Approximate Homomorphic Encryption with Reduced Approximation Error

Andrey Kim[1], Antonis Papadimitriou[2], and Yuriy Polyakov[2]

[1]NJIT
[2]Duality Technologies

September 21, 2020

Abstract

The Cheon-Kim-Kim-Song (CKKS) homomorphic encryption scheme is currently the most efficient method to perform approximate homomorphic computations over real and complex numbers. Although the CKKS scheme can already be used to achieve practical performance for many advanced applications, e.g., in machine learning, its broader use in practice is hindered by several major usability issues, most of which are brought about by relatively high approximation errors and the complexity of dealing with them.

We present a reduced-error CKKS variant that removes the approximation errors due to the Learning With Errors (LWE) noise in the encryption and key switching operations. We also propose and implement its RNS instantiation that has a lower error than the original CKKS scheme implementation based on multiprecision integer arithmetic. While formulating the RNS instantiation, we develop an intermediate RNS variant that has a smaller approximation error than the prior RNS variant of CKKS. The high-level idea of our main RNS-specific improvements is to remove the approximate scaling error using an automated procedure that computes different scaling factors for each level and performs all necessary adjustments. The rescaling procedure and scaling factor adjustments in our implementation are done automatically and are not exposed to the application developer.

We implement both RNS variants in PALISADE and compare their approximation error and efficiency to the prior RNS variant. Our results for uniform ternary secret key distribution, which is the most efficient setting included in the community homomorphic encryption security standard, show that the reduced-error CKKS RNS implementation typically has an approximation error that is 6 to 9 bits smaller for computations with multiplications than the prior RNS variant. For computations without a multiplication, the approximation error can be up to 20 bits lower than in the prior RNS variant. As compared to the original CKKS using multiprecision integer arithmetic, our reduced-error CKKS RNS implementation has an error that is smaller by 4 and up to 20 bits for computations with multiplications and without multiplications, respectively. For the sparse ternary secret key setting, which was used in the original CKKS paper, the approximate error reduction of reduced-error CKKS w.r.t. original CKKS typically ranges from 6 to 8 bits for computations with multiplications.

Contents

1 Introduction

The Cheon-Kim-Kim-Song (CKKS) homomorphic encryption (HE) scheme is currently the most efficient method to perform approximate homomorphic computations over real and complex numbers [13]. The CKKS scheme can already be used to achieve practical performance for many advanced applications, e.g., in machine learning for genomics [4,5,22,23]. Its broader use in practice is hindered by several major usability issues. One of the main challenges is the approximation error inherent to almost every operation in CKKS. A significant error is introduced during encryption and keeps growing as computations are performed. To minimize the growth of approximation error, the original CKKS scheme introduced a rescaling operation [13]. But the rescaling operation brought about several other usability issues, e.g., the need for a user to decide when rescaling should be called to achieve desired precision and optimize the efficiency. Another major challenge is specific to the rescaling approximation error in the Residue Number System (RNS) variants of CKKS, which are preferred in practice for better efficiency [5,10].

Approximation errors in CKKS. All approximation errors in both multiprecision and RNS CKKS are summarized in Table 1. Here, we briefly describe each approximation error.

Table 1: Approximation errors in the original CKKS and prior RNS CKKS vs our variants of CKKS and RNS CKKS. The errors $\boldsymbol{r}_{\mathsf{encode}}$, $\boldsymbol{e}_{\mathsf{fresh}}$, and $\boldsymbol{e}_{\mathsf{ks}}$ in our variants get scaled down by Δ (Δ_ℓ), and hence their contribution becomes negligible. In reduced-error CKKS, the dominant source of approximation error is $\boldsymbol{r}_{\mathsf{rs}}$. The addition of existing error $\boldsymbol{f}$ in unary operations is omitted for brevity.

	Errors in CKKS		Errors in RNS CKKS	
Algorithm	Original CKKS [13]	Ours	Prior RNS CKKS [5,10]	Ours
Encode	$\boldsymbol{r}_{\mathsf{encode}}$, $\boldsymbol{r}_{\mathsf{float}}$	$\boldsymbol{r}_{\mathsf{float}}$	$\boldsymbol{r}_{\mathsf{encode}}$, $\boldsymbol{r}_{\mathsf{float}}$	$\boldsymbol{r}_{\mathsf{float}}$
Encrypt	$\boldsymbol{e}_{\mathsf{fresh}}$	-	$\boldsymbol{e}_{\mathsf{fresh}}$	-
Add	$\boldsymbol{f}_+ = \boldsymbol{f}_1 + \boldsymbol{f}_2$	$\boldsymbol{f}_+$	$\boldsymbol{f}_+$	$\boldsymbol{f}_+$
Mult.	$\frac{\boldsymbol{f}_\times}{\Delta} \approx \frac{\boldsymbol{m}_2\boldsymbol{f}_1+\boldsymbol{m}_1\boldsymbol{f}_2+\boldsymbol{e}_{\mathsf{ks}}}{\Delta}$	$\frac{\boldsymbol{f}_\times}{\Delta}$	$\frac{\boldsymbol{f}_\times}{\Delta_\ell}$	$\frac{\boldsymbol{f}_\times}{\Delta_\ell}$
Automorphism	$\boldsymbol{e}_{\mathsf{ks}}$	-	$\boldsymbol{e}_{\mathsf{ks}}$	-
Rescale	$\boldsymbol{r}_{\mathsf{rs}}$	$\boldsymbol{r}_{\mathsf{rs}}$	$\boldsymbol{r}_{\mathsf{rs}}$, $\boldsymbol{u}_\Delta$	$\boldsymbol{r}_{\mathsf{rs}}$
Decrypt	-	-	-	-
Decode	$\boldsymbol{r}_{\mathsf{float}}$	$\boldsymbol{r}_{\mathsf{float}}$	$\boldsymbol{r}_{\mathsf{float}}$	$\boldsymbol{r}_{\mathsf{float}}$
Scalar Add	$\boldsymbol{f} + \boldsymbol{r}_{\mathsf{encode}}$, $\boldsymbol{r}_{\mathsf{float}}$	$\boldsymbol{f}$, $\boldsymbol{r}_{\mathsf{float}}$	$\boldsymbol{f} + \boldsymbol{r}_{\mathsf{encode}}$, $\boldsymbol{r}_{\mathsf{float}}$	$\boldsymbol{f}$, $\boldsymbol{r}_{\mathsf{float}}$
Scalar Mult.	$\boldsymbol{f}_{\times\mathsf{c}}/\Delta \approx \frac{\boldsymbol{m}_{\mathsf{c}}\boldsymbol{f}+\boldsymbol{m}\boldsymbol{r}_{\mathsf{encode}}}{\Delta}$	$\boldsymbol{f}_{\times\mathsf{c}}/\Delta$	$\boldsymbol{f}_{\times\mathsf{c}}/\Delta_\ell$	$\boldsymbol{f}_{\times\mathsf{c}}/\Delta_\ell$
Crosslevel Add	$\boldsymbol{f}_+$	$\boldsymbol{f}_+$	$\boldsymbol{f}_+$, $\boldsymbol{u}_\Delta$	$\boldsymbol{f}_{1,\times\mathsf{c}} + \boldsymbol{f}_2$
Crosslevel Mult.	$\boldsymbol{f}_\times/\Delta$	$\boldsymbol{f}_\times/\Delta$	$\boldsymbol{f}_\times/\Delta_\ell$, $\boldsymbol{u}_\Delta$	$\approx \frac{\boldsymbol{m}_2\boldsymbol{f}_{1,\times\mathsf{c}}+\boldsymbol{m}_1\boldsymbol{f}_2+\boldsymbol{e}_{\mathsf{ks}}}{\Delta_\ell}$

The security of the CKKS scheme is based on the Ring Learning With Errors (RLWE) problem, where Gaussian noise is introduced to achieve the desired hardness properties [13]. In the case of CKKS, this LWE noise modifies the least significant bits of the plaintext during encryption, hence resulting in a lossy encryption scheme. If the ciphertext ct encrypts a plaintext $\boldsymbol{m}$, the decryption of ct outputs a noisy result $\tilde{\boldsymbol{m}} = \boldsymbol{m} + \boldsymbol{f}$. The central problem in CKKS is to keep the error $\boldsymbol{f}$ relatively small to meet the desired precision requirements. We will refer to this type of approximation error as an LWE approximation error. The LWE approximation errors are introduced during encryption and key switching, and will be denoted as $\boldsymbol{e}_{\mathsf{fresh}}$ and $\boldsymbol{e}_{\mathsf{ks}}$, respectively.

For leveled HE schemes, there is another source of noise related to the integer-division rounding during the modulus switching operation. This noise depends on the norm of the secret key. In CKKS, modulus switching is called rescaling as it effectively rescales the underlying encrypted plaintext and drops a certain number of least significant bits from the message. Due to the lossy nature of CKKS, this rescaling noise brings about an approximation error. We call this error as a rescaling rounding error, and denote it by $\boldsymbol{r}_{\mathsf{rs}}$. There is another related procedure in CKKS called modular reduction, which does modulus switching without scaling the encrypted message (or noise). This operation does not introduce any noise/approximation error, and is not included in Table 1.

Besides LWE and rescaling rounding errors, there are other sources of errors that contribute to the output approximation error in the CKKS scheme. In the encoding and decoding procedures, these sources of error arise from precision limitations, e.g., if using double to represent real numbers. We call these errors as precision errors and will denote them as $\boldsymbol{r}_{\mathsf{float}}$. Precision errors can be reduced by increasing the floating-point precision in computations. The encoding procedure also includes another rounding error caused by converting (rounding) encoded real-number plaintexts to integer plaintexts. We will call this error $\boldsymbol{r}_{\mathsf{encode}}$.

The RNS variants of CKKS introduce another approximation error caused by approximate scaling in the rescale operation. The RNS variants use a chain of small primes q_i that are only approximately close to the scaling factor $\Delta = 2^p$, and the differences between q_i and 2^p bring about this approximation error, which will be denoted as $\boldsymbol{u}_{\Delta}$. This error is typically few bits higher than the LWE approximation error, and hence the RNS variants have a lower precision than the multiprecision integer instantiation of CKKS.

Addition and multiplication essentially add up approximation errors of both input ciphertexts, resulting in an increased approximation error in the output ciphertext by at most 1 bit (in the worst case of two correlated ciphertexts). There are also somewhat special types of addition/multiplication called scalar and crosslevel addition/multiplication. Their approximation errors are shown in Table 1 and explained in more detail further in the paper.

To better understand the contribution of our work, note that $\boldsymbol{u}_{\Delta} > \{\boldsymbol{e}_{\mathsf{fresh}}, \boldsymbol{e}_{\mathsf{ks}}\} > \boldsymbol{r}_{\mathsf{rs}}$. We intend to remove $\boldsymbol{u}_{\Delta}$, $\boldsymbol{e}_{\mathsf{fresh}}$, and $\boldsymbol{e}_{\mathsf{ks}}$, hence effectively reducing the output approximation error to the rescaling rounding error $\boldsymbol{r}_{\mathsf{rs}}$ and its accumulation from multiple ciphertexts.

Our work. The main goal of our work is to modify the CKKS scheme and its RNS variants to systematically remove many of the approximation errors listed in Table 1, achieving a significantly reduced output approximation error and improving the overall usability of the scheme.

Our first idea is to redefine the multiplication operation in CKKS as

$$\mathsf{ct}_{\mathsf{mult}'} = \mathsf{Mult}'(\mathsf{ct}_1, \mathsf{ct}_2) = \mathsf{Mult}\left(\mathsf{Rescale}(\mathsf{ct}_1, \Delta), \mathsf{Rescale}(\mathsf{ct}_2, \Delta)\right).$$

Reordering the rescaling and multiplication operations this way, i.e., reversing the order of multiplication and rescaling in the original CKKS scheme, brings about several benefits. First, if we rescale before the first multiplication, we can remove (scale down) the prior encoding approximation errors, the LWE encryption approximation error, and any addition and key switching approximation errors if these operations are performed before the first multiplication. If we decrypt the ciphertext before the first multiplication, i.e., in computations without multiplications, we will only observe the effect of the floating-point precision error $\boldsymbol{r}_{\mathsf{float}}$, which for the case of double-precision floating-point numbers (52 bits of precision) would typically be about 48-50 bits. Second, delaying the rescaling operation until the following multiplication (in computations with multiplications) enables us to eliminate key-switching approximation errors. The only approximation errors that are left in the non-RNS CKKS are the rescaling rounding error $\boldsymbol{r}_{\mathsf{rs}}$, accumulated error due to additions (after first multiplication) and multiplications, and a relatively small floating-point precision error $\boldsymbol{r}_{\mathsf{float}}$.

Our second idea is to redefine the rescaling operation in RNS by introducing different scaling factors Δ_ℓ at each level to eliminate the approximate scaling error $\boldsymbol{u}_\Delta$. The main algorithmic challenges in the implementation of this idea are related to handling various computation paths, such as adding two ciphertexts that are several levels apart (referred to as *crosslevel* addition), and finding the prime moduli q_i that do not lead to the divergence of the level-specific scaling factor towards zero or infinity for deeper computations. While addressing these challenges, we also restrict (automate) rescaling to being done right before multiplication (following our definition of Mult'). We also redefine the addition operation to include a scalar multiplication and rescaling to bring two ciphertexts to the same scaling factor. Though this appears to make the CKKS algorithms more complex, we fully automate these procedures in our software implementation, achieving the same practical precision as in the non-RNS CKKS instantiation, as seen in Table 1.

We also provide an efficient implementation of our reduced-error (RE) CKKS variant in RNS along with an intermediate RNS variant that is faster, but at the expense of increasing the output approximation error. Table 2 shows representative results for four different benchmarks: addition of multiple vectors, summation over a vector, binary tree multiplication, and evaluation of a polynomial over a vector. These results suggest that the reduced-error CKKS RNS implementation has an approximation error around 7 bits smaller (we observed values in the range from 6 to 9 bits) for computations with multiplications than the prior RNS variant. For computations without a multiplication, the approximation error can be up to 20 bits lower than in the prior RNS variant. As compared to the original CKKS using multiprecision integer arithmetic (which is equivalent in precision to our RNS variant with delayed exact rescaling), our reduced-error CKKS RNS implementation has an error that is smaller by about 4 and up to 20 bits for computations with multiplications and without multiplications, respectively. Performance results in Section 5 demonstrate that the runtime of our RE-CKKS RNS implementation is typically at most 2x slower than the prior RNS variant, which is a relatively small cost paid for the increased precision. For comparison, the runtime improvement of RNS-HEEAN over the multiprecision HEAAN implementation was 8.3 times for multiplication [10], and the precision gain of the multiprecision HEAAN implementation over RNS-HEAAN is only half of what we report in our work.

Table 2: Representative results showing the precision of our RE-CKKS RNS implementation vs original CKKS and prior CKKS RNS variant for the HE-standard-compliant setting of uniform ternary secrets; $\Delta_i \approx 2^{40}$.

Computation	Prior CKKS RNS [5, 10]	CKKS [13]	RE-CKKS RNS (our work)
$\sum_{i=0}^{32} \mathbf{x}_i$	23.9	23.9	43.8
$\sum_{i=0}^{2048} x_i$	21.1	21.1	40.4
$\prod_{i=1}^{16} \mathbf{x}_i$	17.8	22.4	26.0
$\sum_{i=0}^{64} \mathbf{x}^i$	14.9	17.4	21.3

Table 3: Representative results showing the precision of our RE-CKKS RNS implementation vs original CKKS and prior CKKS RNS variant for sparse ternary secrets (this setting was used in the original CKKS construction [13]); $\Delta_i \approx 2^{40}$.

Computation	Prior CKKS RNS [5, 10]	CKKS [13]	RE-CKKS RNS (our work)
$\sum_{i=0}^{32} \mathbf{x}_i$	24.6	24.6	44.6
$\sum_{i=0}^{2048} x_i$	22.1	22.1	42.0
$\prod_{i=1}^{16} \mathbf{x}_i$	17.8	23.2	29.7
$\sum_{i=0}^{64} \mathbf{x}^i$	14.9	18.2	25.0

Although the original CKKS scheme was instantiated for sparse ternary secrets [13], we use uniform ternary secrets as the main setting in our work because the sparse secrets are not currently included in the homomorphic encryption security standard [2], and hybrid attacks specific to the sparse setting were recently devised [16, 25]. This choice has a direct effect on the precision gain one gets from our RE-CKKS variant. Our theoretical estimates suggest that in the sparse setting the precision gain for a computation with multplications becomes about 6-8 bits (higher than 4 bits that we observe for uniform ternary secrets). Some representative experimental results for the sparse setting, which align with our theoretical estimates, are illustrated in Table 3. Note that the precision gain of our RE-CKKS RNS implementation gets as high as 12 bits over the prior RNS variant.

