Rescale}_r$ is an approximate rescaling as in CKKS, while $\mathsf{Rescale}_{P_I}$ gives an exact result. One caution here is that $\mathsf{Rescale}_{P_I}$ works properly only when both the underlying message of ct is much smaller than $\prod_{i \in I} p_i$ and the message of

$$\left(\prod_{i \in I} p_i \right)^{-1} \cdot (\mathsf{ct} - \mathsf{BaseConv}_I(\mathsf{ct}))$$

is much smaller than $\prod_{i \in [k] \setminus I} p_i$. This holds only in very specific circumstances but can be useful in the next subsections.

Theorem 5 (Rescale Correctness)**.** *The rescaling operations in Definitions 7 and 8 are correct.*

Proof. The correctness of Definition 7 follows directly from Theorem 3. It remains to show the correctness of Definition 8.

Since $|[z_j]_{p_I}| \ll \prod_{i \in I} p_i / \max_{i \in I}(p_i)$, by Theorem 4 we have that

$$\left(\prod_{i \in I} p_i \right)^{-1} \cdot (\mathsf{ct} - \mathsf{BaseConv}_I(\mathsf{ct}))$$

gives the correct rescaling. Next, the condition $|z_j| \ll \prod_{0 \le i < k} p_i / \max_{i \notin I} p_i$ tells us that the message after rescaling by p_I is less than $\prod_{i \notin I} p_i / \max_{i \notin I} p_i$. According to Theorem 4, this ensures the correctness of the last base conversion. This finishes the proof. □ □

4.2 The Nested RNS-based Encryption Scheme

Using modular reduction in the previous section and the rescaling in the prior subsection, we construct a nested two-layer RNS system to encode large integers. Let $k \mid N/2$ be the number of first layer moduli as before, and let $\ell \mid (N/2)/k$ be an integer which will be the number of second layer moduli. There are two sets of mutually coprime integers, namely $\{p_i\}_{0 \le i < k}$ and $\{r_j\}_{0 \le j < \ell}$, where $\max_j(r_j) \ll (\prod_i p_i)^{1/2}$. Let $r = \prod_j r_j$. We define an FHE scheme over $\mathbb{Z}_r$ as follows.

Encoding Structure. In addition to the encoding IEcd, we attach an additional CRT. Let $\mathsf{CRT}' : \prod_{0 \le j < \ell} \mathbb{Z}_{r_j} \to \mathbb{Z}_r$ be a CRT isomorphism and iCRT' be its inverse. We extend CRT' and iCRT' to maps between $(\prod_{0 \le j < \ell} \mathbb{Z}_{r_j})^{\frac{N}{2k\ell}}$ and $\mathbb{Z}_r^{\frac{N}{2k\ell}}$. Let $\iota' : (\prod_{0 \le j < \ell} \mathbb{Z}_{r_j})^{\frac{N}{2k\ell}} \to \mathbb{Z}_p^{\frac{N}{2k}}$ be an inclusion extended from $\mathbb{Z}_{r_j} \mapsto [-r_j/2, r_j/2)$. The encoding $\mathsf{JEcd} : \mathbb{Z}_r^{\frac{N}{2k\ell}} \to \mathcal{R}$ is defined as

$$\mathsf{JEcd}(\boldsymbol{m}) = \mathsf{IEcd} \circ \iota' \circ \mathsf{iCRT}'(\boldsymbol{m})$$

and the decoding $\mathsf{JDcd} : \mathcal{R} \to \mathbb{Z}_r^{\frac{N}{2k\ell}}$ is defined as its approximate inverse,

$$\mathsf{JDcd}(p(X)) = \mathsf{CRT}' \circ \iota'^{-1} \circ \mathsf{IDcd}(p(X))$$

where ι'^{-1} involves modular reduction by r_j for each j. To clarify, the modulus for each slot is distributed as in Figure 4. Note that the first layer ordering and the second layer ordering are different. In particular, the first layer ordering maps $\prod \mathbb{Z}_p^{\frac{N}{2k}}$ to $\prod_i \mathbb{Z}_{p_i}^{\frac{N}{2k}}$ whereas the second layer ordering maps $\prod \mathbb{Z}_r^{\frac{N}{2k\ell}}$ to $(\prod_j \mathbb{Z}_{r_j})^{\frac{N}{2k\ell}}$.

r_0	r_1	r_0	r_1	r_0	r_1	r_0	r_1	r_0	r_1	r_0	r_1	r_0	r_1	r_0	r_1
p_0	p_0	p_0	p_0	p_1	p_1	p_1	p_1	p_2	p_2	p_2	p_2	p_3	p_3	p_3	p_3

Figure 4: The assigned modulus for each slot $0 \le i < N/2$.