We also implemented RE-CKKS in the HEAAN library [12], which uses multiprecision arithmetic for rescaling, and ran precision experiments there for selected computations. The observed precision improvement of RE-CKKS over CKKS [13] was approximately the same (within 0.2 bits) as in our PALISADE implementation.

Contributions. Our contributions can be summarized as follows:

- We propose a reduced-error variant of CKKS that reduces the approximation compared to the original CKKS scheme by 4 bits for computations with multiplications and up to 20 bits for computations without multiplications. The main idea of our modifications is to redefine the multiplication operation by "reversing" the order of multiplication and rescaling.

- We adapt this variant to RNS, while keeping the precision roughly the same, by developing a procedure that automatically computes different scaling factors for each level and performs rescaling automatically. This procedure required a development of an original algorithm for finding the RNS primes that keep the scaling factor as close to the starting value as possible, thus preventing the divergence of the scaling factor towards zero or infinity for practical numbers of levels. The procedure also required several algorithms for handling ciphertexts at different levels.

- While developing the RNS variant of reduced-error CKKS, we propose an intermediate RNS variant that has a higher approximation error but runs faster. Both of our RNS variants have errors that are lower than the prior RNS variant [5, 10].

- We implement both RNS variants in PALISADE and make them publicly available.

Related Work. The CKKS scheme was originally proposed in [13] and implemented in the HEEAN library [12] using a mixture of multiprecision and RNS arithmetic. The main drawback in the original implementation was the use of multiprecision integer arithmetic for rescaling and some other operations, which is in practice less efficient than the so-called RNS variants [3, 19]. Then several homomorphic encryption libraries independently developed and implemented RNS variants of CKKS, including RNS-HEAAN [11], PALISADE [1], SEAL [24], and HELib [20]. The typical RNS variant [5, 10], which is based an approximate rescaling, works with small primes q_i that are only approximately close to the actual scaling factor, which introduces an approximation error that is higher than the LWE error present in the original CKKS and its HEAAN implementation. The main differences between various RNS variants is in how key switching is done, e.g., RNS-HEAAN and HELib use the GHS technique [18], SEAL uses a special version of the hybrid key switching [21] and also supports residue decomposition [3], PALISADE supports all of these key switching techniques. The documentation of the SEAL library also mentioned the idea of using different scaling factors for each level but did not provide any (automated) procedure to work with different scaling factors in practice (our paper shows that this can be very challenging and requires the development of new algorithms). Up to this point, it has been widely believed that CKKS in RNS is not practically usable because of many sources of approximation errors and the complexity of dealing with them [26].

Cohen et al. explored the idea of reducing the LWE error in CKKS by using fault-tolerant computations over the reals [14]. The high-level idea is to run multiple computations for the same encrypted values and then compute the average. While this is theoretically possible, the practical performance costs would be high enough to make this approach impractical. In contrast, our idea of rescaling before multiplication has a very small performance cost compared to this approach.

1.1 Organization

The rest of the paper is organized as follows. Section 2 provides the necessary background on the original CKKS scheme and its RNS instantiation. Section 3 describes our reduced-error CKKS variant. Section 4 details our RNS instantiation of the reduced-error CKKS variant, focusing on RNS-specific algorithms. Section 5 discusses implementation details and experimental results. Section 6 concludes the paper.

2 Preliminaries

All logarithms are base 2 unless otherwise indicated. For complex z, we denote by $\|z\|_2 = \sqrt{z\bar{z}}$ its ℓ_2 norm. For an integer Q, we identify the ring $\mathbb{Z}_Q$ with $(-Q/2, Q/2]$ as a representative interval. For a power-of-two N, we denote cyclotomic rings $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$, $\mathcal{S} = \mathbb{R}[X]/(X^N + 1)$, and $\mathcal{R}_Q := \mathcal{R}/Q\mathcal{R}$. Ring elements are in bold, e.g. $\boldsymbol{a}$.

We use $\boldsymbol{a} \leftarrow \chi$ to denote the sampling of $\boldsymbol{a}$ according to a distribution χ. The distribution χ is called *uniform ternary* if all the coefficients of $\boldsymbol{a} \leftarrow \chi$ are selected uniformly from $\{-1, 0, 1\}$. This distribution is commonly used for secret key generation as it is the most efficient option conforming to the HE standard [2]. A *sparse ternary* distribution corresponds to the case when h coefficients are randomly chosen to be non-zero and all others are set to zero, where h is the Hamming weight. The sparse ternary secret distribution was used in the original CKKS scheme [13]. We say that the distribution χ is *discrete Gaussian* with standard deviation σ if all coefficients of $\boldsymbol{a} \leftarrow \chi$ are selected from discrete Gaussian distribution with standard deviation σ. Discrete Gaussian distribution is commonly used to generate error polynomials to meet the desired hardness requirement [2].

For radix base ω, let us define the decomposition of $\boldsymbol{a} \in \mathcal{R}_{Q_\ell}$ by $\mathcal{WD}_\ell(\boldsymbol{a})$ and powers of ω, $\mathcal{PW}_\ell(\boldsymbol{a})$. Let $\mathsf{dnum} = \lceil \log_\omega(Q_\ell) \rceil$, then for $\boldsymbol{a} \in \mathcal{R}_{Q_\ell}$:

$$\mathcal{WD}_\ell(\boldsymbol{a}) = \left([\boldsymbol{a}]_\omega, \left[\left\lfloor \frac{\boldsymbol{a}}{\omega} \right\rceil \right]_\omega, \ldots, \left[\left\lfloor \frac{\boldsymbol{a}}{\omega^{\mathsf{dnum}-1}} \right\rceil \right]_\omega \right) \in \mathcal{R}^{\mathsf{dnum}},$$

$$\mathcal{PW}_\ell(\boldsymbol{a}) = \left([\boldsymbol{a}]_{Q_\ell}, [\boldsymbol{a} \cdot \omega]_{Q_\ell}, \ldots, [\boldsymbol{a} \cdot \omega^{\mathsf{dnum}-1}]_{Q_\ell} \right) \in \mathcal{R}_{Q_\ell}^{\mathsf{dnum}}.$$

For any $(\boldsymbol{a}, \boldsymbol{b}) \in \mathcal{R}_\ell^2$, $\mathcal{WD}_\ell$ and $\mathcal{PW}_\ell$ satisfy the following congruence relation:

$$\langle \mathcal{WD}_\ell(\boldsymbol{a}), \mathcal{PW}_\ell(\boldsymbol{b}) \rangle \equiv \boldsymbol{a} \cdot \boldsymbol{b} \pmod{Q_\ell}.$$

2.1 CKKS Scheme

The original CKKS scheme is formulated for cyclotomic polynomial rings $\mathcal{R} = \mathbb{Z}[X]/\langle X^N + 1 \rangle$, where N is a ring dimension that is a power of two [1]. With a scaling factor $\Delta = 2^p$ and a zero-level modulus $q_0 = 2^{p_0}$ (usually q_0 is taken to be larger than Δ for correct decryption), a modulus at the level ℓ is typically defined as $Q_\ell = 2^{p_0 + \ell \cdot p} = q_0 \cdot \Delta^\ell$, i.e., the scheme works with residue rings $\mathcal{R}_{Q_\ell} = \mathcal{R}/Q_\ell\mathcal{R} = \mathbb{Z}_{Q_\ell}[X]/\langle X^N + 1 \rangle$. We denote $M = 2N$, and by $\mathbb{Z}_M^* = \{x \in \mathbb{Z}_M : \mathsf{gcd}(x, M) = 1\}$ the unit multiplication group in $\mathbb{Z}_M$. The canonical embedding $\tau : \mathcal{S} \to \mathbb{C}^N$ is defined as $\tau(\boldsymbol{a}) = \left(\boldsymbol{a}(\zeta^j) \right)_{j \in \mathbb{Z}_M^*}$ for $\zeta = \exp(2\pi i / M)$. It's ℓ_∞-norm is called the *canonical embedding norm* and is denoted as $\|\boldsymbol{a}\|^{\mathsf{can}} = \|\tau(\boldsymbol{a})\|_\infty$. For a power-of-two $n \le N/2$, we also define mappings $\tau_n' : \mathcal{S} \to \mathbb{C}^n$ used to encode and decode a vector of length n in the CKKS scheme [9, 13]. The algorithms are [13, 21]:

- $\mathsf{Setup}(1^\lambda)$. For an integer $L \ge 0$ that corresponds to the largest ciphertext modulus level, given the security parameter λ, output the ring dimension N. Set the small distributions χ_{key}, χ_{err}, and χ_{enc} over $\mathcal{R}$ for secret, error, and encryption, respectively.
- KeyGen. Sample a secret $\boldsymbol{s} \leftarrow \chi_{\mathsf{key}}$, a random $\boldsymbol{a} \to \mathcal{R}_{Q_L}$, and error $\boldsymbol{e} \leftarrow \chi_{\mathsf{err}}$. Set the secret key $\mathsf{sk} \leftarrow (1, \boldsymbol{s})$ and public key $\mathsf{pk} \leftarrow (\boldsymbol{b}, \boldsymbol{a}) \in \mathcal{R}_{Q_L}^2$, where $b \leftarrow -\boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e} \pmod{Q_L}$.

The hybrid key switching [21] is selected because it is more efficient than the GHS approach used in the original CKKS scheme [10, 13] and incurs a smaller approximation error than the digit decomposition approach [8] for relatively large digits, which are often required for the efficient instantiation of this key switching method.

- $\mathsf{KeySwitchGen}_{\mathsf{sk}}(\boldsymbol{s}')$. For a power-of-two P that corresponds to the auxiliary modulus, sample a random $\boldsymbol{a}_k' \leftarrow \mathcal{R}_{PQ_L}$ and error $\boldsymbol{e}_k' \leftarrow \chi_{\mathsf{err}}$. For a predefined power-of-two base ω, output the switching key as

$$\mathsf{swk} = (\mathsf{swk}_0, \mathsf{swk}_1) = \left(\{\boldsymbol{b}_k'\}_{k=0}^{\mathsf{dnum}-1}, \{\boldsymbol{a}_k'\}_{k=0}^{\mathsf{dnum}-1} \right) \in \mathcal{R}_{PQ_L}^{2 \times \mathsf{dnum}},$$

  where

$$\boldsymbol{b}_k' \leftarrow -\boldsymbol{a}_k' \cdot \boldsymbol{s} + \boldsymbol{e}_k' + P \cdot \mathcal{PW}_L(\boldsymbol{s}')_k \pmod{PQ_L}.$$

  and $\mathsf{dnum} = \lceil \log_\omega(Q_L) \rceil$. Set $\mathsf{evk} \leftarrow \mathsf{KeySwitchGen}_{\mathsf{sk}}(\boldsymbol{s}^2)$. Set $\mathsf{rk}^{(\kappa)} \leftarrow \mathsf{KeySwitchGen}_{\mathsf{sk}}(\boldsymbol{s}^{(\kappa)})$.

[1] CKKS also supports general cyclotomic rings but they are typically less efficient.

- $\mathsf{KeySwitch}_{\mathsf{swk}}(\mathsf{ct})$. For $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}^2_{Q_\ell}$, $\mathsf{swk} = (\mathsf{swk}_0, \mathsf{swk}_1)$ [2] output

$$\left(\boldsymbol{c}_0 + \left\lceil \frac{\langle \mathcal{WD}_\ell(\boldsymbol{c}_1), \mathsf{swk}_0 \rangle}{P} \right\rfloor, \left\lceil \frac{\langle \mathcal{WD}_\ell(\boldsymbol{c}_1), \mathsf{swk}_1 \rangle}{P} \right\rfloor\right) \pmod{Q_\ell}.$$

To keep the noise from key switching small, we can take $P \approx \omega$.

- $\mathsf{Enc}_{\mathsf{pk}}(\boldsymbol{m})$. For $\boldsymbol{m} \in \mathcal{R}$, sample $\boldsymbol{v} \leftarrow \chi_{\mathsf{enc}}$ and $\boldsymbol{e}_0, \boldsymbol{e}_1 \leftarrow \chi_{\mathsf{err}}$. Output $\mathsf{ct} \leftarrow \boldsymbol{v} \cdot \mathsf{pk} + (\boldsymbol{m} + \boldsymbol{e}_0, \boldsymbol{e}_1) \pmod{Q_L}$.
- $\mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct})$. For $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}^2_{Q_\ell}$, output $\tilde{\boldsymbol{m}} = \boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} \pmod{Q_\ell}$.
- $\mathsf{CAdd}(\mathsf{ct}, x)$. For $\mathsf{ct} = (\boldsymbol{b}, \boldsymbol{a}) \in \mathcal{R}^2_{Q_\ell}$ with scaling factor $\Delta^{\ell'}$ and scalar $x \in \mathbb{C}^n$, first encode x with same scaling factor $\boldsymbol{m} = \mathsf{Encode}(x, \Delta^{\ell'})$, and output $\mathsf{ct}_{\mathsf{cadd}} \leftarrow (\boldsymbol{b} + \boldsymbol{m}, \boldsymbol{a}) \pmod{Q_\ell}$.
- $\mathsf{Add}(\mathsf{ct}_1, \mathsf{ct}_2)$. For $\mathsf{ct}_1, \mathsf{ct}_2 \in \mathcal{R}^2_{Q_\ell}$, output $\mathsf{ct}_{\mathsf{add}} \leftarrow \mathsf{ct}_1 + \mathsf{ct}_2 \pmod{Q_\ell}$.
- $\mathsf{CMult}(\mathsf{ct}, x)$. For $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}^2_{Q_\ell}$ and scalar $x \in \mathbb{C}^n$, first encode x, $\boldsymbol{m} = \mathsf{Encode}(x, \Delta)$ and output $\mathsf{ct}_{\mathsf{cmult}} \leftarrow (\boldsymbol{c}_0 \cdot \boldsymbol{m}, \boldsymbol{c}_1 \cdot \boldsymbol{m}) \pmod{Q_\ell}$.
- $\mathsf{Mult}_{\mathsf{evk}}(\mathsf{ct}_1, \mathsf{ct}_2)$. For $\mathsf{ct}_i = (\boldsymbol{b}_i, \boldsymbol{a}_i) \in \mathcal{R}^2_{Q_\ell}$, let $(\boldsymbol{d}_0, \boldsymbol{d}_1, \boldsymbol{d}_2) = (\boldsymbol{b}_1 \cdot \boldsymbol{b}_2, \boldsymbol{a}_1 \cdot \boldsymbol{b}_2 + \boldsymbol{a}_2 \cdot \boldsymbol{b}_1, \boldsymbol{a}_1 \cdot \boldsymbol{a}_2) \pmod{Q_\ell}$. Output

$$\mathsf{ct}_{\mathsf{mult}} \leftarrow (\boldsymbol{d}_0, \boldsymbol{d}_1) + \mathsf{KeySwitch}_{\mathsf{evk}}(0, \boldsymbol{d}_2) \pmod{Q_\ell}.$$

- $\mathsf{Aut}_{\mathsf{rk}^{(\kappa)}}(\mathsf{ct}, \kappa)$. For $\mathsf{ct} = (\boldsymbol{b}, \boldsymbol{a}) \in \mathcal{R}^2_{Q_\ell}$ and automorphism index κ, output

$$\mathsf{ct}_{\mathsf{aut}} \leftarrow (\boldsymbol{b}^{(\kappa)}, 0) + \mathsf{KeySwitch}_{\mathsf{rk}^{(\kappa)}}(0, \boldsymbol{a}^{(\kappa)}) \pmod{Q_\ell}.$$

- $\mathsf{Rescale}(\mathsf{ct}, \Delta^{\ell'})$. For a ciphertext $\mathsf{ct} \in \mathcal{R}^2_{Q_\ell}$ and a rescaling factor $\Delta^{\ell'}$, output $\mathsf{ct}' \leftarrow \left\lceil \Delta^{-\ell'} \cdot \mathsf{ct} \right\rfloor \pmod{Q_{\ell-\ell'}}$.