Arithmetics Operations. Encryption, decryption, addition, and multiplication are directly inherited from CKKS. As addition and multiplication give only addition and multiplication over $\mathbb{Z}$, we need homomorphic modular reduction to control its size. For this, we rely on the modular reduction in Algorithm 3. We extend this in a similar manner so that each slot that corresponds to r_j modular reduce by r_j. We denote this extension as $\mathsf{Mod}'_r = \mathsf{IMod}_{\{r_j\}_{0 \le j < \ell}}$. Note that $\mathsf{Rescale}_r$ in Definition 7 extends to $\mathsf{Rescale}'_r = \mathsf{Rescale}_{\{r_j\}_{0 \le j < \ell}}$.

Theorem 6 (Scheme Correctness)**.** *Let* $\mathsf{ct}_1, \mathsf{ct}_2 \in \mathcal{R}_Q^2$ *be encrypting* $\boldsymbol{m_1}, \boldsymbol{m_2} \in \mathbb{Z}_r^{\frac{N}{2k\ell}}$*, respectively. Then* $\mathsf{Mod}'_r(\mathsf{ct}_1 + \mathsf{ct}_2)$ *and* $\mathsf{Mod}'_r(\mathsf{ct}_1 \cdot \mathsf{ct}_2)$ *encrypt* $\boldsymbol{m_1} + \boldsymbol{m_2}$ *and* $\boldsymbol{m_1} \odot \boldsymbol{m_2}$*, respectively.*

Proof. This immediately follows from the correctness of IMod_r which was proved in Theorem 3. The condition $\max_j(r_j) \ll (\prod_i p_i)^{1/2}$ is used to ensure that the product $\mathsf{ct}_1 \cdot \mathsf{ct}_2$ does not overflow. □ □

Automorphisms Automorphisms are inherited from the single-layer scheme. The inner rotation $\mathsf{InLeftRot}_\beta$ rotates over p_i's, so it is not so useful in the second layer. Now the outer rotation $\mathsf{OutLeftRot}_\alpha$ is classified into two cases: inner and outer rotations for $\mathbb{Z}_r^{\frac{N}{2k\ell}}$.

Definition 9 (Inner Rotation)**.** Let $\mathsf{ct} \in \mathcal{R}_Q^2$ be encrypting $\boldsymbol{m} \in \mathbb{Z}_r^{\frac{N}{2k\ell}}$. The inner rotation by index $0 \le j < \ell$ is defined as

$$\mathsf{OInLeftRot}_j(\mathsf{ct}) = \mathsf{OutLeftRot}_j(\mathsf{ct}).$$

The outer rotations need an additional level of masking to be supported with $\mathsf{OutLeftRot}$'s, as in the outer rotations of layer 1. Since we rely only on inner rotations, we do not illustrate this further.

Cleaning. Since the first layer is already an exact FHE scheme, we do not need any extra cleaning. Therefore, it suffices to control the errors only in the first layer, which one can refer to Section 3.

4.3 Homomorphic Base Conversion

Let $s \ll (\prod_j r_j)^{1/2}$ be a large modulus. We somewhat repeat the homomorphic base conversion in Section 3.3 but this time in the second layer. Note that we already have all the ingredients: addition, multiplication, modular reduction, and rescaling. In particular, a combination of multiplication and rescaling gives real number computation as in the philosophy of CKKS, which is essential in evaluating the fast base conversion in [HPS19].

In order to compute

$$v \approx \sum_{i=0}^{\ell-1} \frac{y_i}{r_i}$$

where $y_i = [[x]_{r_i} \cdot \hat{r_i}^{-1}]_{r_i}$, we need homomorphic division by r_i.