  Typically rescaling operation is done after multiplication and by one level.

The CKKS scheme supports an efficient packing of n (up to $N/2$) real numbers into a single ciphertext. The encoding and decoding operations are defined as follows:

- $\mathsf{Encode}(\mathbf{x}, \Delta)$. For $\mathbf{x} \in \mathbb{C}^n$, output the polynomial $\boldsymbol{m} \leftarrow \left\lceil \tau_n'^{-1}(\Delta \cdot \mathbf{x}) \right\rfloor \in \mathcal{R}$.
- $\mathsf{Decode}(\boldsymbol{m}, \Delta)$. For a plaintext $\boldsymbol{m} \in \mathcal{R}$, output the polynomial $\mathbf{x} \leftarrow \tau_n'(\boldsymbol{m}/\Delta) \in \mathbb{C}^n$.

2.2 RNS Representation

Our implementation utilizes the Chinese Remainder Theorem (referred to as integer CRT) representation to break multi-precision integers in $\mathbb{Z}_q$ into vectors of smaller integers to perform operations efficiently using native (64-bit) integer types. The integer CRT representation is also often referred to as the Residue-Number-System (RNS) representation. We use a zero level modulus q_0 and a chain of same-size prime moduli $q_1, q_2, \ldots, q_L$ satisfying $q_i \equiv 1 \bmod 2N$ for $i = 1, \ldots, L$. Here, the modulus Q_ℓ is computed as $\prod_{i=0}^{\ell} q_i$. All polynomial multiplications are performed on ring elements in the polynomial CRT representation where all integer components are represented in the integer CRT basis.

[2] We can adapt swk to perform key switching for level $\ell < L$.

2.3 CKKS Scheme in RNS

RNS CKKS variants perform all operations in RNS. In other words, the power-of-two modulus $Q_\ell = 2^{p_0+\ell \cdot p}$ is replaced with $\prod_{i=0}^{\ell} q_i$, where q_i's are chosen as described above to support efficient number theoretic transforms (NTT) for converting native-integer polynomials w.r.t. each CRT modulus from coefficient representation to the evaluation one, and vice versa. The primes q_i for $i = 1, \ldots, \ell$ are chosen to be as close to 2^p as possible to minimize the error introduced by rescaling.

The two major changes in the RNS instantiation compared to the CKKS scheme deal with rescaling and key switching.

2.3.1 Rescaling in RNS

To efficiently perform rescaling in RNS from Q_ℓ to $Q_{\ell-1}$, the scaling down by 2^p is replaced with scaling down by q_ℓ. For $i \in [L]$, q_i are chosen, such that $2^p/q_i$ is in the range $(1-2^{-\epsilon}, 1+2^{-\epsilon})$, where ϵ is kept as small as possible. The new rescaling operation to scale down by one level is defined as

- $\mathsf{Rescale}(\mathsf{ct}, q_\ell)$. For a ciphertext $\mathsf{ct} \in \mathcal{R}_\ell^2$, output $\mathsf{ct}' \leftarrow \lceil q_\ell^{-1} \cdot \mathsf{ct} \rfloor \pmod{Q_{\ell-1}}$.

The maximum approximation error introduced by rescaling from ℓ to $\ell - 1$ is

$$\left| q_\ell^{-1} \cdot \boldsymbol{m} - 2^{-p} \cdot \boldsymbol{m} \right| \leq 2^{-\epsilon} \cdot \left| 2^{-p} \cdot \boldsymbol{m} \right| .$$

To minimize the cumulative approximation error growth in deeper computations, one can also alternate q_i w.r.t. 2^p. For instance, if $q_1 < 2^p$, then $q_2 > 2^p$ and $q_3 < 2^p$, etc. [5, 10]

2.3.2 Key Switching in RNS

To take advantage of RNS, we have to modify certain operations, such as base ω decomposition, to make them RNS-friendly. We use the hybrid key switching method described in [21]. Instead of the base ω decomposition, RNS digit decomposition is used. First, we use the partial products $\{\tilde{Q}_j\}_{0 \leq j < \mathsf{dnum}} = \{\prod_{i=j\alpha}^{(j+1)\alpha-1} q_i\}_{0 \leq j < \mathsf{dnum}}$, where $\alpha = (L+1)/\mathsf{dnum}$ for a pre-fixed parameter dnum. For level ℓ and $\mathsf{dnum}' = \lceil (\ell+1)/\alpha \rceil$ we then have:

$$\mathcal{WD}'_\ell(\boldsymbol{a}) = \left(\left[\boldsymbol{a} \frac{\tilde{Q}_0}{Q_\ell} \right]_{\tilde{Q}_0}, \ldots, \left[\boldsymbol{a} \frac{\tilde{Q}_{\mathsf{dnum}'-1}}{Q_\ell} \right]_{\tilde{Q}_{\mathsf{dnum}'-1}} \right) \in \mathcal{R}^{\mathsf{dnum}'},$$

$$\mathcal{PW}'_\ell(\boldsymbol{a}) = \left(\left[\boldsymbol{a} \frac{Q_\ell}{\tilde{Q}_0} \right]_{Q_\ell}, \ldots, \left[\boldsymbol{a} \frac{Q_\ell}{\tilde{Q}_{\mathsf{dnum}'-1}} \right]_{Q_\ell} \right) \in \mathcal{R}_{Q_\ell}^{\mathsf{dnum}'}.$$

For any $(\boldsymbol{a}, \boldsymbol{b}) \in \mathcal{R}_\ell^2$, $\mathcal{WD}'_\ell$ and $\mathcal{PW}'_\ell$ satisfy the following congruence relation:

$$\left\langle \mathcal{WD}'_\ell(\boldsymbol{a}), \mathcal{PW}'_\ell(\boldsymbol{b}) \right\rangle \equiv \boldsymbol{a} \cdot \boldsymbol{b} \pmod{Q_\ell}.$$

This key switching procedure is similar to the one used in CKKS with the only difference in the decomposition method.

- $\mathsf{KeySwitchGen}_{\mathsf{sk}}(\boldsymbol{s}')$. For auxiliary modulus $P = \prod_{i=0}^{k} p_i$, sample a random $\boldsymbol{a}'_k \leftarrow \mathcal{R}_{PQ_L}$ and error $\boldsymbol{e}'_k \leftarrow \chi_{\mathsf{err}}$. For a pre-fixed parameter dnum, output the switching key as

$$\mathsf{swk} = (\mathsf{swk}_0, \mathsf{swk}_1) = \left(\left\{\boldsymbol{b}'_k\right\}_{k=0}^{\mathsf{dnum}-1}, \left\{\boldsymbol{a}'_k\right\}_{k=0}^{\mathsf{dnum}-1}\right) \in \mathcal{R}_{PQ_L}^{2\times\mathsf{dnum}},$$

where

$$\boldsymbol{b}'_k \leftarrow -\boldsymbol{a}'_k \cdot \boldsymbol{s} + \boldsymbol{e}'_k + P \cdot \mathcal{PW}'\left(\boldsymbol{s}'\right)_k \pmod{PQ_L}.$$

- $\mathsf{KeySwitch}_{\mathsf{swk}}(\mathsf{ct})$. For $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}_{Q_\ell}^2$, $\mathsf{swk} = (\mathsf{swk}_0, \mathsf{swk}_1)$ [3] output

$$\left(\boldsymbol{c}_0 + \left\lceil \frac{\left\langle \mathcal{WD}'_\ell\left(\boldsymbol{c}_1\right), \mathsf{swk}_0\right\rangle}{P} \right\rfloor, \left\lceil \frac{\left\langle \mathcal{WD}'_\ell\left(\boldsymbol{c}_1\right), \mathsf{swk}_1\right\rangle}{P} \right\rfloor\right) \pmod{Q_\ell}.$$

To keep the noise from key switching small, we can take $P \approx \max_j(\tilde{Q}_j)$.

3 Reducing the Approximation Error in the CKKS Scheme

We first describe all approximation errors in the original CKKS scheme (for the case of uniform ternary secrets and hybrid key switching) and then we discuss how many of these errors can be removed. We choose the uniform ternary secret distribution (in contrast to sparse ternary secrets) because sparse ternary secrets are not currently supported by the HE standard [2], and uniform ternary secrets are the most efficient option that is supported by the HE standard. The hybrid key switching [21] is selected because it is more efficient than the GHS approach used in the original CKKS scheme and incurs a smaller approximation error than the digit decomposition approach [8] for relatively large digits, which are required for the efficient instantiation of the digit decomposition key switching method.

3.1 Approximation Errors in the CKKS Scheme

Encryption & Decryption. In the original CKKS [13] scheme, to encode the message $\mathbf{x} \in \mathbb{C}^n$, we apply the inverse embedding transformation $\mu = \tau_n'^{-1}(\mathbf{x}) \in \mathcal{S}$ and then scale μ by a factor $\Delta = 2^p$ and round to obtain the plaintext $\boldsymbol{m} := \lceil \Delta \cdot \mu \rfloor \in \mathcal{R}$. To encrypt $\boldsymbol{m}$ with the public key pk, we sample $\boldsymbol{v} \leftarrow \chi_{\mathsf{enc}}$ and $\boldsymbol{e}_0, \boldsymbol{e}_1 \leftarrow \chi_{\mathsf{err}}$, and output

$$\mathsf{ct} = \mathsf{Enc}(\boldsymbol{m}) = \mathsf{pk} \cdot \boldsymbol{v} + (\boldsymbol{e}_0 + \boldsymbol{m}, \boldsymbol{e}_1) \in \mathcal{R}_Q^2.$$

The full process is as follows

$$\mathbf{x} \xrightarrow{\tau_n'^{-1}(\cdot)} \mu \xrightarrow{\lceil \cdot \times \Delta \rfloor} \boldsymbol{m} \xrightarrow{\mathsf{Enc}_{\mathsf{pk}}(\cdot)} \mathsf{ct}.$$

To decrypt the ciphertext ct, we need to compute the inner product with sk modulo Q:

$$\tilde{\boldsymbol{m}} = \mathsf{Dec}_{\mathsf{sk}}\left(\mathsf{ct}\left(\boldsymbol{m}\right)\right) = \left[\langle \mathsf{ct}, \mathsf{sk} \rangle\right]_Q = \boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} \in \mathcal{R}_Q.$$

To decode $\tilde{\boldsymbol{m}}$, we divide it by Δ, i.e., $\tilde{\mu} = \tilde{\boldsymbol{m}}/\Delta$, and apply the embedding transformation $\tilde{\mathbf{x}} = \tau'_n(\tilde{\mu})$:

$$\mathsf{ct} \xrightarrow{\mathsf{Dec}_{\mathsf{sk}}(\cdot)} \tilde{\boldsymbol{m}} \xrightarrow{\cdot \div \Delta} \tilde{\mu} \xrightarrow{\tau'_n(\cdot)} \tilde{\mathbf{x}}.$$

[3] We can adapt swk to perform key switching for level $\ell < L$.

There are several sources of errors that contribute to the output error $\tilde{\mathbf{x}} - \mathbf{x}$. The $\tau_n^{'-1}$ and τ_n' maps are exact in theory, but in practice introduce precision (rounding) errors that depend on the floating-point precision and the value of n. We omit these errors for now, as we can always reduce them by increasing the floating-point precision. The same applies to multiplication $\times\Delta$ and division $\div\Delta$ in the encoding and decoding parts. However, in the encoding procedure, we do not only scale, but also round the scaled value, and the rounding introduces an approximation error $\boldsymbol{r}_{\mathsf{encode}}$ with $\|\boldsymbol{r}_{\mathsf{encode}}\|_\infty \le 1/2$. Public key encryption introduces a fresh encryption (LWE) error $\boldsymbol{e}_{\mathsf{fresh}}$. After encryption, the ciphertext ct satisfies the following relation:

$$\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} = \boldsymbol{m} + \boldsymbol{e}_{\mathsf{fresh}} = \Delta \cdot \mu + \boldsymbol{r}_{\mathsf{encode}} + \boldsymbol{e}_{\mathsf{fresh}} = \Delta \cdot \mu + \boldsymbol{f}_{\mathsf{enc}} \in \mathcal{R}_Q.$$

Instead of analyzing $\boldsymbol{f}$, $\boldsymbol{e}$, $\boldsymbol{r}$, it is more natural to analyze the scaled errors $\phi = \boldsymbol{f}/\Delta$, $\epsilon = \boldsymbol{e}/\Delta$, $\rho = \boldsymbol{r}/\Delta$ since the division by the scaling factor is part of the decoding procedure, and the scaled error is the one that is related to the error before applying the σ transformation in the decoding. In what follows, we will mainly refer to ϵ instead of $\boldsymbol{e}$.

One way to reduce the contribution of $\boldsymbol{f}_{\mathsf{enc}}$ is to increase the scaling factor Δ of the scheme. To keep the encryption secure under the RLWE problem, we need to increase the ring dimension in the underlying lattice problem, which may be inefficient in many cases.

We also provide a heuristic bound for fresh encryption noise/approximation error. It will be used for estimating the reduction of approximation error in our CKKS variant.

Lemma 3.1 *Given a uniform ternary secret key $\boldsymbol{s}$, we have the following heuristic bound for fresh encryption noise:*

$$\|\boldsymbol{f}_{\mathsf{enc}}\|^{\mathsf{can}} \le \frac{32}{3}\sqrt{6}\sigma N + 6\sigma\sqrt{N}.$$

Proof. See Appendix B. Note that for the sparse ternary secret setting with Hamming weight h, the bound would be formulated as $\|\boldsymbol{f}_{\mathsf{enc}}\|^{\mathsf{can}} \le 8\sqrt{2}\sigma N + 6\sigma\sqrt{N} + 16\sigma\sqrt{hN}$ [13].

Addition. The addition procedure $\mathsf{ct}_{\mathsf{add}} = \mathsf{Add}(\mathsf{ct}_1, \mathsf{ct}_2)$ for two ciphertexts at the same level ℓ is done as component-wise addition and leads to the following relation:

$$\boldsymbol{c}_{\mathsf{add},0} + \boldsymbol{c}_{\mathsf{add},1} \cdot \boldsymbol{s} = \Delta \cdot (\mu_1 + \mu_2) + (\boldsymbol{f}_1 + \boldsymbol{f}_2) \in \mathcal{R}_{Q_\ell}.$$

The addition does not introduce any additional errors, but instead adds the errors together, which is exactly what happens in the unencrypted case of adding two approximate numbers together.

Scalar Addition The scalar addition procedure $\mathsf{ct}_{\mathsf{cadd}} = \mathsf{CAdd}(\mathsf{ct}, \mathsf{const})$ leads to the following relation:

$$\boldsymbol{c}_{\mathsf{cadd},0} + \boldsymbol{c}_{\mathsf{cadd},1} \cdot \boldsymbol{s} = \Delta \cdot (\mu + \mu_{\mathsf{const}}) + (\boldsymbol{f} + \boldsymbol{r}_{\mathsf{encode}}),$$

where $\mathsf{Encode}(\mathsf{const}, \Delta) = \Delta\mu_{\mathsf{const}} + \boldsymbol{r}_{\mathsf{encode}}$. In addition to the encoding error, the scalar addition also introduces a floating-point precision error. Both errors in the scalar addition are relatively small compared to the ciphertext error.

Key Switching. There are several known key switching procedures

$$\mathsf{ct}_{\mathsf{ks}} = \mathsf{KeySwitch}_{\mathsf{swk}}(\mathsf{ct}),$$

which switch the ciphertext ct satisfying the relation:

$$\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s}_1 = \Delta \cdot \mu + \boldsymbol{f} \in \mathcal{R}_{Q_\ell},$$

to the ciphertext $\mathsf{ct}_{\mathsf{ks}}$ satisfying the relation:

$$\boldsymbol{c}_{\mathsf{ks},0} + \boldsymbol{c}_{\mathsf{ks},0} \cdot \boldsymbol{s}_2 = \Delta \cdot \mu + \boldsymbol{f} + \boldsymbol{e}_{\mathsf{ks}} \in \mathcal{R}_{Q_\ell}.$$

The key switching step introduces an LWE-related error $\boldsymbol{e}_{\mathsf{ks}}$.