As $\mathsf{Rescale}_{\{r_i\}_{0 \le i < \ell}}$ itself is not so precise, we multiply and divide by $\prod_{i \in I} p_i$ via $\mathsf{Rescale}_{p_I}$, resulting in an exact result. The algorithm is described in Algorithm 4.

```
Algorithm 4: $\mathsf{JMod}_s$
  Setting: $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$, $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R}$, $k, \ell \mid N/2$, $I \subseteq \{0, 1, \ldots, k-1\}$,
           $\max_j r_j^2 \ll \prod_{i \notin I} p_i$, $\prod_{i \in I} p_i > (k + 4 + \sum_{i=0}^{k-1} p_i) \cdot \ell/2$.
  Input  : $\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}} \circ \mathsf{JEcd}(\boldsymbol{z}) \in \mathcal{R}_Q^2$ where $\boldsymbol{z} \in \mathbb{Z}_r^{\frac{N}{2k\ell}}$ satisfies
           $$\|\boldsymbol{z}\|_\infty / r \ll \min\left(1/\max_{i \in I}(p_i), 1/\max_{0 \le j < \ell}(r_j)\right)$$
           and
           $$\|\iota' \circ \mathsf{iCRT}'(\boldsymbol{z})\|_\infty \ll \prod_{i \notin I} p_i / \max_i p_i.$$
  Output: $\mathsf{ct}_{\mathsf{out}} = \mathsf{Enc} \circ \mathsf{JEcd}([\boldsymbol{z}]_s + s\boldsymbol{e}) \in \mathcal{R}_Q^2$ for some small $\boldsymbol{e}$.
  $\mathsf{ct}' \leftarrow \mathsf{ct} \cdot \mathsf{IEcd} \circ \iota'(([\hat{r_i}^{-1}]_{r_i})_{0 \le i < \ell})$;
  $\mathsf{ct}_y \leftarrow \mathsf{Mod}'_r(\mathsf{ct}')$;
  $\mathsf{ct}_v \leftarrow \mathsf{Rescale}_{\{r_i\}_{0 \le i < \ell}}(\mathsf{ct}_y \cdot \mathsf{IEcd}(\prod_{i \in I} p_i))$;
  for $i = 0$ to $\log_2(\ell) - 1$ do
      $\mathsf{ct}_v \leftarrow \mathsf{ct}_v + \mathsf{OInLeftRot}_{\ell/2^{i+1}}(\mathsf{ct}_v)$;
  end for
  $\mathsf{ct}_v \leftarrow \mathsf{Rescale}_{p_I}(\mathsf{ct}_v)$;
  $\mathsf{ct}_{\mathsf{out}} \leftarrow 0$;
  for $i = 0$ to $\ell - 1$ do
      $\mathsf{ct}' \leftarrow \mathsf{ct}_y \cdot \mathsf{IEcd} \circ \iota'([[\hat{r_j}]_s]_{r_{j-i}})_{0 \le j < \ell}$;
      $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{ct}_{\mathsf{out}} + \mathsf{OInLeftRot}_i(\mathsf{ct}')$;
  end for
  $\mathsf{ct}_v \leftarrow \mathsf{ct}_v \cdot \mathsf{IEcd} \circ \iota'([[r]_s]_{r_i})_{0 \le i < \ell}$;
  $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{ct}_{\mathsf{out}} - \mathsf{ct}_v$;
  $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{Mod}'_r(\mathsf{ct}_{\mathsf{out}})$;
  return $\mathsf{ct}_{\mathsf{out}}$
```

Theorem 7 (Correctness of JMod_s)**.** *The homomorphic modular reduction in Algorithm 4 is correct.*

Proof. We first check the required conditions of $\mathsf{Rescale}_{\{r_i\}_{0\le i<\ell}}$ and $\mathsf{Rescale}_{p_I}$.

For checking $\mathsf{Rescale}_{\{r_i\}_{0\le i<\ell}}$, we see that

$$\left(\prod_{i\in I} p_i\right)\cdot \|\iota' \circ \mathsf{iCRT}'(\boldsymbol{z})\|_\infty \ll \left(\prod_{0\le i<k} p_i\right) / \max_{0\le i<k}(p_i)$$

which satisfies the condition of Definition 7. For checking $\mathsf{Rescale}_{p_I}$, let $\boldsymbol{v}$ be the vector that ct_v is encrypting (via IEcd) before applying rescaling in Line 7 of Algorithm 4. By definition, it should be encrypting

$$\approx \left(\prod_{i\in I} p_i\right)\cdot \sum_{i=0}^{\ell-1}\frac{y_i}{r_i}$$

for each slot where $\sum_{i=0}^{\ell-1}\frac{y_i}{r_i}$ is sufficiently close to the integer v. In particular, the error is less than $\|z\|_\infty/r < 1/\max_{i\in I} p_i$ due to Lemma 1. This checks the first condition of Definition 8. For the second condition, we see that

$$\|\boldsymbol{v}\|_\infty \le \max\left(\prod_{i\in I} p_i\right)\cdot\left|\sum_{i=0}^{\ell-1}\frac{y_i}{r_i}\right| < \frac{\ell}{2}\cdot\prod_{i\in I} p_i \ll \frac{\prod_{0\le i<k} p_i}{\max_{i\notin I} p_i}$$

as desired.