Lemma 3.2 *For the key switching method described in Section 2.1, we have the following heuristic bound for key switching noise:*

$$\|\boldsymbol{e}_{ks}\|^{can} \leq \frac{8\sqrt{3} \cdot \mathit{dnum} \cdot \omega\sigma N}{3P} + \sqrt{3N} + \frac{8\sqrt{2}N}{3}.$$

Proof. See Appendix B. Note that for the sparse ternary secret setting with Hamming weight h, the bound would be formulated as $\|\boldsymbol{e}_{\mathsf{ks}}\|^{\mathsf{can}} \leq \frac{8\sqrt{3}\cdot\mathsf{dnum}\cdot\omega\sigma N}{3P} + \sqrt{3N} + 8\sqrt{\frac{hN}{3}}$.

Multiplication. The multiplication procedure $\mathsf{ct}_{\mathsf{mult}} = \mathsf{Mult}(\mathsf{ct}_1, \mathsf{ct}_2)$ for two ciphertexts at the same level ℓ is done in two steps: tensoring and key switching. The ciphertext after tensoring satisfies the following equation:

$$\boldsymbol{c}_{\mathsf{tensor},0} + \boldsymbol{c}_{\mathsf{tensor},1} \cdot \boldsymbol{s} + \boldsymbol{c}_{\mathsf{tensor},2} \cdot \boldsymbol{s}^2 \equiv (\Delta \cdot \mu_1 + \boldsymbol{f}_1) \cdot (\Delta \cdot \mu_2 + \boldsymbol{f}_2) = \Delta^2 \cdot \mu_1\mu_2 + \boldsymbol{f}_\times \in \mathcal{R}_{Q_\ell}.$$

In the tensoring step the error term $\boldsymbol{f}_\times$ is approximate multiplication error of $(\Delta \cdot \mu_i + \boldsymbol{f}_i)$ for the unencrypted case. Hence tensoring does not introduce new approximation errors.

The key switching part switches $\mathsf{ct}' = (0, \boldsymbol{c}_{\mathsf{tensor},2})$ as a ciphertext under the key $\boldsymbol{s}^2$ to the ciphertext $\mathsf{ct}'' = \mathsf{KeySwitch}_{\mathsf{evk}}(\mathsf{ct}')$ under the key $\boldsymbol{s}$, and the result is added to $(\boldsymbol{c}_{\mathsf{tensor},0}, \boldsymbol{c}_{\mathsf{tensor},1})$. The ciphertext after the key switching satisfies the following equation:

$$\boldsymbol{c}_{\mathsf{mult},0} + \boldsymbol{c}_{\mathsf{mult},1} \cdot \boldsymbol{s} \equiv \Delta^2 \cdot \mu_1\mu_2 + \boldsymbol{f}_\times + \boldsymbol{e}_{\mathsf{ks}} = \Delta^2 \cdot \mu_1\mu_2 + \boldsymbol{f}_{\mathsf{mult}} \in \mathcal{R}_{Q_\ell},$$

where

$$\boldsymbol{f}_{\mathsf{mult}} = \Delta \cdot (\mu_1 \boldsymbol{f}_2 + \mu_2 \boldsymbol{f}_1) + \boldsymbol{f}_1 \boldsymbol{f}_2 + \boldsymbol{e}_{\mathsf{ks}} = \boldsymbol{f}_\times + \boldsymbol{e}_{\mathsf{ks}},$$

and since the scaling factor becomes Δ^2 after multiplication, we have the following relation for the scaled error:

$$\phi_{\mathsf{mult}} = \frac{\boldsymbol{f}_{\mathsf{mult}}}{\Delta^2} = \mu_1\phi_2 + \mu_2\phi_1 + \phi_1\phi_2 + \frac{\epsilon_{\mathsf{ks}}}{\Delta} = \phi_\times + \frac{\epsilon_{\mathsf{ks}}}{\Delta}. \tag{1}$$

In Equation (1), we see that the scaled switching error ϵ_{ks} is divided by Δ. We can perform the key switching procedure in such a way that the term $\boldsymbol{e}_{\mathsf{ks}}$ is much smaller than Δ, which makes the impact of ϕ_{mult} essentially the same as the impact of $\phi_\times$ in an unencrypted case.

Scalar Multiplication The scalar multiplication procedure $\mathsf{ct}_{\mathsf{cmult}} = \mathsf{CMult}(\mathsf{ct}, \mathsf{const})$ is described using the following relation:

$$\boldsymbol{c}_{\mathsf{cmult},0} + \boldsymbol{c}_{\mathsf{cmult},1} \cdot \boldsymbol{s} = \Delta^2 \cdot (\mu\mu_{\mathsf{const}}) + \Delta \cdot (\mu_{\mathsf{const}}\boldsymbol{f} + \mu\boldsymbol{r}_{\mathsf{encode}}) + \boldsymbol{f}\boldsymbol{r}_{\mathsf{encode}} = \Delta^2 \cdot \mu\mu_{\mathsf{const}} + \boldsymbol{f}_{\mathsf{cmult}} \in \mathcal{R}_{Q_\ell},$$

where

$$\begin{aligned} \mathsf{Encode}(x, \Delta) &= \Delta \cdot \mu_{\mathsf{const}} + \boldsymbol{r}_{\mathsf{encode}}, \\ \boldsymbol{f}_{\mathsf{cmult}} &= \Delta \cdot (\mu_{\mathsf{const}}\boldsymbol{f} + \mu\boldsymbol{r}_{\mathsf{encode}}) + \boldsymbol{f}\boldsymbol{r}_{\mathsf{encode}} = \boldsymbol{f}_{\times\mathsf{c}}, \\ \phi_{\mathsf{cmult}} &= \mu\rho_{\mathsf{encode}} + \mu_{\mathsf{const}}\phi + \phi\rho_{\mathsf{encode}} = \phi_{\times\mathsf{c}}. \end{aligned}$$

Rescaling. In the CKKS scheme the main reason for rescaling is not to manage the noise, as in the case of the Brakerski-Gentry-Vaikuntantanathan (BGV) scheme [7], but to scale down the encrypted message and truncate some least significant bits. The size of the encrypted message increases after multiplication and decreases after rescaling. Other operations, like additions or rotations, do not affect the magnitude of the message. So we should balance multiplications and rescaling operations to control the magnitude of message and its precision. Normally it is advised to perform a rescaling right after each multiplication.

The rescaling procedure $\mathsf{ct}_{\mathsf{rs}} = \mathsf{Rescale}(\mathsf{ct}, \Delta)$ for a ciphertext at level ℓ is done by dividing by the scaling factor and rounding. The procedure is as follows:

$$\mathsf{ct}_{\mathsf{rs}} = \mathsf{Rescale}(\mathsf{ct}, \Delta) = \left(\left\lceil \frac{\boldsymbol{c}_0}{\Delta} \right\rfloor, \left\lceil \frac{\boldsymbol{c}_1}{\Delta} \right\rfloor \right) = \left(\frac{\boldsymbol{c}_0}{\Delta} + \boldsymbol{r}_0, \frac{\boldsymbol{c}_1}{\Delta} + \boldsymbol{r}_1 \right),$$

where $\boldsymbol{r}_0$ and $\boldsymbol{r}_1$ are error terms introduced by rounding, with coefficients in $[-1/2, 1/2]$.

The ciphertext after the multiplication and rescaling procedure $\mathsf{ct}_{\mathsf{mult+rs}} = \mathsf{Rescale}(\mathsf{Mult}(\mathsf{ct}_1, \mathsf{ct}_2), \Delta)$ satisfies the following relation:

$$\begin{aligned} \boldsymbol{c}_{\mathsf{mult+rs},0} + \boldsymbol{c}_{\mathsf{mult+rs},1} \cdot \boldsymbol{s} &\equiv \frac{(\Delta \cdot \mu_1 + \boldsymbol{f}_1) \cdot (\Delta \cdot \mu_2 + \boldsymbol{f}_2) + \boldsymbol{e}_{\mathsf{ks}}}{\Delta} + \boldsymbol{r}_0 + \boldsymbol{r}_1\boldsymbol{s} \\ &= \Delta \cdot \mu_1\mu_2 + \boldsymbol{f}_{\mathsf{mult+rs}} \in \mathcal{R}_{Q_{\ell-1}}, \end{aligned}$$

where

$$\begin{aligned} \boldsymbol{f}_{\mathsf{mult+rs}} &= \frac{\boldsymbol{f}_\times}{\Delta} + \frac{\boldsymbol{e}_{\mathsf{ks}}}{\Delta} + \boldsymbol{r}_0 + \boldsymbol{r}_1\boldsymbol{s} = \frac{\boldsymbol{f}_\times}{\Delta} + \frac{\boldsymbol{e}_{\mathsf{ks}}}{\Delta} + \boldsymbol{r}_{\mathsf{rs}}, \\ \phi_{\mathsf{mult+rs}} &= \phi_\times + \frac{\epsilon_{\mathsf{ks}}}{\Delta} + \rho_{\mathsf{rs}}, \end{aligned}$$

where $\boldsymbol{r}_{\mathsf{rs}} = \boldsymbol{r}_0 + \boldsymbol{r}_1\boldsymbol{s}$ is the rounding error, and $\rho_{\mathsf{rs}} = \boldsymbol{r}_{\mathsf{rs}}/\Delta$ is the scaled rounding error. Thus after the rescaling procedure, the scaled approximation error $\epsilon_{\mathsf{ks}}/\Delta$ is negligible and gets completely absorbed by the rounding error ρ_{rs}.

Lemma 3.3 *Given a uniform ternary secret key $\boldsymbol{s}$, we have the following heuristic bound for the rounding error that is introduced by rescaling*

$$\|\boldsymbol{r}_{rs}\|^{can} \leq \sqrt{3N} + \frac{16\sqrt{2}N}{3}.$$

Proof. See Appendix B. Note that for the sparse ternary secret setting with Hamming weight h, the bound would be formulated as $\|\boldsymbol{r}_{\mathsf{rs}}\|^{\mathsf{can}} \leq \sqrt{3N} + 8\sqrt{\frac{hN}{3}}$ [13].

Modulus Reduction. The CKKS scheme also has a modulus reduction procedure that does not change the message or approximation error. This modulus reduction procedure is done simply by evaluating the ciphertext ct at modulus Q_ℓ modulo smaller modulus $Q_{\ell'}$. As $Q_{\ell'}|Q_\ell$, the method does not introduce any additional errors.

Automorphism (Rotation & Conjugation). Similar to the multiplication procedure, the automorphism procedure $\mathsf{ct}_{\mathsf{aut}} = \mathsf{Aut}_{\mathsf{rk}^{(\kappa)}}(\mathsf{ct}, \kappa)$ is done in two steps: automorphism κ and key switching. The ciphertext after automorphism satisfies the following relation:

$$\boldsymbol{c}_0^{(\kappa)} + \boldsymbol{c}_1^{(\kappa)} \cdot \boldsymbol{s}^{(\kappa)} \equiv \Delta \cdot \mu^{(\kappa)} + \boldsymbol{f}^{(\kappa)} \in \mathcal{R}_{Q_\ell}.$$

The key switching part switches $\mathsf{ct}' = (0, \boldsymbol{c}_1^{(\kappa)})$ as a ciphertext under the key $\boldsymbol{s}^{(\kappa)}$ to the ciphertext $\mathsf{ct}'' = \mathsf{KeySwitch}_{\mathsf{rk}^{(\kappa)}}(\mathsf{ct}')$ under the key $\boldsymbol{s}$, and the result is added to $(\boldsymbol{c}_0^{(\kappa)}, 0)$. The ciphertext after the key switching satisfies the following equation:

$$\boldsymbol{c}_{\mathsf{aut},0} + \boldsymbol{c}_{\mathsf{aut},1} \cdot \boldsymbol{s} \equiv \Delta \cdot (\mu^{(\kappa)}) + \boldsymbol{f}^{(\kappa)} + \boldsymbol{e}_{\mathsf{ks}} = \Delta \cdot \mu^{(\kappa)} + \boldsymbol{f}_{\mathsf{aut}} \in \mathcal{R}_{Q_\ell},$$

where

$$\boldsymbol{f}_{\mathsf{aut}} = \boldsymbol{f}^{(\kappa)} + \boldsymbol{e}_{\mathsf{ks}} \text{ and } \phi_{\mathsf{aut}} = \frac{\boldsymbol{f}_{\mathsf{aut}}}{\Delta} = \phi^{(\kappa)} + \epsilon_{\mathsf{ks}}.$$

In case of automorphism operations, the key switching error ϵ_{ks} is not negligible anymore compared to $\phi^{(\kappa)}$, as the scaling factor in the case of automorphism is not squared but stays the same.

3.2 Eliminating LWE and Encoding Approximation Errors

One can see that the rescaling operation does not necessarily need to be done right after the multiplication, and instead can be done right before the next multiplication (or before decryption). In other words, we do not rescale after the multiplication and keep the scaling factor as Δ^2. For the first level, we can encrypt the message μ with the scaling factor Δ^2 to make the encryption noise negligible. The ciphertext ct will satisfy the following relation:

$$\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} \equiv \left\lceil \Delta^2 \cdot \mu \right\rfloor + \boldsymbol{e}_{\mathsf{fresh}} = \Delta^2 \cdot \mu + \boldsymbol{f}' \in \mathcal{R}_{Q_\ell}.$$

All other operations, like additions and automorphisms, are done the same way. The approximation errors will be summed together and in practice will be much smaller than the scaling factor Δ^2. The rescaling operation is done right before the next multiplication so that the scaled LWE and encoding errors are dominated by the rounding error after the rescaling. So we can make all LWE and encoding errors negligible compared to the rounding rescaling errors, starting with the second level.

As the rescaling operation is performed right before the multiplication, we can treat it as part of the multiplication. We can redefine the multiplication Mult' as a combination of rescaling operations and multiplication:

$$\mathsf{ct}_{\mathsf{mult}'} = \mathsf{Mult}'(\mathsf{ct}_1, \mathsf{ct}_2) = \mathsf{Mult}\left(\mathsf{Rescale}(\mathsf{ct}_1, \Delta), \mathsf{Rescale}(\mathsf{ct}_2, \Delta)\right).$$

With this new definition of Mult', we keep the same number of levels while slightly increasing the modulus for the fresh ciphertext from $q_0 \cdot \Delta^L$ to $q_0 \cdot \Delta^{L+1}$. We also ensure that fresh encryption noise and key switching noise, which appear after multiplication or automorphism operations, will be negligible and absorbed by the rescaling rounding error. In other words, we can eliminate all LWE and encoding approximation errors, by making them negligible compared to rescaling rounding errors.

We also reduce the total rounding error when we add ciphertexts. If we perform the rescaling right after multiplication, the rounding error is introduced for each ciphertext and the rescaling errors will be added when we perform addition of the ciphertexts. In the case of the new multiplication Mult', we do rescaling after the additions, and hence we end up only with a single rounding error.