Next, we check if the algorithm correctly evaluates $v = \left\lfloor \sum_{i=0}^{\ell-1} y_i/r_i \right\rceil$. As we are computing in parallel, we describe only computations inside a single slot and illustrate how the underlying message changes through computations.

After the Line 3 of Algorithm 4, we have

$$\left(\left\lfloor\left(\prod_{i\in I} p_i\right)\cdot\frac{y_j}{r_j}\right\rceil + e_j\right)_{0\le j<\ell}$$

for some small error $e = (e_j)_{0\le j<\ell}$. According to Theorem 2, each e_j satisfies

$$|e_j| \le \frac{1}{4}\left(k + \sum_{i=0}^{k-1} p_i\right).$$

Through rotation sum, we get

$$\sum_{j=0}^{\ell-1}\left\lfloor\left(\prod_{i\in I} p_i\right)\cdot\frac{y_j}{r_j}\right\rceil + \sum_{j=0}^{\ell-1} e_j = \left\lfloor\left(\prod_{i\in I} p_i\right)\cdot\sum_{j=0}^{\ell-1}\frac{y_i}{r_j}\right\rceil + e'$$

for some small $e' \in \mathbb{Z}$ upper bounded as

$$|e'| \le \sum_{j=0}^{\ell-1}|e_j| + \ell - 1 < \frac{1}{4}\left(k + 4 + \sum_{i=0}^{k-1} p_i\right)\cdot \ell.$$

Lastly, due to the condition

$$\prod_{i\in I} p_i > \frac{1}{2}\left(k + 4 + \sum_{i=0}^{k-1} p_i\right)\cdot\ell > 2\cdot|e'|,$$

$\mathsf{Rescale}_{p_I}(\mathsf{ct}_v)$ computes the exact rescale so it gives $\lfloor\sum_{j=0}^{\ell-1} y_j/r_j\rceil$ as desired.

The rest of the algorithm is checked by the correctness of individual operations. This finished the proof. □ □

5 Efficiency Analysis

In this section, we discuss the efficiency of the proposed methods and compare them with the previous works.

5.1 Achieving Arbitrary Precision

We first analyze how much precision we can achieve through our approach, and how it compares to the state-of-the-art high-precision works. To start with, we observe the (asymptotic) relations between the moduli we use during (nested) modular computations. In particular, we have the following inequalities.

- $\log(p_i) \leq n$ for some n due to the efficiency of discrete bootstrapping [BKSS24].
- $r_j^2 < \prod_{i=0}^{k-1} p_i / \max_i p_i$ from Theorem 3.
- $s^2 < \min\left(r/\max_{i \in I} p_i, r/\max_{0 \leq j < \ell} r_j\right)$ from Theorem 7.
- $r_j^2 < \prod_{i \notin I} p_i / \max_{0 \leq i < k} p_i$ from Theorem 7.

Roughly speaking, we have

$$\log r = \sum_{j=0}^{\ell-1} \log r_j = \sum_{j=0}^{\ell-1} \sum_{i=0}^{k-1} O(\log p_i) = \sum_{j=0}^{\ell-1} \sum_{i=0}^{k-1} O(n) = O(k\ell n)$$

and

$$\log s = O(\log r) = O(k\ell n).$$

Since the only requirement for $k\ell$ is that $k\ell \leq N$, we have that both $\log r$ and $\log s$ can be as large as $O(N)$. In other words, we can instantiate $O(N)$ bit modular multiplication using a single RLWE ciphertext of dimension N.

We compare this with the previous high-precision FHE methods. For a fair comparison, we mainly compare with the RLWE-based schemes. We start with the original BGV/BFV whose maximum precision is determined by the available modulus $\log QP$ which depends on N. For typical choices of N (e.g. $\log N = 15, 16$), it usually supports up to tens of modulus. To support higher precision, one may consider a radix or CRT version of BGV/BFV (as the CGGI/DM family did in [CLOT21, CKLM24a, CKLM24b]). The CRT-based one might be preferable in terms of computational complexity. In this case, bootstrapping is also naturally supported as we can bootstrap individual ciphertexts to bootstrap the whole ciphertext tuple. However, this scheme is not very efficient as the size of the ciphertext tuple depends on the number of CRT moduli, leading to a very large size for supporting high precision. To overcome this problem, one may consider exploiting the polynomial ring as in CLPX [CLPX18] or its generalization [GV24]. Although they support high precision within a single ciphertext, high-precision bootstrapping is not efficiently supported. In particular in [GV24], they use BFV bootstrapping with large precision as a subroutine which prevents one from instantiating very large precision. On the CKKS side, one may consider the recent radix-based approach from [Kim25]. Similar to the CRT BGV/BFV approach, it supports both high-precision arithmetic and bootstrapping but struggles to achieve good latency due to having a large number of ciphertexts. On the other hand, our method solves all the problems mentioned above: it supports arbitrary precision and bootstrapping within a single RLWE ciphertext. See Table 2 for a detailed comparison.