With the modified multiplication, the encryption of a message μ at level ℓ will satisfy the following condition:

$$\boldsymbol{c}_0 + \boldsymbol{c}_1 \boldsymbol{s} \equiv \Delta^2 \cdot \mu + \boldsymbol{f}'.$$

Let $\boldsymbol{f}'/\Delta^2 = \phi'$. After Mult' operation we have:

$$\begin{aligned}\boldsymbol{c}_{\mathsf{mult}',0} + \boldsymbol{c}_{\mathsf{mult}',1}\boldsymbol{s} &\equiv \left(\frac{\Delta^2 \cdot \mu_1 + \boldsymbol{f}_1'}{\Delta} + \boldsymbol{r}_{\mathsf{rs},1}\right) \cdot \left(\frac{\Delta^2 \cdot \mu_2 + \boldsymbol{f}_2'}{\Delta} + \boldsymbol{r}_{\mathsf{rs},2}\right) + \boldsymbol{e}_{\mathsf{ks}} \\ &= \left(\Delta \cdot \left(\mu_1 + \phi_1'\right) + \boldsymbol{r}_{\mathsf{rs},1}\right) \cdot \left(\Delta \cdot \left(\mu_2 + \phi_2'\right) + \boldsymbol{r}_{\mathsf{rs},2}\right) + \boldsymbol{e}_{\mathsf{ks}} \\ &= \Delta^2 \cdot \mu_1\mu_2 + \boldsymbol{f}_{\mathsf{mult}'},\end{aligned}$$

where

$$\begin{aligned}\boldsymbol{f}_{\mathsf{mult}'} &= \Delta^2 \cdot \left(\mu_1\phi_2' + \mu_2\phi_1' + \phi_1'\phi_2'\right) + \\ &\quad + \Delta \cdot \left(\left(\mu_1 + \phi_1'\right)\boldsymbol{r}_{\mathsf{rs},2} + \left(\mu_2 + \phi_2'\right)\boldsymbol{r}_{\mathsf{rs},1}\right) + \boldsymbol{r}_{\mathsf{rs},1}\boldsymbol{r}_{\mathsf{rs},2} + \boldsymbol{e}_{\mathsf{ks}}, \\ \phi_{\mathsf{mult}'} &= \mu_1\left(\phi_2' + \rho_{\mathsf{rs},2}\right) + \mu_2\left(\phi_1' + \rho_{\mathsf{rs},1}\right) + \left(\phi_1' + \rho_{\mathsf{rs},1}\right)\left(\phi_2' + \rho_{\mathsf{rs},2}\right) + \frac{\epsilon_{\mathsf{ks}}}{\Delta}.\end{aligned}$$

Remark We can also substitute Δ^2 in fresh encryption with a tighter scaling factor $\Delta \cdot \Delta'$, where $\Delta' = 2^{p'} < 2^p = \Delta$. We need to choose Δ' in such a way that the sum of all LWE errors during the computations on the level L, including fresh encryption noise, is smaller than Δ'. In this case, in Mult' on the first level we need to do rescaling by Δ' instead of Δ. The modulus Q_L for the fresh ciphertext will be increased by a smaller factor Δ' and become $Q_L = q_0 \cdot \Delta^L \cdot \Delta'$. We use this tighter scaling factor Δ' in our implementation.

3.3 Theoretical Estimates of Error Reduction

Computation without multiplications. If only additions and automorphism operations are performed, no rescaling errors introduced and the LWE noise is the main source of approximation error. With standard parameters $\sigma = 3.2$, $P = \omega = Q_L^{1/3}$, from Lemma 3.1 the fresh encryption error is bounded by $\approx 83.6N$, and from Lemma 3.2 the key switching error is bounded by $\approx 44.3N$. The total number of error bits is $\log(83.6\alpha N + 44.3\beta N)$, where α is the number of fresh ciphertexts used, and β is the number of automorphism operations performed. The extra modulus Δ' in Reduced-Error (RE) CKKS is taken to fully absorb the error: $\Delta' > 83.6\alpha N + 44.3\beta N$. The total error before decryption is bounded by $\boldsymbol{r}_{\mathsf{float}}$, which is in practice only 2-5 bits less than the precision of floating-point arithmetic. This is illustrated by the experimental results presented in Tables 4 and 5 for $\Delta \approx 2^{50}$.

Computation with multiplications. The extra modulus Δ' used during encryption in RE-CKKS effectively reduces the encryption noise from fresh $\boldsymbol{e}_{\mathsf{fresh}}$ to rescaling $\boldsymbol{r}_{\mathsf{rs}}$ at the first multiplication step. From Lemmas 3.1 and 3.3, we have the following ratio of the upper bounds for fresh encryption and rescaling rounding errors (for the case of uniform ternary secrets):

$$\log\left(\frac{\boldsymbol{e}_{\mathsf{fresh}}}{\boldsymbol{r}_{\mathsf{rs}}}\right) \approx \log\left(\frac{\frac{32}{3}\sqrt{6}\sigma N + 6\sigma\sqrt{N}}{\sqrt{3N} + \frac{8\sqrt{2}N}{3}}\right) \approx \log\left(4\sqrt{3}\sigma\right) \approx 4.5.$$

At the next multiplication, the input error for RE-CKKS can be estimated as

$$\boldsymbol{f}'_{\mathsf{mult+rs}} \approx (\mu_1 \boldsymbol{r}_{\mathsf{rs}} + \mu_2 \boldsymbol{r}_{\mathsf{rs}}) + \boldsymbol{r}_{\mathsf{rs}}$$

as compared to

$$\boldsymbol{f}_{\mathsf{mult+rs}} \approx (\mu_1 \boldsymbol{e}_{\mathsf{fresh}} + \mu_2 \boldsymbol{e}_{\mathsf{fresh}}) + \boldsymbol{r}_{\mathsf{rs}}$$

for the original CKKS scheme. As $\boldsymbol{r}_{\mathsf{rs}} \ll \boldsymbol{e}_{\mathsf{fresh}}$, the rescaling rounding error typically has no effect on multiplications in the original CKKS, while in the case of RE-CKKS, $\boldsymbol{r}_{\mathsf{rs}}$ still gives a significant contribution. In practice, this implies there may be a small decline in the precision gain of RE-CKKS over CKKS for subsequent multiplications (typically not more than 0.5 bits), but this decline will become progressively smaller for further multiplications as the rounding errors from prior multiplications accumulate, and the current error will become much larger than the rounding error $\boldsymbol{r}_{\mathsf{rs}}$.

Hence in theory the upper bound of RE-CKKS error is about 4.5 bits smaller than the upper bound of CKKS error after the first multiplication, and it may slighly decline for further multiplications. This is consistent with the implementation results presented in Section 5, where the RE-CKKS error is about 4 bits smaller than the CKKS error across different circuits with multiplications, and we also observe a decline of precision gain from 4 (for first multiplication) to 3.5 bits (for deeper multiplications) for a binary tree multiplication benchmark (Table 6).

Note that in the sparse ternary secret key setting with Hamming weight $h = 64$, the precision gain of RE-CKKS over CKKS is higher:

$$\log\left(\frac{\boldsymbol{e}_{\mathsf{fresh}}}{\boldsymbol{r}_{\mathsf{rs}}}\right) \approx \log\left(\frac{8\sqrt{2}\sigma N + 6\sigma\sqrt{N} + 16\sigma\sqrt{hN}}{\sqrt{3N} + 8\sqrt{\frac{hN}{3}}}\right) \approx \log\left(\sqrt{6}\sigma\sqrt{\frac{N}{h}}\right) \approx \frac{1}{2}\log N.$$

For example, for $N = 2^{14}$ the gain of RE-CKKS over CKKS is about 7 bits. But since the sparse setting is not currently supported by the HE standard [2], we implement and examine the uniform ternary secret setting instead.

4 Reducing the Approximation Error in the RNS Instantiation of CKKS

In this section, we describe the procedures needed for eliminating the scaling factor approximation error in RNS and apply the RE-CKKS improvements presented in Section 3 to the RNS setting.

4.1 Eliminating the Scaling Factor Approximation Error in RNS CKKS

For the RNS setting, the noise control is more challenging as instead of a suitable ciphertext modulus $Q = 2^{p_0+p\cdot L} = q_0 \cdot \Delta^L$, we should use a ciphertext modulus $Q = \prod_{i=0}^{L} q_i$ - product of primes q_i. The rescaling operation is done by dividing by q_i, which are no longer powers of two.

The works [5, 10] that independently developed RNS variants of CKKS suggested to keep the scaling factor Δ constant, and pick the RNS moduli q_i close to Δ.

Let q_i be such that $\Delta/q_i = 1 + \alpha_i$, where $|\alpha_i|$ is kept as small as possible. Consider again the multiplication procedure with rescaling at some level ℓ:

$$\begin{aligned}\boldsymbol{c}_{\mathsf{mult+rs},0} + \boldsymbol{c}_{\mathsf{mult+rs},1}\boldsymbol{s} &\equiv \frac{(\Delta \cdot \mu_1 + \boldsymbol{f}_1)\cdot(\Delta\cdot\mu_2 + \boldsymbol{f}_2) + \boldsymbol{e}_{\mathsf{ks}}}{q_\ell} + \boldsymbol{r}_{\mathsf{rs}} \\ &= \Delta\cdot\mu_1\mu_2 + \boldsymbol{u}_\Delta + \frac{\boldsymbol{f}_\times}{q_\ell} + \frac{\boldsymbol{e}_{\mathsf{ks}}}{q_\ell} + \boldsymbol{r}_{\mathsf{rs}} = \Delta\cdot\mu_1\mu_2 + \boldsymbol{f}_{\mathsf{mult+rs}},\end{aligned}$$

where

$$\boldsymbol{u}_\Delta = \alpha_\ell\cdot\Delta\cdot\mu_1\mu_2, \boldsymbol{f}_{\mathsf{mult+rs}} = \boldsymbol{u}_\Delta + \frac{\boldsymbol{f}_\times}{q_\ell} + \frac{\boldsymbol{e}_{\mathsf{ks}}}{q_\ell} + \boldsymbol{r}_{\mathsf{rs}}.$$

The scaling factor error term $\boldsymbol{u}_\Delta$ appears here due to the difference between the scaling factor Δ and prime q_ℓ, and typically is the largest among the summands in the RNS instantiation of CKKS. We can see that $\boldsymbol{u}_\Delta$ depends on the distribution of specially chosen prime numbers, and is hence hard to control. We can consider optimizing the prime moduli selection to minimize the scaling factor error at each level. But if we consider operations over ciphertexts at different levels, we would have to deal with different scaling factor errors and the optimal configuration of prime moduli would be different. This implies that we would have to analyze the noise growth and find an optimal configuration of prime moduli for each specific computation circuit separately. A more detailed discussion of this issue is provided in Appendix A.

4.1.1 Using a Different Scaling Factor for Each Level

There is a way to eliminate the scaling factor error completely. As moduli q_i are public, we can integrate $\boldsymbol{u}_\Delta$ into the scaling factor and adjust the scaling factor after each rescaling. Let the ciphertext ct encrypt μ at some level ℓ with the scaling factor Δ_ℓ. The ciphertext ct satisfies the following relation:

$$\boldsymbol{c}_0 + \boldsymbol{c}_1\cdot\boldsymbol{s} \equiv \lceil \Delta_\ell\cdot\mu \rfloor + \boldsymbol{e}_{\mathsf{fresh}} = \Delta_\ell\cdot\mu + \boldsymbol{f}_{\mathsf{enc}} \pmod{Q_\ell}.$$

With different scaling factors at different levels, we no longer have the approximate scaling error. However, as the evaluation circuits are often quite complex, we now face different problems. Depending on the order of rescaling operations when evaluating the circuit, we can have different scaling factors for ciphertexts at the same level or different final scaling factors.

A naive solution to resolve these problems is to adjust the scaling factors at the same level by multiplying by corresponding constants. This seems to be highly inefficient and could double the number of levels in the worst case, as we would need to introduce an extra scalar multiplication for many normal operations.

Instead, we enforce the rescaling to be done automatically right after each multiplication of ciphertexts. With this automated rescaling, we ensure that all ciphertexts at the same level have the same scaling factors. The ciphertext after the multiplication procedure with rescaling

$$\mathsf{ct}_{\mathsf{mult+rs}} = \mathsf{Rescale}\left(\mathsf{Mult}\left(\mathsf{ct}_1, \mathsf{ct}_2\right), q_\ell\right),$$

will satisfy the following relation:

$$\begin{aligned}\boldsymbol{c}_{\mathsf{mult+rs},0} + \boldsymbol{c}_{\mathsf{mult+rs},1}\boldsymbol{s} &\equiv \frac{(\Delta_\ell \cdot \mu_1 + \boldsymbol{f}_1)\cdot(\Delta_\ell \cdot \mu_2 + \boldsymbol{f}_2) + \boldsymbol{e}_{\mathsf{ks}}}{q_\ell} + \boldsymbol{r}_{\mathsf{rs}} \\ &= \Delta_{\ell-1}\cdot \mu_1 \cdot \mu_2 + \boldsymbol{f}_{\mathsf{mult+rs}} \pmod{Q_{\ell-1}},\end{aligned}$$

where

$$\boldsymbol{f}_{\mathsf{mult+rs}} = \frac{\boldsymbol{f}_\times}{q_\ell} + \frac{\boldsymbol{e}_{\mathsf{ks}}}{q_\ell} + \boldsymbol{r}_{\mathsf{rs}},$$

and we have the relation between moduli and scaling factors:

$$\Delta_{\ell-1} := \frac{\Delta_\ell^2}{q_\ell}.$$

The following table shows how the scaling factors change during the computations depending on the level of the ciphertext:

Level	fresh Δ_ℓ OR after $\mathsf{Mult} + \mathsf{Rescale}$
L	$\Delta_L = q_L$
$L-1$	$\Delta_{L-1} = \Delta_L^2/q_L = q_L$
$L-2$	$\Delta_{L-2} = \Delta_{L-1}^2/q_{L-1} = q_L^2/q_{L-1}$
$\cdots$	$\cdots$
ℓ	$\Delta_\ell = \Delta_{\ell+1}^2/q_{\ell+1}$
$\cdots$	$\cdots$
0	$\Delta_0 = \Delta_1^2/q_1$

The choice of the initial scaling factor $\Delta_L = q_L$ will become clear from Section 4.1.3.

4.1.2 Handling the Operations between Ciphertexts at Different Levels

With the approach of automated rescaling, we always get the same scaling factors for the same level. However, we still have to deal with ciphertexts at different levels, i.e., with different scaling factors. Let us say we have two ciphertexts ct_1, ct_2 with levels $\ell_1 > \ell_2$ and scaling factors Δ_{ℓ_1} and Δ_{ℓ_2}. We have to adjust them to be at level ℓ_2 and to have the scaling factor Δ_{ℓ_2}.

- $\mathsf{Adjust}\left(\mathsf{ct}_1, \ell_2\right)$. For a ciphertext ct_1 with level ℓ_1 and scaling factor Δ_{ℓ_1}, drop moduli $\{q_{\ell_2+2}, \ldots, q_{\ell_1}\}$, multiply the result by a constant $\left\lceil \frac{\Delta_{\ell_2}\cdot q_{\ell_2+1}}{\Delta_{\ell_1}} \right\rfloor = \frac{\Delta_{\ell_2}\cdot q_{\ell_2+1}}{\Delta_{\ell_1}} + \delta$, with $\delta \in [-1/2, 1/2]$ and finally rescale by q_{ℓ_2+1}.

Let a ciphertext $\mathsf{ct}_1 = (\boldsymbol{c}_0, \boldsymbol{c}_1)$ satisfy the following relation:

$$\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} = \Delta_{\ell_1} \cdot \mu + \boldsymbol{f} \pmod{Q_{\ell_1}}.$$

The adjustment procedure $\mathsf{ct}_{\mathsf{adjust}} = \mathsf{Adjust}\left(\mathsf{ct}_1, \ell_2\right)$ for a ciphertext ct_1 leads to the following relation:

$$\boldsymbol{c}_{\mathsf{adjust},0} + \boldsymbol{c}_{\mathsf{adjust},1} \cdot \boldsymbol{s} = \frac{1}{q_{\ell_2+1}}\left(\Delta_{\ell_1}\cdot \mu + \boldsymbol{f}\right)\cdot\left(\frac{\Delta_{\ell_2}\cdot q_{\ell_2+1}}{\Delta_{\ell_1}} + \delta\right) + \boldsymbol{r}_{\mathsf{rs}} = \Delta_{\ell_2}\cdot\mu + \boldsymbol{f}_{\mathsf{adjust}} \pmod{Q_{\ell_2}},$$

with

$$\boldsymbol{f}_{\mathsf{adjust}} = \frac{\Delta_{\ell_2}}{\Delta_{\ell_1}} \cdot \boldsymbol{f} + \frac{\delta\Delta_{\ell_1} \cdot \mu + \delta\boldsymbol{f}}{q_{\ell_2+1}} + \boldsymbol{r}_{\mathsf{rs}},$$

where the second error term is introduced by scalar multiplication and the error $\boldsymbol{r}_{\mathsf{rs}}$ is introduced by the rescaling. Consider scaled errors $\boldsymbol{f}/\Delta_\ell = \phi^{(\ell)}$, $\boldsymbol{r}/\Delta_\ell = \rho^{(\ell)}$, $\boldsymbol{e}/\Delta_\ell = \epsilon^{(\ell)}$, then we have

$$\phi^{(\ell_2)}_{\mathsf{adjust}} = \phi^{(\ell_1)} + \frac{\delta\Delta_{\ell_1} \cdot \mu}{\Delta^2_{\ell_2+1}} + \frac{\delta\phi^{(\ell_2+1)}}{\Delta_{\ell_2+1}} + \rho^{(\ell_2)}_{\mathsf{rs}}. \tag{2}$$

We now can redefine addition and multiplication operations for ciphertexts at different levels.

- $\mathsf{CrossLevelAdd}(\mathsf{ct}_1, \mathsf{ct}_2)$ If $\ell_1 = \ell_2$, output $\mathsf{Add}(\mathsf{ct}_1, \mathsf{ct}_2)$, else w.l.o.g. $\ell_1 > \ell_2$. We first adjust ct_1 to level ℓ_2, $\mathsf{ct}'_1 = \mathsf{Adjust}(\mathsf{ct}_1, \ell_2)$, and then output $\mathsf{Add}(\mathsf{ct}'_1, \mathsf{ct}_2)$.

- $\mathsf{CrossLevelMult}(\mathsf{ct}_1, \mathsf{ct}_2)$ If $\ell_1 = \ell_2$, output $\mathsf{Mult}(\mathsf{ct}_1, \mathsf{ct}_2)$, else w.l.o.g. $\ell_1 > \ell_2$. We first adjust ct_1 to level ℓ_2, $\mathsf{ct}'_1 = \mathsf{Adjust}(\mathsf{ct}_1, \ell_2)$, and then output $\mathsf{Mult}(\mathsf{ct}'_1, \mathsf{ct}_2)$.

In Equation (2), we want Δ_{ℓ_1} and Δ_{ℓ_2+1} to be close to each other to keep the error $\phi^{(\ell_2)}_{\mathsf{adjust}}$ small.

Algorithm 1 Selection of RNS prime moduli in RNS-HEAAN [10] and PALISADE [5]; FirstPrime finds the first prime modulus $q_L > 2^p$ such that $q_L = 1 \pmod{2N}$. PreviousPrime and NextPrime decrement/increment with step $2N$ until a prime modulus is found.

```
procedure SelectModuli($N, L, p, p_0$)
    $q_L :=$ FirstPrime($p, 2N$)
    $q_{\mathsf{next}} := q_L$
    $q_{\mathsf{prev}} := q_L$
    flip $:= 0$
    for $\ell = L-1, \ldots, 1$ do
        if flip $\pmod 2 = 0$ then
            $q_\ell :=$ PreviousPrime($q_{\mathsf{prev}}, 2N$)
            $q_{prev} := q_\ell$
        else
            $q_\ell :=$ NextPrime($q_{\mathsf{next}}, 2N$)
            $q_{next} := q_\ell$
        flip $:=$ flip $+ 1$
    $q_0 :=$ PreviousPrime($p_0, 2N$)
```

4.1.3 Choosing the Primes to Avoid the Divergence of Scaling Factors

We initially tried to reuse the alternating logic for selecting the prime moduli in the CKKS RNS instantiations [5, 10], which was introduced to minimize the approximate scaling error. The algorithm showing this logic is listed in Algorithm 1. However, the scaling factors chosen using this logic diverge after ≈ 20 or ≈ 30 levels (for double-precision floats used in our implementation), as illustrated in Figure 1. As soon as the scaling factor significantly deviates from 2^p, the scaling factor quickly diverges from 2^p either towards 0 or infinity due to the exponential nature of scaling factor computation (the scaling factor is squared at each level). As this situation is not acceptable, we had to devise alternative algorithms.

To address this problem, we developed two other algorithms (Algorithms 2 and 3) where instead of minimizing the difference between Δ_ℓ and 2^p, we minimize the difference between two subsequent scaling factors. Algorithm 2 directly applies this logic. Algorithm 3 refines this logic by also alternating the selection of moduli w.r.t to the previous scaling factor (first a larger prime modulus is selected, then a smaller modulus, etc.), i.e., it combines Algorithms 1 and 2 to further minimize the error introduced by the deviation of the current scaling factor. Figure 1 shows that the deviation of the scaling factors from 2^p is very small for both Algorithms 2 and 3 up to 50 levels. Eventually both algorithms diverge, but it happened after 200 levels for all ring dimensions N we ran experiments for. As Algorithm 3 has smoother behaviour, we chose it for our implementation.

Algorithm 2 Selecting the prime moduli using the closest-prime-to-scaling-factor logic; FirstPrime finds the first prime modulus $q_L > 2^p$ such that $q_L = 1 \pmod{2N}$. PreviousPrime and NextPrime decrement/increment with step $2N$ until a prime modulus is found. ClosestPrime chooses the nearest between PreviousPrime and NextPrime.

```
procedure SelectModuli($N, L, p, p_0$)
    $q_L :=$ FirstPrime($p, 2N$)
    $\Delta_L := q_L$
    $\Delta_{L-1} := q_L$
    for $\ell = L-2, \ldots, 1$ do
        $\Delta_\ell := \frac{(\Delta_{\ell+1})^2}{q_{\ell+1}}$
        $q_\ell :=$ ClosestPrime($\lceil \Delta_\ell \rfloor - [\lceil \Delta_\ell \rfloor]_{2N} + 1, 2N$)
    $q_0 :=$ PreviousPrime($p_0, 2N$)
```

Algorithm 3 Selecting the prime moduli using a hybrid of Algorithms 1 and 2; FirstPrime finds the first prime modulus $q_L > 2^p$ such that $q_L = 1 \pmod{2N}$. PreviousPrime and NextPrime decrement/increment with step $2N$ until a prime modulus is found.

```
procedure SelectModuli($N, L, p, p_0$)
    $q_L :=$ FirstPrime($p, 2N$)
    $\Delta_L := q_L$
    $\Delta_{L-1} := q_L$
    flip $:= 0$
    for $\ell = L-2, \ldots, 1$ do
        $\Delta_\ell := \frac{(\Delta_{\ell+1})^2}{q_{\ell+1}}$
        if flip $\pmod 2 = 0$ then
            $q_\ell :=$ PreviousPrime($\lceil \Delta_\ell \rfloor - [\lceil \Delta_\ell \rfloor]_{2N} + 1, 2N$)
        else
            $q_\ell :=$ NextPrime($\lceil \Delta_\ell \rfloor - [\lceil \Delta_\ell \rfloor]_{2N} + 1, 2N$)
        flip $:=$ flip $+ 1$
    $q_0 :=$ PreviousPrime($p_0, 2N$)
```

Note that we chose $\Delta_L = q_L$ to reuse this scaling factor at level $L-1$, hence getting one level for "free" (without squaring and division).

In our implementation we also added a condition to check that the scaling factor is within a factor of 2 of 2^p. If this condition is not met, PALISADE throws an exception.

4.2 Applying the Reduced-Error CKKS Modifications

With different scaling factors at different levels, we no longer have the approximate scaling error. Hence now we can apply the RE-CKKS techniques to further reduce the approximation error. For the original CKKS scheme we considered the idea of modified multiplication where rescaling is done right before the next multiplication. The same idea can be adapted to the RNS instantiation of CKKS to reduce the LWE related noise:

$$\mathsf{ct}_{\mathsf{mult}'} = \mathsf{Mult}'(\mathsf{ct}_1, \mathsf{ct}_2) = \mathsf{Mult}\left(\mathsf{Rescale}(\mathsf{ct}_1, q_\ell), \mathsf{Rescale}(\mathsf{ct}_2, q_\ell)\right).$$

With the modified multiplication, we also ensure that the ciphertexts at the same level have the same scaling factor, as we do not shuffle Rescale and Mult operations, but just delay the Rescale operation to be done right before next Mult, instead of right after the multiplication. This delay of the Rescale operation has the same effect as eliminating LWE errors in RE-CKKS.

For level L, we add an extra modulus q' satisfying $q' = 1 \pmod{2N}$, such that the sum of all LWE errors during the computations at level L, including fresh encryption noise, is smaller than q'. The following table shows how the scaling factors change during a computation depending on the level of the ciphertext:

Level	fresh Δ_ℓ OR after Mult'
L	$\Delta_L \cdot \Delta' = q_L \cdot q'$
$L-1$	$\Delta_L^2 = q_L^2$
$L-2$	$\Delta_{L-1}^2 = q_L^2$
$\cdots$	$\cdots$
$\ell+1$	Δ_ℓ^2
$\cdots$	$\cdots$
0	Δ_1^2

With the modified multiplication, the encryption of a message μ at level ℓ will satisfy the following condition (for an encryption with an extra level we need to substitute Δ_ℓ^2 with $\Delta_L \cdot \Delta'$):

$$\boldsymbol{c}_0 + \boldsymbol{c}_1 \boldsymbol{s} \equiv \Delta_\ell^2 \cdot \mu + \boldsymbol{f}'.$$

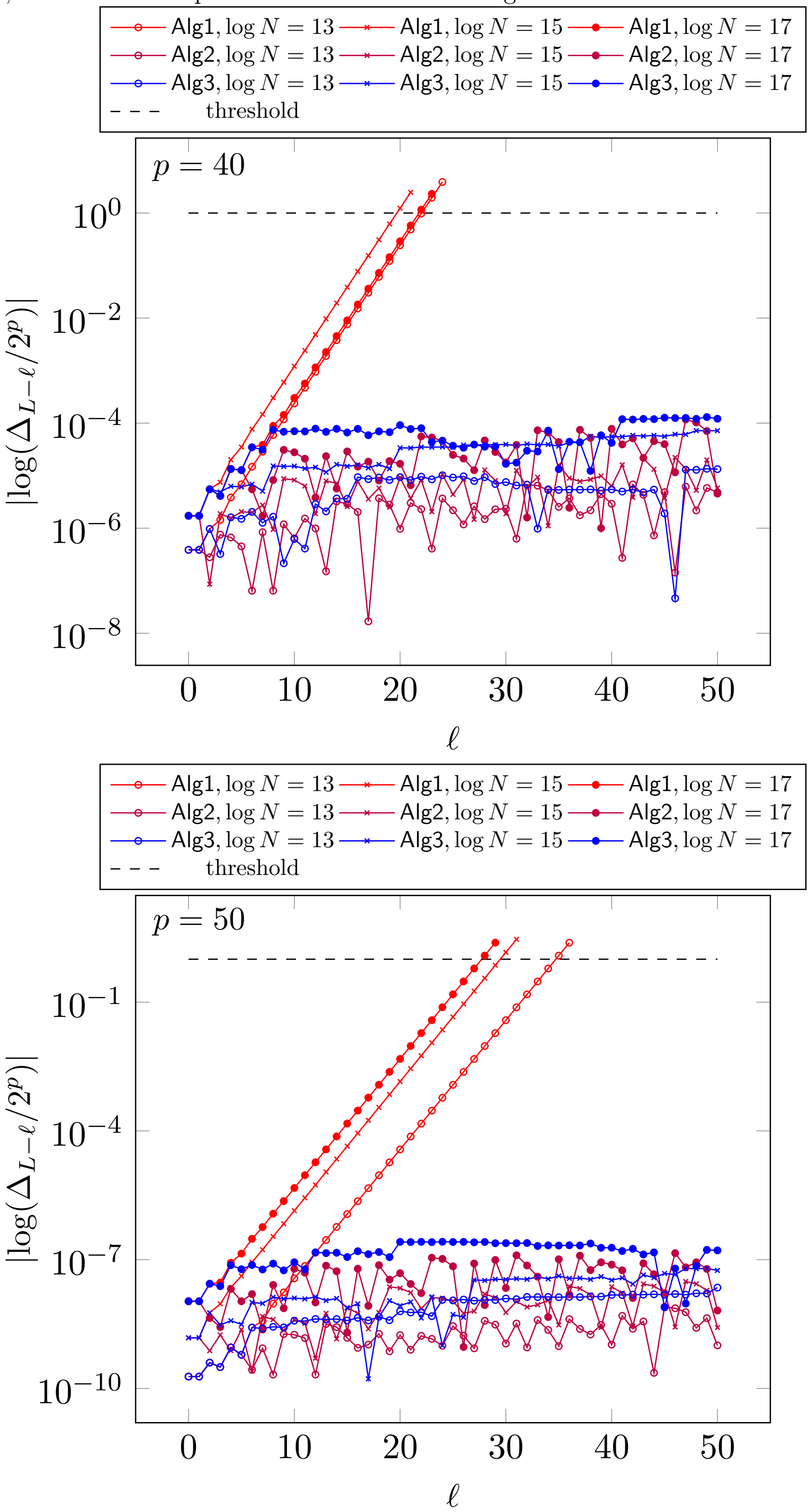
Let $\boldsymbol{f}'/\Delta_\ell^2 = \phi'^{(\ell)}$. After Mult' operation we have:

$$\begin{aligned}\boldsymbol{c}_{\mathsf{mult}',0} + \boldsymbol{c}_{\mathsf{mult}',1}\boldsymbol{s} &\equiv \left(\frac{\Delta_\ell^2 \cdot \mu_1 + \boldsymbol{f}_1'}{q_\ell} + \boldsymbol{r}_{1,\mathsf{rs}}\right) \cdot \left(\frac{\Delta_\ell^2 \cdot \mu_2 + \boldsymbol{f}_2'}{q_\ell} + \boldsymbol{r}_{2,\mathsf{rs}}\right) + \boldsymbol{e}_{\mathsf{ks}} \\ &= \left(\Delta_{\ell-1} \cdot \left(\mu_1 + \phi_1'\right) + \boldsymbol{r}_{1,\mathsf{rs}}\right) \cdot \left(\Delta_{\ell-1} \cdot \left(\mu_2 + \phi_2'\right) + \boldsymbol{r}_{2,\mathsf{rs}}\right) + \boldsymbol{e}_{\mathsf{ks}} \\ &= \Delta_{\ell-1}^2 \cdot \mu_1\mu_2 + \boldsymbol{f}_{\mathsf{mult}'},\end{aligned}$$

where

$$\begin{aligned}\boldsymbol{f}_{\mathsf{mult}'} = \Delta_{\ell-1}^2 \cdot \left(\mu_1\phi_2'^{(\ell)} + \mu_2\phi_1'^{(\ell)} + \phi_1'^{(\ell)}\phi_2'^{(\ell)}\right) + \\ + \Delta_{\ell-1} \cdot \left(\left(\mu_1 + \phi_1'^{(\ell)}\right)\boldsymbol{r}_{\mathsf{rs},2} + \left(\mu_2 + \phi_2'^{(\ell)}\right)\boldsymbol{r}_{\mathsf{rs},1}\right) + \boldsymbol{r}_{\mathsf{rs},1}\boldsymbol{r}_{\mathsf{rs},2} + \boldsymbol{e}_{\mathsf{ks}},\end{aligned}$$

Figure 1: Deviation of scaling factors from the base value 2^p for $p = 40$ and $p = 50$ and different values of ring dimension N; threshold corresponds to a factor of 2x change.



$$\phi_{\mathsf{mult}'} = \mu_1 \left(\phi_2'^{(\ell)} + \rho_{\mathsf{rs},2}^{(\ell-1)}\right) + \mu_2 \left(\phi_1'^{(\ell)} + \rho_{\mathsf{rs},1}^{(\ell-1)}\right) + \left(\phi_1'^{(\ell)} + \rho_{\mathsf{rs},1}^{(\ell-1)}\right)\left(\phi_2'^{(\ell)} + \rho_{\mathsf{rs},2}^{(\ell-1)}\right) + \frac{\epsilon_{\mathsf{ks}}^{(\ell-1)}}{\Delta_{\ell-1}}.$$

4.2.1 Handling the Operations between Ciphertexts at Different Levels for Reduced-Error CKKS

The same approach as in Section 4.1.2 can be applied to handle the operations between ciphertexts at different levels.

- $\mathsf{Adjust}(\mathsf{ct}_1, \ell_2)$. For a ciphertext ct_1 at level ℓ_1 and scaling factor $\Delta_{\ell_1}^2$, drop moduli $\{q_{\ell_2+2}, \ldots, q_{\ell_1}\}$, multiply the result by a constant $\left\lceil \frac{\Delta_{\ell_2}^2 \cdot q_{\ell_2+1}}{\Delta_{\ell_1}^2} \right\rfloor = \frac{\Delta_{\ell_2}^2 \cdot q_{\ell_2+1}}{\Delta_{\ell_1}^2} + \delta$, where $\delta \in [-1/2, 1/2]$, and finally rescale by q_{ℓ_2+1}.