In addition to what was discussed above, our scheme has a new functionality that all the other approaches did not have, modular arithmetic for arbitrary modulus. The

CRT or radix-based approaches cannot support arbitrary modulus as they use smaller chunks to enable higher precision. CLPX and GBFV also have limited choice of moduli, as they need to be cyclotomic moduli. Furthermore, they cannot use all the cyclotomic moduli options as the type of the modulus affects the performance. On the other hand, we may choose an arbitrary modulus as long as it satisfies the upper bound. In this regard, our scheme enables arbitrary precision and modulus for the first time which can be useful in many applications such as homomorphic signing discussed in Section 1.

5.2 In Moderate Precision

As our method allows us to use as many slots as we want, we may choose a moderate target modulus (e.g. 32 or 64 bits) for modular arithmetic. The first layer without homomorphic modular reduction (i.e. IMod_r) already gives modulo arithmetic over $\mathbb{Z}_p$ where $p = \prod_{i=0}^{k-1} p_i$. On top of the first layer CRT system, we may rely on the modular reduction IMod_r to reduce by an arbitrary modulus. If we need a relatively larger modulus, we may use multiple r_i's to represent a large integer or use JMod_s for modular reducing over large arbitrary modulus. Long story short, all the intermediate steps can be used to instantiate modular integer computations, and these options provide a trade-off between precision and the number of parallelisms.

For simplicity, we discuss the trade-off for the first layer CRT system without IMod_r. The trade-off is naturally extended to higher layers. Recall that we use k moduli $p_0, p_1, \ldots, p_{k-1}$ to instantiate modulo $p = \prod_{i=0}^{k-1} p_i$ arithmetic. As we use k slots for each $\mathbb{Z}_p$ addition and multiplication, we have $\frac{N}{2k}$ many $\mathbb{Z}_p$ slots. The $k = 1$ case corresponds to the small integer modular arithmetic illustrated in [KN24], whereas we may choose k as the number of primes less than or equal to n bits which gives the highest possible modulus in the first layer. Between these two cases are middle grounds, where we can adjust the trade-off between the precision p and the number of slots $\frac{N}{2k}$.

As the radix-based method [Kim25] also provides modular arithmetic for moderate precision, we may compare our method with [Kim25]. There are two key differences between the two approaches. One difference is that our method has better latency due to in-ciphertext parallelization, whereas the radix-based one does not naturally support high-precision computation in a single ciphertext. They may put all the digits inside a single ciphertext and use slot rotations to compute across the digits, but they cannot enjoy full parallelism as the modular reduction algorithm (i.e. denoted as Reduce in [Kim25]) is not directly parallelizable. On the other hand, the CRT structure allows us to evaluate everything completely in parallel, resulting in much better latency. The downside of the CRT-based approach is that it does not directly support moduli of the form d^k which is a standard data type for modern computers. To solve this issue, it is tempting to use the modular reduction IMod_r. Although this provides an extra overhead (i.e. more bootstrapping), the latency of this approach is still plausible as the overhead does not depend on the target precision. Nevertheless, the CRT-based solution does not have the full capability that the radix-based method has, such as efficient comparison, shifting, and arbitrary function evaluation. In this regard, there is a clear trade-off between the radix-based and the CRT-based approaches.

6 Experiments

We provide some proof-of-concept implementations for our method, focusing on the double layer RNS system. We built our code upon the Lattigo [lat24] library in golang. All of our parameters satisfy ≈ 128 bits of security according to [BTPH22]. All of our experiments are run single-threaded on Apple M1 Pro with 32GB of RAM running macOS Sequoia 14.4.1. The running time and precision are measured after at least 20 iterations. Unless stated otherwise, the running time of other works were tested in the same environment. Throughout the section, the precision listed in the tables refers to the logarithm of the size of the underlying error, as a discrete CKKS ciphertext.

6.1 Parameters

We first designed a discrete bootstrapping [BKSS24] parameter set and constructed rest of the parameters upon it. We describe the discrete bootstrapping parameter set we used throughout the implementations in Table 3. This parameter can afford up to 7 multiplicative depths for evaluating look-up tables.