Let a ciphertext $\mathsf{ct}_1 = (\boldsymbol{c}_0, \boldsymbol{c}_1)$ satisfy the following relation:

$$\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} = \Delta_{\ell_1}^2 \cdot \mu + \boldsymbol{f}' \pmod{Q_\ell}.$$

The adjustment procedure $\mathsf{ct}_{\mathsf{adjust}} = \mathsf{Adjust}(\mathsf{ct}_1, \ell_2)$ for a ciphertext ct_1 leads to the following relation:

$$\boldsymbol{c}_{\mathsf{adjust},0} + \boldsymbol{c}_{\mathsf{adjust},1} \cdot \boldsymbol{s} = \frac{1}{q_{\ell_2+1}} \left(\Delta_{\ell_1}^2 \cdot \mu + \boldsymbol{f}'\right) \cdot \left(\frac{\Delta_{\ell_2}^2 \cdot q_{\ell_2+1}}{\Delta_{\ell_1}^2} + \delta\right) + \boldsymbol{r}_{\mathsf{rs}} = \Delta_{\ell_2}^2 \cdot \mu + \boldsymbol{f}'_{\mathsf{adjust}} \pmod{Q_{\ell_2}},$$

with

$$\boldsymbol{f}'_{\mathsf{adjust}} = \frac{\Delta_{\ell_2}^2}{\Delta_{\ell_1}^2} \cdot \boldsymbol{f}' + \frac{\delta \Delta_{\ell_1}^2 \cdot \mu + \delta \boldsymbol{f}'}{q_{\ell_2+1}} + \boldsymbol{r}_{\mathsf{rs}},$$

where the second error term is introduced by scalar multiplication and error $\boldsymbol{r}_{\mathsf{rs}}$ is introduced by the rescaling. Then we have

$$\phi_{\mathsf{adjust}}'^{(\ell_2)} = \phi'^{(\ell_1)} + \frac{\delta \Delta_{\ell_1}^2 \cdot \mu}{\Delta_{\ell_2+1}^2 \Delta_{\ell_2}} + \frac{\delta \phi'^{(\ell_2+1)}}{\Delta_{\ell_2}} + \frac{\rho_{\mathsf{rs}}^{(\ell_2)}}{\Delta_{\ell_2}}.$$

We see that the rescaling part $\frac{\rho_{\mathsf{rs}}^{(\ell_2)}}{\Delta_{\ell_2}}$ becomes negligible.

5 Implementation Details and Results

We implemented both proposed RNS variants of CKKS in PALISADE and evaluated their performance using four representative benchmarks: addition of multiple vectors, summation over a vector, component-wise multiplication of multiple vectors, evaluation of a polynomial over a vector.

We introduce the following notation to distinguish between different RNS variants of CKKS:

- Reduced-Error CKKS with Delayed Exact (RE-CKKS-DE) rescaling: includes all techniques for reducing the approximation error presented in this work;

- CKKS with Delayed Exact (CKKS-DE) rescaling: includes only the RNS-specific techniques described in Section 4.1 + delayed rescaling;

- CKKS with Immediate Approximate (CKKS-IA) rescaling: classical RNS variant, as implemented in RNS-HEAAN and prior versions of PALISADE.

Note that the approximation error of CKKS-DE is approximately the same as the error of the multiprecision CKKS implementation in the HEAAN library. In our comparison of experimentally observed precision for CKKS-DE in PALISADE vs CKKS in the HEAAN library for selected computations (where delayed rescaling in CKKS-DE did not give any advantage to PALISADE over HEAAN), we did not observe differences higher than 0.2 bits, and the differences we saw were not statistically significant.

5.1 Setting the Parameters

The coefficients of error polynomials were sampled using the discrete Gaussian distribution with distribution parameter $\sigma = 3.2$. We used uniform ternary distribution for secrets, which is the most efficient setting that is compliant with the HE standard [2].

As noted previously, ${Q'}_L$ for RE-CKKS is larger than Q_L for original CKKS by Δ'. The value of Δ' in our experiments is approximately 2^{20}. This may lead to a doubled ring dimension for RE-CKKS as compared to CKKS in regions where the effective ciphertext modulus $P{Q'}_L$ is close to the LWE work factor threshold between two subsequent ring dimensions (see Table 1 of [2] for the threshold values). However, we can accommodate for this difference when selecting the auxiliary moduli for hybrid key switching, paying a relatively small price in the performance of key switching. For example, when we look at the benchmarks of addition of multiple vectors (Table 4) and summation over a vector (Table 5), we get $Q_L \approx 2^{40}$, $Q'_L \approx 2^{60}$, and $P \approx 2^{60}$, which implies that the effective ciphertext modulus for CKKS is $\approx 2^{100}$ while for RE-CKKS it is $\approx 2^{120}$. The threshold for $N = 2^{12}$ in the uniform ternary secret setting is $\approx 2^{109}$. We can change the effective modulus for RE-CKKS by reducing P to 2^{49} or less, which reduces the effective modulus to 2^{109} or lower, allowing us to use the same ring dimension as for CKKS.

5.2 Software Implementation and Experimental Setup

We implemented all proposed RNS variants of CKKS in PALISADE v1.10. The evaluation environment was a commodity desktop computer system with an Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz and 64 GB of RAM, running Ubuntu 18.04 LTS. The compiler was g++ 9.3.0. All experiments were executed in the single-threaded mode.

We ran the experiments in the full packing mode, i.e., we packed a vector $\mathbf{x} \in \mathbb{C}^{N/2}$ of size $N/2$ per ciphertext. The entries x_i were randomly generated from the complex unit circle $\{z \in \mathbb{C} : \|z\|_2 = 1\}$. To estimate the precision after the decryption output $\tilde{\mathbf{x}}$, we evaluated the average of $\|x_i - \tilde{x}_i\|_2$ and then computed the logarithm of it.

5.3 Experimental Results

Addition of multiple vectors. Table 4 compares the precision and runtimes for the use case of adding k vectors together for all four RNS variants. This use case does not require any key switching and rescaling operations, and illustrates the pure effect of eliminating fresh LWE encryption noise. The precision of RE-CKKS-DE is about 20 bits higher than for CKKS at $\Delta_i \approx 2^{40}$ for all considered values of k, which implies that $\Delta' = 2^{20}$ gives us a direct improvement in precision. For $\Delta_i \approx 2^{50}$, we get a smaller improvement in precision because of the 52-bit precision of the double-precision

Table 4: Comparison of precision and runtime when computing $\sum_{i=0}^{k} \mathbf{x}_i$ for Reduced-Error CKKS with Delayed Exact (RE-CKKS-DE) rescaling, CKKS with Delayed Exact (CKKS-DE) rescaling, and CKKS with Immediate Approximate (CKKS-IA) rescaling RNS variants; CKKS-DE has the same approximation error as the multiprecision CKKS implementation in the HEAAN library and CKKS-IA is equivalent to the RNS implementation in RNS-HEAAN and previous versions of PALISADE; $\Delta_i \approx 2^p, q_0 \approx 2^{60}, \Delta' \approx 2^{20}, K = \lceil \log Q_L \rceil, \lambda > 128$ bits.

		RE-CKKS-DE				CKKS-DE				CKKS-IA	
p	k	$\log N$	K	prec.	time	$\log N$	K	prec.	time	prec.	time
40	2	12	60	45.8	0.04 ms	12	40	25.9	0.02 ms	25.9	0.02 ms
	4	12	60	45.3	0.11 ms	12	40	25.4	0.06 ms	25.4	0.04 ms
	8	12	60	44.9	0.24 ms	12	40	24.9	0.12 ms	24.9	0.08 ms
	16	12	60	44.3	0.51 ms	12	40	24.4	0.25 ms	24.4	0.17 ms
	32	12	60	43.8	1.06 ms	12	40	23.9	0.51 ms	23.9	0.34 ms
	64	12	60	43.3	2.2 ms	12	40	23.4	1.07 ms	23.4	0.74 ms
50	2	13	70	48.1	0.08 ms	13	50	34.9	0.04 ms	34.9	0.03 ms
	4	13	70	48.4	0.22 ms	13	50	34.4	0.11 ms	34.4	0.07 ms
	8	13	70	48.0	0.48 ms	13	50	33.9	0.23 ms	33.9	0.16 ms
	16	13	70	49.6	1.04 ms	13	50	33.4	0.53 ms	33.4	0.34 ms
	32	13	70	48.9	2.16 ms	13	50	32.9	1.1 ms	32.9	0.76 ms
	64	13	70	48.1	4.42 ms	13	50	32.4	2.17 ms	32.4	1.54 ms

floating-point arithmetic used to represent real numbers. The precision is reduced from 52 to roughly 48-49 bits because of the decoding error $\boldsymbol{r}_{\mathsf{float}}$. The runtime slowdown of RE-CKKS-DE vs CKKS-DE for both values of Δ_i is exactly 2x because RE-CKKS-DE works with two RNS limbs (the regular one + the extra modulus Δ'). This slowdown for $\Delta_i \approx 2^{40}$ can be removed by working with a composite modulus $q_0\Delta' \approx 2^{60}$ as it fits a single 64-bit word. But we did not implement this optimization as it only works for special cases, and the runtime of about 1 ms is already very small for practical purposes. There is also some performance improvement for CKKS-IA as compared to CCKS-DE, but it is determined by how the code is written (extra memory allocations are done in the case of CCKS-DE) and has no algorithmic cause.

Table 5: Comparison of precision and runtime when computing $\sum_{i=0}^{N/2} x_i$ for RE-CKKS-DE, CKKS-DE, and CKKS-IA RNS variants (see Table 4 for the definition of RNS variants); $\Delta_i \approx 2^p \approx q_0, \Delta' \approx 2^{20}, K = \lceil \log Q_L \rceil, \lambda > 128$ bits.

	RE-CKKS-DE				CKKS-DE				CKKS-IA	
p	$\log N$	K	prec.	time	$\log N$	K	prec.	time	prec.	time
40	12	60	40.4	17.33 ms	12	40	21.1	8.94 ms	21.1	8.89 ms
50	13	70	47.2	38.67 ms	13	50	28.3	19.79 ms	28.3	19.77 ms

Summation over a vector. Table 5 shows the precision and runtimes for the computation adding up all components of a vector. This use case requires key switching but does not need to rescale as there are no multiplications involved. We can see that the precision improvement of RE-CKKS-DE over CKKS-DE is still about 20 bits for $\Delta_i \approx 2^{40}$ and it is slightly smaller for $\Delta_i \approx 2^{50}$ due to the floating-point approximation error. This implies that Δ' removes both encryption and key switching LWE approximation errors, and we only deal with the floating-point precision error here. The runtime slowdown of RE-CKKS-DE compared to all other RNS variants is slightly under 2x. It can be attributed to the extra modulus Δ' and increased computational complexity of hybrid key switching related to this.

Table 6: Comparison of precision and runtime when computing $\prod_{i=1}^{2^k} \mathbf{x}_i$ for RE-CKKS-DE, CKKS-DE, and CKKS-IA RNS variants (see Table 4 for the definition of RNS variants); $\Delta_i \approx 2^p, q_0 \approx 2^{60}, \Delta' \approx 2^{20}, K = \lceil \log Q_L \rceil, \lambda > 128$ bits.

		RE-CKKS-DE				CKKS-DE				CKKS-IA	
p	k	$\log N$	K	prec.	time	$\log N$	K	prec.	time	prec.	time
40	1	13	120	28.9	5.61 ms	13	100	24.9	3.24 ms	21.8	4.01 ms
	2	14	160	27.1	47.85 ms	14	140	23.4	32.93 ms	20.1	34.25 ms
	3	14	200	26.5	0.14 s	14	180	22.9	99.47 ms	20.7	0.1 s
	4	14	240	26.0	0.38 s	14	220	22.4	0.29 s	17.8	0.29 s
	5	14	280	25.4	0.91 s	14	260	21.9	0.73 s	17.3	0.73 s
	6	14	320	24.9	2.29 s	14	300	21.4	1.79 s	15.9	1.78 s
	7	15	360	23.4	11.27 s	15	340	19.9	8.94 s	14.3	8.86 s
50	1	13	130	38.9	6.22 ms	13	110	34.9	3.17 ms	32.8	4 ms
	2	14	180	37.1	47.83 ms	14	160	33.4	32.77 ms	32.3	34.23 ms
	3	14	230	36.5	0.14 s	14	210	32.9	0.1 s	29.0	0.1 s
	4	14	280	36.0	0.38 s	14	260	32.4	0.29 s	29.5	0.29 s
	5	14	330	35.4	0.97 s	14	310	31.9	0.73 s	27.7	0.73 s
	6	15	380	33.9	4.8 s	15	360	30.4	3.76 s	27.3	3.73 s
	7	15	430	33.4	11.3 s	15	410	29.9	8.94 s	25.9	8.85 s

Binary tree multiplication. Table 6 illustrates the precision and runtimes for the case of binary tree multiplication. This uses case examines the effect of reduced approximation error for the multiplication operation followed by key switching and rescaling. First, we want to point out that the precision improvement of RE-CKKS-DE over CKKS-DE is about 3.5 to 4 bits (with the highest precision gain after the first multiplication), as theoretically predicted in Section 3.3. Second, CKKS-DE gains additional 3 to 6 bits over CKKS-IA. This implies that the RE-CKKS-DE RNS variant can be up to 9 bits more precise than the prior RNS variants. The performance penalty of higher precision does not typically exceed 25-30%, which is a relatively small cost.

Evaluation of a polynomial over a vector. Table 7 shows the precision and runtimes for the case of evaluating a polynomial over a vector of real numbers, which is a very common operation in CKKS as a polynomial approximation is often used to approximate ”hard”, nonlinear functions, such as logistic function, multiplicative inverse, sine wave, etc. This use case examines the combined effect of multiplications and cross-level additions. We can observe that the precision gain of RE-CKKS-DE over CKKS-DE is still 3.5-4 bits. The precision gain of CKKS-DE over CKKS-IA is less

Table 7: Comparison of precision and runtime when computing $\sum_{i=0}^{k} \mathbf{x}^i$ for RE-CKKS-DE, CKKS-DE, and CKKS-IA RNS variants (see Table 4 for the definition of RNS variants); $\Delta_i \approx 2^p, q_0 \approx 2^{60}, \Delta' \approx 2^{20}, K = \lceil \log Q_L \rceil, \lambda > 128$ bits.

		RE-CKKS-DE				CKKS-DE				CKKS-IA	
p	k	$\log N$	K	prec.	time	$\log N$	K	prec.	time	prec.	time
40	2	13	120	28.4	7 ms	13	100	24.3	3.37 ms	21.8	4.14 ms
	4	14	160	26.1	50.88 ms	14	140	22.2	35.63 ms	19.4	29.39 ms
	8	14	200	24.8	0.13 s	14	180	21.0	0.1 s	19.1	75.14 ms
	16	14	240	23.6	0.29 s	14	220	19.8	0.26 s	16.9	0.17 s
	32	14	280	22.4	0.64 s	14	260	18.6	0.58 s	16.3	0.38 s
	48	14	320	21.8	1.12 s	14	300	17.9	1.05 s	15.1	0.67 s
	64	14	320	21.3	1.33 s	14	300	17.4	1.26 s	14.9	0.82 s
50	2	13	130	38.4	7.69 ms	13	110	34.3	3.36 ms	32.8	4.14 ms
	4	14	180	36.0	51.19 ms	14	160	32.1	35.8 ms	29.5	29.48 ms
	8	14	230	34.8	0.13 s	14	210	31.0	0.1 s	28.0	75.42 ms
	16	14	280	33.6	0.29 s	14	260	29.8	0.26 s	27.7	0.17 s
	32	14	330	32.4	0.68 s	14	310	28.6	0.58 s	26.4	0.38 s
	48	15	380	30.8	2.34 s	15	360	26.9	2.21 s	26.1	1.4 s
	64	15	380	30.3	2.78 s	15	360	26.4	2.65 s	25.6	1.72 s

pronounced in this case (not higher than 3 bits). The performance penalty is the worst for smaller-degree polynomials (up to 2x), but becomes somewhat smaller for larger-degree polynomials.

6 Concluding Remarks

Our results suggest that a relatively high precision can be achieved for RE-CKKS in RNS for significantly smaller scaling factors than in the original CKKS scheme and its prior RNS variants. This implies RE-CKKS requires smaller ciphertext moduli to achieve the same precision as the original CKKS or its RNS instantiation, which may yield certain concrete performance improvements.