Table 3: Main FHE parameter set used for the experiments. Here N denotes the ring dimension, QP denotes the maximum key switching modulus, $(h, \tilde{h})$ denote the dense and spare Hamming weight for the sparse secret encapsulation [BTPH22], respectively, and $\log Q$ and $\log P$ denote the size and number of primes used for ciphertext and auxiliary modulus, respectively. When written as $X \times Y$, it denotes having Y many X bit moduli. Names in the second row under $\log Q$ denote how many moduli are reserved for each operation.

$\log N$	$\log QP$	$(h, \tilde{h})$	$dnum$	$\log Q$						$\log P$
				Base	StC	Mult	LUT	EvalExp	CtS	
16	1518	$(192, 32)$	5	52	42×3	52×3	52×7	52×8	48×3	52×5

For the first layer modular reduction, we followed the framework of [KN24] while instantiating the homomorphic look-up table with Hermite interpolation. In particular, we chose $p_i \leq 2^6 - 2 = 62$ and the interpolation polynomial degree $\leq 2^7 - 2 = 126$ so that we can evaluate it with 7 multiplicative depths. There are $k = 16$ moduli in the first layer. For the second layer, we chose $r_j \approx 2^{15}$ which satisfies the condition $r_j^2 \ll \min_i(\hat{p_i})$ needed for Theorem 3 as well as the conditions of Algorithm 4. There are $\ell = 64$ moduli in the second layer. The parameter details are specified in Table 4. Note that the base bootstrapping (which enables the modular reduction for the first layer) takes 15.0 seconds. We slightly modify the nested CRT parameters (e.g. k, ℓ, $\log(r_j)$) in the following experiments, but they are kept the same unless stated otherwise.

Table 4: Nested CRT Parameters. Here $\log(p)$ denotes the size of the first layer moduli p_i, $\log(r)$ denotes the size of the second layer moduli r_j, and k, ℓ denote the number of first and second layer moduli, respectively.

$\log(p)$	$\log(r)$	k	ℓ	$\log(\prod_{i=1}^{k-1} p_i)$	$\log(\prod_{j=0}^{\ell-1} r_j)$	$\mathsf{IntMod}_{\{p_i\}_{0\leq i<k}}$ time
6	15	16	64	77.1	960	15.2 sec

6.2 Results and Analysis

We describe our single and double-layer RNS system experiments which follow the algorithms in Sections 3 and 4. Note that each step enables unique functionality, which is not only a subroutine for the higher layers but also a useful operation as it is. In this regard, we start from the very bottom layer and illustrate the functionality of each layer.

First Layer CRT Multiplication. We observe that the asymmetric modular reduction already enables modular arithmetic with relatively smaller modulus. To be more precise, it provides a trade-off between the number of slots and the size of the moduli.

Table 5 shows that executing the same asymmetric modular reduction algorithm, but with different moduli choices, can be used to achieve various plaintext modulus size. The more plaintext moduli we use, the less parallelism we have.

First Layer Modular Reduction. We experimented the first layer modular reduction Mod'_r. It enables $\mathbb{Z}_r \simeq \prod_{j=0}^{\ell-1} \mathbb{Z}_{r_j}$ for 15 bit primes r_j and variable ℓ. This also introduces a trade-off between plaintext modulus size and the number of slots, and the modulus size can vary from 15 bits to $\approx 30\,000$ bits. We chose specific r to handle the later examples. The experimental results are shown in Table 6.

Second Layer Modular Reduction We experimented the second layer modular reduction (denoted as JMod_s) which computes modular arithmetic over a very large s (can be at most $\approx 10\,000$ bits in our setting). The experimental results for homomorphic modular multiplication are shown in Table 7. In particular, we experimented the elliptic curve prime $2^{255} - 19$ and P-384 and a modulus of size 2048 bits, which were motivated in Section 1.

6.3 Comparison with Prior Works

We compare our results with the state-of-the-art methods on homomorphic integer multiplication, namely [GV24], [Kim25], and TFHE-rs [Zam22] which bootstrap in a reasonable amount of time on moderately large (≥ 64 bit) precision. We give the detailed timing numbers in Table 8.

Compared to TFHE-rs [Zam22], our method outperforms their homomorphic multiplication both in terms of latency and throughput. Interestingly, even for a single multiplication it is preferable to use our method although it is known that CGGI/DM schemes provide good latency. For instance, when evaluating 256 bit multiplication, our method is an order of magnitude faster in terms of latency and three orders of magnitude faster in terms of throughput. However, this does not mean that our scheme is better in every aspect, as the radix-based approach they use allows wider range of operations.