Another benefit of RE-CKKS is that it can be used to increase the CKKS bootstrapping precision in RNS variants of CKKS, which is currently a major practical limitation for the RNS instantiations of CKKS [21]. For example, the extra 6 to 9 bits may provide enough room for more accurate polynomial approximations of the modular reduction function. But the precision improvements in CKKS bootstrapping require careful modifications at various stages of the bootstrapping procedure, e.g., in the scaling operations. Hence this problem deserves a separate study and is beyond the scope of our present work.

The main motivation of our study was to improve the usability of the CKKS scheme by eliminating several approximation errors and automating the execution of rescaling. We believe we have achieved this goal, and consider our work as a significant step towards making the CKKS scheme more practical. For instance, all operations related to rescaling or the approximation error management are completely hidden from the application developer in our PALISADE implementation, and the API for CKKS is the same as for integer-arithmetic homomorphic encryption schemes, such as Brakersky-Gentry-Vaikuntanathan [7] and Brakerski/Fan-Vercauteren [6, 17] schemes.

References

[1] PALISADE Lattice Cryptography Library (release 1.10.3). `https://palisade-crypto.org/`, 2020.

[2] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[3] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca. A full RNS variant of FV like somewhat homomorphic encryption schemes. In *International Conference on Selected Areas in Cryptography*, pages 423–442. Springer, 2016.

[4] M. Blatt, A. Gusev, Y. Polyakov, and S. Goldwasser. Secure large-scale genome-wide association studies using homomorphic encryption. *Proceedings of the National Academy of Sciences*, 117(21):11608–11613, 2020.

[5] M. Blatt, A. Gusev, Y. Polyakov, K. Rohloff, and V. Vaikuntanathan. Optimized homomorphic encryption solution for secure genome-wide association studies. *BMC Medical Genomics*, 13(7):1–13, 2020.

[6] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Annual Cryptology Conference*, pages 868–886. Springer, 2012.

[7] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[8] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Annual cryptology conference*, pages 505–524. Springer, 2011.

[9] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song. Bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 360–384. Springer, 2018.

[10] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song. A full RNS variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*, pages 347–368. Springer, 2018.

[11] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song. RNSHEAAN, 2018. `https://github.com/KyoohyungHan/FullRNS-HEAAN`.

[12] J. H. Cheon, A. Kim, M. Kim, and Y. Song. HEAAN, 2016. `https://github.com/snucrypto/HEAAN`.

[13] J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017.

[14] R. Cohen, J. Frankle, S. Goldwasser, H. Shaul, and V. Vaikuntanathan. How to trade efficiency and accuracy using fault-tolerant computations over the reals, 2019. `https://crypto.iacr.org/2019/affevents/ppml/page.html`.

[15] A. Costache and N. P. Smart. Which ring based somewhat homomorphic encryption scheme is best? In *Cryptographers' Track at the RSA Conference*, pages 325–340. Springer, 2016.

[16] B. R. Curtis and R. Player. On the feasibility and impact of standardising sparse-secret LWE parameter sets for homomorphic encryption. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 1–10, 2019.

[17] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012:144, 2012.

[18] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In *Annual Cryptology Conference*, pages 850–867. Springer, 2012.

[19] S. Halevi, Y. Polyakov, and V. Shoup. An improved RNS variant of the BFV homomorphic encryption scheme. In *Cryptographers' Track at the RSA Conference*, pages 83–105. Springer, 2019.

[20] S. Halevi and V. Shoup. HElib, 2014. `https://github.com/homenc/HElib`.

[21] K. Han and D. Ki. Better bootstrapping for approximate homomorphic encryption. In *Cryptographers' Track at the RSA Conference*, pages 364–390. Springer, 2020.

[22] A. Kim, Y. Song, M. Kim, K. Lee, and J. H. Cheon. Logistic regression model training based on the approximate homomorphic encryption. *BMC medical genomics*, 11(4):83, 2018.

[23] M. Kim, Y. Song, B. Li, and D. Micciancio. Semi-parallel logistic regression for GWAS on encrypted data. *BMC Medical Genomics*, 13(7):1–13, 2020.

[24] Microsoft SEAL, 2020. `https://github.com/Microsoft/SEAL`.

[25] Y. Son and J. H. Cheon. Revisiting the hybrid attack on sparse secret LWE and application to HE parameters. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 11–20, 2019.

[26] Y. Song. The CKKS (a.k.a. HEAAN) FHE scheme, 2020. `https://simons.berkeley.edu/talks/heaan-fhe`.

A Approximate Scaling Error in RNS

Consider approximate plaintexts $\boldsymbol{m}_1 = 2^p \cdot \mu_1 + \boldsymbol{e}_1$ and $\boldsymbol{m}_2 = 2^p \cdot \mu_2 + \boldsymbol{e}_2$.

When we multiply the plaintexts, we get

$$\boldsymbol{m}_1 \cdot \boldsymbol{m}_2 \approx 2^{2p} \cdot \mu_1\mu_2 + 2^p \cdot \mu_1\boldsymbol{e}_2 + 2^p \cdot \mu_2\boldsymbol{e}_1.$$

Choose RNS moduli q_i such that $2^p/q_i$ stays in the range $(1 - 2^{-\epsilon}, 1 + 2^{-\epsilon})$, where $2^{-\epsilon}$ is kept as small as possible.

The rescaling at level ℓ for both cases can be written as (the rounding error is ignored in this analysis)

$$\frac{\boldsymbol{m}_1 \cdot \boldsymbol{m}_2}{2^p} \approx 2^p \cdot \mu_1\mu_2 + \mu_1\boldsymbol{e}_2 + \mu_2\boldsymbol{e}_1,$$

$$\frac{\boldsymbol{m}_1 \cdot \boldsymbol{m}_2}{q_\ell} \approx \frac{2^{2p}}{q_\ell} \cdot \mu_1\mu_2 + \frac{2^p}{q_\ell} \cdot \mu_1\boldsymbol{e}_2 + \frac{2^p}{q_\ell} \cdot \mu_2\boldsymbol{e}_1.$$

If we ignore the noise terms, the difference between $\frac{\boldsymbol{m}_1 \cdot \boldsymbol{m}_2}{2^p}$ and $\frac{\boldsymbol{m}_1 \cdot \boldsymbol{m}_2}{q_\ell}$ can be written as

$$\left| 2^p \cdot \mu_1\mu_2 - \frac{2^{2p}}{q_\ell} \cdot \mu_1\mu_2 \right| \leq 2^{-\epsilon} \cdot 2^p \cdot \mu_1\mu_2 = 2^{p-\epsilon} \cdot \mu_1\mu_2.$$

In other words, we introduce an approximation error of roughly $p - \epsilon$ bits. In the RNS variants, $p - \epsilon$ is typically larger than the number of bits in the CKKS LWE approximation error. In other words,

$$\left| 2^{-\epsilon} \cdot 2^p \cdot \mu_1\mu_2 \right| > |\mu_1\boldsymbol{e}_2| + |\mu_2\boldsymbol{e}_1|.$$

Next we show what happens when we apply two rescaling operations. We use ϵ_i for each q_i. First, we multiply two approximate plaintexts (after initial approximate rescaling by q_ℓ) and get the following scaling factor

$$\left(2^p + 2^{p-\epsilon_\ell}\right)^2 \approx 2^{2p} + 2^{2p-\epsilon_\ell+1}.$$

After second rescaling, we have (the noise terms are dropped for simplicity):

$$\begin{aligned}
\left\lceil \frac{\left\lceil \frac{\boldsymbol{m}_1 \cdot \boldsymbol{m}_2}{q_\ell} \right\rfloor \left\lceil \frac{\boldsymbol{m}_3 \cdot \boldsymbol{m}_4}{q_\ell} \right\rfloor}{q_{\ell-1}} \right\rfloor &\approx \left\| \left\lceil \frac{2^p + 2^{p-\epsilon_\ell+1}}{q_{\ell-1}} \cdot \left(2^p \cdot \prod_{i=1}^{4} \mu_i \right) \right\rfloor \right\| \\
&\approx \left\| \left\lceil \frac{2^p}{q_{\ell-1}} \left(1 + 2^{-\epsilon_\ell+1}\right) \cdot \left(2^p \cdot \prod_{i=1}^{4} \mu_i \right) \right\rfloor \right\| \\
&\approx \left\| \left\lceil \left(1 + 2^{-\epsilon_{\ell-1}}\right) \cdot \left(1 + 2^{-\epsilon_\ell+1}\right) \cdot \left(2^p \cdot \prod_{i=1}^{4} \mu_i \right) \right\rfloor \right\| \\
&\approx \left\| \left\lceil \left(1 + 2^{-\epsilon_{\ell-1}} + 2^{-\epsilon_\ell+1}\right) \cdot \left(2^p \cdot \prod_{i=1}^{4} \mu_i \right) \right\rfloor \right\|.
\end{aligned}$$

So the error in this case is bounded by

$$\left(2^{-\epsilon_{\ell-1}} + 2^{-\epsilon_\ell+1}\right) \cdot \left\| \left\lceil 2^p \cdot \prod_{i=1}^{4} \mu_i \right\rfloor \right\|.$$

After three rescaling operations, we have

$$\left(2^{-\epsilon_{\ell-2}} + 2^{-\epsilon_{\ell-1}+1} + 2^{-\epsilon_\ell+2}\right) \cdot \left\| \left\lceil 2^p \cdot \prod_{i=1}^{8} \mu_i \right\rfloor \right\|.$$

This analysis implied that all moduli are incrementally (monotonously) increased or decreased, which is the worst case. In the RNS variants of CKKS [5, 10], we alternate the moduli w.r.t. 2^p. So instead we should expect something like this.

After two rescaling operations:

$$\left(-2^{-\epsilon_{\ell-1}}+2^{-\epsilon_\ell+1}\right)\cdot\left\|\left\lceil 2^p\cdot\prod_{i=1}^{4}\mu_i\right\rfloor\right\|$$

After three rescaling operations:

$$\left(2^{-\epsilon_{\ell-2}}-2^{-\epsilon_{\ell-1}+1}+2^{-\epsilon_\ell+2}\right)\cdot\left\|\left\lceil 2^p\cdot\prod_{i=1}^{8}\mu_i\right\rfloor\right\|$$

In this case, the approximation error grows more slowly. For instance, if we assume that $\epsilon_{\ell-2}\approx\epsilon_{\ell-1}\approx\epsilon_\ell$, then the error after 3 rescaling operations will be roughly 3x the error after the first rescaling operation, whereas in the monotonic case it would be about 7x. In reality, the values of current ϵ_ℓ become progressively smaller and, hence, the first error term becomes larger. The optimal choice of moduli is more involving. So in the implementation we use a relatively simple alternating logic, and the largest values of ϵ_ℓ are used for the initial levels (last RNS moduli) to keep the approximation error as small as possible.

Although one could find the optimal value of prime moduli that would give the lowest approximation error for the logic described above, in practical scenarios we deal with two methods for modulus switching: rescaling and simple level reduction (w/o rescaling). In this case, the choice of prime moduli resulting in the lowest approximation error may be different from the scenarios with normal rescaling only.

B Proofs of Lemmas

We follow the heuristic approach in [13,15,18]. Assume that a polynomial $\boldsymbol{a}$ is sampled from some distribution with independent and identically distributed entries. Since $\boldsymbol{a}(\zeta_M)$ is the inner product of coefficient vector of $\boldsymbol{a}$ and fixed vector $(1,\zeta_M,\ldots,\zeta_M^{N-1})$ of Euclidean norm $\sqrt{N}$, the random variable $\boldsymbol{a}(\zeta_M)$ has variance $\sigma^2 N$, where σ^2 is the variance of each coefficient of $\boldsymbol{a}$. Moreover, we can assume that $\boldsymbol{a}(\zeta_M)$ is distributed similarly to a gaussian distribution over complex plane since it is a sum of many independent and identically distributed entries. We will use the following bound $\|\boldsymbol{a}\|^{\mathsf{can}}\leq 6\sigma\sqrt{N}$ for the canonical embedding norm of $\boldsymbol{a}$. For a multiplication of two independent polynomials $\boldsymbol{a}$, $\boldsymbol{b}$ close to gaussian distributions with variances σ_1^2 and σ_2^2, we will use a high-probability bound $\|\boldsymbol{a}\cdot\boldsymbol{b}\|^{\mathsf{can}}\leq 16\sigma_1\sigma_2 N$.

Proof of Lemma 3.1. We choose binary $\boldsymbol{v}\leftarrow\chi_{\mathsf{enc}}$, and discrete gaussian $\boldsymbol{e}_0,\boldsymbol{e}_1\leftarrow\chi_{\mathsf{err}}$, then set $\mathsf{ct}=\boldsymbol{v}\cdot\mathsf{pk}+(\boldsymbol{m}+\boldsymbol{e}_0,\boldsymbol{e}_1)$. The bound of $\boldsymbol{f}_{\mathsf{fresh}}$ of fresh encryption noise is computed by the following inequality

$$\begin{aligned}\|\boldsymbol{f}_{\mathsf{fresh}}\|^{\mathsf{can}}&=\|\boldsymbol{v}\cdot\boldsymbol{e}+\boldsymbol{e}_0+\boldsymbol{e}_1\cdot\boldsymbol{s}\|^{\mathsf{can}}\leq\|\boldsymbol{v}\cdot\boldsymbol{e}\|^{\mathsf{can}}+\|\boldsymbol{e}_0\|^{\mathsf{can}}+\|\boldsymbol{e}_1\cdot\boldsymbol{s}\|^{\mathsf{can}}\\&\leq 16\cdot\sqrt{\frac{2}{3}}\sigma N+6\sigma\sqrt{N}+16\cdot\sqrt{\frac{2}{3}}\sigma N=\frac{32}{3}\sqrt{6}\sigma N+6\sigma\sqrt{N}.\end{aligned}$$

QED.

Proof of Lemma 3.2. The key switching noise comes from the error terms $\{\boldsymbol{e}'\}$ in swk_1 and from rounding parts that we denote by $\boldsymbol{r}_0,\boldsymbol{r}_1$ with coefficients smaller than $1/2$. The bound of $\boldsymbol{e}_{\mathsf{ks}}$ of key switching noise is computed by the following inequality

$$\begin{aligned}\|\boldsymbol{e}_{\mathsf{ks}}\|^{\mathsf{can}} &= \left\|\frac{\langle \mathcal{WD}_\ell\left(\boldsymbol{c}_1\right), \boldsymbol{e}'\rangle}{P} + \boldsymbol{r}_0 + \boldsymbol{r}_1 \cdot \boldsymbol{s}\right\|^{\mathsf{can}} \leq \left\|\frac{\langle \mathcal{WD}_\ell\left(\boldsymbol{c}_1\right), \boldsymbol{e}'\rangle}{P}\right\|^{\mathsf{can}} + \|\boldsymbol{r}_0\|^{\mathsf{can}} + \|\boldsymbol{r}_1 \cdot \boldsymbol{s}\|^{\mathsf{can}} \\ &\leq \frac{16 \cdot \mathsf{dnum} \cdot \omega\sigma N}{\sqrt{12}P} + 6 \cdot \sqrt{\frac{1}{12}N} + 16 \cdot \sqrt{\frac{1}{12}}\sqrt{\frac{2}{3}}N = \frac{8\sqrt{3} \cdot \mathsf{dnum} \cdot \omega\sigma N}{3P} + \sqrt{3N} + \frac{8\sqrt{2}N}{3}.\end{aligned}$$

QED.

Proof of Lemma 3.3. The bound of $\boldsymbol{r}_{\mathsf{rs}}$ is computed by the following inequality

$$\begin{aligned}\|\boldsymbol{r}_{\mathsf{rs}}\|^{\mathsf{can}} &= \|\boldsymbol{r}_0 + \boldsymbol{r}_1 \cdot \boldsymbol{s}\|^{\mathsf{can}} \leq \|\boldsymbol{r}_0\|^{\mathsf{can}} + \|\boldsymbol{r}_1 \cdot \boldsymbol{s}\|^{\mathsf{can}} \\ &\leq 6 \cdot \sqrt{\frac{1}{12}N} + 16 \cdot \sqrt{\frac{1}{12}}\sqrt{\frac{2}{3}}N = \sqrt{3N} + \frac{8\sqrt{2}N}{3}\end{aligned}$$

QED.