Compared to the radix-based CKKS approach in [Kim25], our scheme is better both in terms of latency and throughput. In particular, when evaluating ≥ 64 bit homomorphic multiplication,[6] our latency is more than an order of magnitude faster while keeping the throughput better.

One may compare our result with the state-of-the-art high precision BGV/BFV bootstrapping in [KSS24]. For achieving ≈ 256 bits of precision, our method is an order magnitude faster in terms of latency but two orders of magnitude slower in terms of throughput. This suggests that there is a trade-off between latency and throughput between these methods. When the plaintext modulus becomes even larger (e.g. more than a thousand bit), then our method becomes preferable as they would need larger ring dimension (e.g. $\log N = 17$) and much worse latency (e.g. several hours). Note that the highest plaintext modulus size ever implemented for BGV/BFV bootstrapping was around 234 bits (in [KSS24]), whereas our bootstrapping supports up to $\approx 30,000$ bits in $\log N = 16$.

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

[6] We compare 64 bit multiplication of [Kim25] and 77.1 bit multiplication of ours.

k	$\log(\prod_{i=0}^{k-1} p_i)$	# Slots	Mod_p time	Amortized time	precision
16	77.1	2048	15.2 sec	7.42 ms	-32.0
8	35.5	4096	14.8 sec	3.61 ms	-32.3
4	20.0	8192	14.6 sec	1.78 ms	-32.3
2	9.75	16384	14.2 sec	0.867 ms	-33.6
1	5.00	32768	13.9 sec	0.424 ms	-33.5

Table 5: Asymmetric Modular Reduction gives adjustable precision modular arithmetic. Here we executed $\mathsf{Mod}_p = \mathsf{IntMod}_{\{p_i\}_{0 \le i < k}}$ with varying k and $\{p_i\}$. The precision denotes the maximum of all errors across different slots and iterations, in log base 2.

$\log(r)$	ℓ	# slots	$\mathbb{Z}_r$ mult time		precision
			latency	amortized time	
960	64	32	31.2 sec	975 ms	-26.0
7679	512	4	31.4 sec	7.85 sec	-26.2

Table 6: Modular Homomorphic Multiplication over $\mathbb{Z}_r \simeq \prod_{i=0}^{\ell-1} \mathbb{Z}_{r_i}$, instantiated as multiplication plus Mod'_{r} for each i. The precision denotes the maximum of all errors across different slots and iterations, in log base 2.

$\log(s)$	# slots	$\mathbb{Z}_s$ mult time		precision
		latency	amortized time	
255	32	254 sec	7.94 sec	-24.34
384	32	259 sec	8.09 sec	-24.7
2048	4	330 sec	82.5 sec	-31.9

Table 7: Modular Homomorphic Multiplication over $\mathbb{Z}_s$, instantiated as multiplication plus JMod_s. The precision denotes the maximum of all errors across different slots and iterations, in log base 2. Here the 255 and 384 bit primes are the elliptic curve primes $2^{255} - 19$ and P-384, respectively, and 2048 bit modulus is an RSA modulus. For $\log(s) = 2048$, we added extra cleaning to get sufficient precision, leading to slightly worse latency (but better precision).

	$\log(t)$	# slots	latency	throughput
TFHE-rs [Zam22]	64	1	36.2 sec	36.2 sec
	128		144 sec	144 sec
	256		574 sec	574 sec
[Kim25]	64	16384	220 sec	13.4 ms
This paper	77.1	2048	15.2 sec	7.42 ms
	128	256	33.6 sec	131 ms
	256	128	33.7 sec	263 ms
[KSS24]	168	≤ 65536	128 sec	≥ 1.95 ms
	234	≤ 131072	392 sec	≥ 2.99 ms

Table 8: Comparison with the state-of-the-art integer multiplications. Here $\log(t)$ refers to the logarithm of the plaintext modulus. We compare latency and throughput of an execution of single multiplication plus bootstrapping. Timings for [KSS24] are borrowed from [KSS24, Table 4] which is measured in a similar environment. All other timings are measured in the machine described at the beginning of the section. Note that our $\log(t) = 77.1$ implementation is relying on Mod_p whereas our $\log(t) = 128, 256$ is relying on Mod'_r.

References

[AAB$^+$24] Marius A. Aardal, Diego F. Aranha, Katharina Boudgoust, Sebastian Kolby, and Akira Takahashi. Aggregating falcon signatures with labrador. In *CRYPTO 2024*, volume 14920 of *Lecture Notes in Computer Science*, pages 71–106. Springer, 2024.

[ABPS24] Shahla Atapoor, Karim Baghery, Hilder V. L. Pereira, and Jannik Spiessens. Verifiable FHE via lattice-based SNARKs. *IACR Communications in Cryptology*, 1(1), 2024.

[ADE$^+$23] E. Aharoni, N. Drucker, G. Ezov, E. Kushnir, H. Shaul, and O. Soceanu. E2E near-standard and practical authenticated transciphering. Cryptology ePrint Archive, Paper 2023/1040, 2023.

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

[BBB$^+$23] L. Bergerat, A. Boudi, Q. Bourgerie, I. Chillotti, D. Ligier, J.-B. Orfila, and S. Tap. Parameter optimization & larger precision for (t)fhe. *J. Cryptol.*, 2023.

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

[Ber06] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In *Public Key Cryptography - PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006.

[BGG$^+$18] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In *CRYPTO 2018*, volume 10991 of *Lecture Notes in Computer Science*, pages 565–596. Springer, 2018.

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

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

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

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

[BS23] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography (version 0.6)*. January 2023. Available at `https://toc.cryptobook.us/`.

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

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

[Cha82] David Chaum. Blind signatures for untraceable payments. In *CRYPTO '82*, pages 199–203. Plenum Press, New York, 1982.

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

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

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

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

[CKLM24a] P. Chartier, M. Koskas, M. Lemou, and F. Méhats. Fully homomorphic encryption on large integers. Cryptology ePrint Archive, Paper 2024/155, 2024.

[CKLM24b] P. Chartier, M. Koskas, M. Lemou, and F. Méhats. Homomorphic sign evaluation with a RNS representation of integers. In *ASIACRYPT*, 2024.

[CLOT21] I. Chillotti, D. Ligier, J.-B. Orfila, and S. Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In *ASIACRYPT*, 2021.

[CLPX18] H. Chen, K. Laine, R. Player, and Y. Xia. High-precision arithmetic in homomorphic encryption. In *CT-RSA*, 2018.

[DKLS24] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Threshold ECDSA in three rounds. In *IEEE Symposium on Security and Privacy*, pages 3053–3071. IEEE, 2024.

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

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

[EY24] Ehsan Ebrahimi and Anshu Yadav. Strongly secure universal thresholdizer. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024*, volume 15486 of *Lecture Notes in Computer Science*, pages 207–239. Springer, 2024.

[Fis06] Marc Fischlin. Round-optimal composable blind signatures in the common reference string model. In *CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 60–77. Springer, 2006.

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

[FW24] Georg Fuchsbauer and Mathias Wolf. Concurrently secure blind schnorr signatures. In *EUROCRYPT 2024*, volume 14652 of *Lecture Notes in Computer Science*, pages 124–160. Springer, 2024.

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

[GHR99] Rosario Gennaro, Shai Halevi, and Tal Rabin. Secure hash-and-sign signatures without the random oracle. In *EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 123–139. Springer, 1999.

[GV24] R. Geelen and F. Vercauteren. Fully homomorphic encryption for cyclotomic prime moduli. Cryptology ePrint Archive, Paper 2024/1587, 2024.

[HPS19] S. Halevi, Y. Polyakov, and V. Shoup. An improved rns variant of the bfv homomorphic encryption scheme. In *CT-RSA*, 2019.

[KG20] Chelsea Komlo and Ian Goldberg. FROST: flexible round-optimized schnorr threshold signatures. In *Selected Areas in Cryptography*, volume 12804 of *Lecture Notes in Computer Science*, pages 34–65. Springer, 2020.

[Kim25] J. Kim. Efficient homomorphic integer computer from CKKS. Cryptology ePrint Archive, Paper 2025/066, 2025.

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

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

[KVMH24] Christian Knabenhans, Alexander Viand, Antonio Merino-Gallardo, and Anwar Hithnawi. vfhe: Verifiable fully homomorphic encryption. In *Proc. of the 12th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 11–22. ACM, 2024.

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

[Lin22] Yehuda Lindell. Simple three-round multiparty schnorr signing with full simulatability. Cryptology ePrint Archive, Paper 2022/374, 2022.

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

[LMSS22] B. Li, D. Micciancio, M. Schultz, and J. Sorrell. Securing approximate homomorphic encryption using differential privacy. In *CRYPTO*, 2022.

[QCY21] Xianrui Qin, Cailing Cai, and Tsz Hon Yuen. One-more unforgeability of blind ECDSA. In *ESORICS 2021*, volume 12973 of *Lecture Notes in Computer Science*, pages 313–331. Springer, 2021.

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