High-precision RNS-CKKS on fixed but smaller word-size architectures: theory and application*

Rashmi Agrawal
rashmi.agrawal@intel.com
Intel Labs

Jung Ho Ahn
gajh@snu.ac.kr
Seoul National University

Flavio Bergamaschi
flavio@intel.com
Intel Labs

Ro Cammarota
rosario.cammarota@intel.com
Intel Labs

Jung Hee Cheon
jhcheon@snu.ac.kr
Seoul National University
CryptoLab Inc.

Fillipe D. M. de Souza
fillipe.souza@intel.com
Intel Labs

Huijing Gong
huijing.gong@intel.com
Intel Labs

Minsik Kang
kaiser351@snu.ac.kr
Seoul National University

Duhyeong Kim
duhyeong.kim@intel.com
Intel Labs

Jongmin Kim
jongmin.kim@snu.ac.kr
Seoul National University

Hubert de Lassus
hubert.de.lassus@intel.com
Intel Labs

Jai Hyun Park
jhyunp@snu.ac.kr
Seoul National University

Michael Steiner
michael.steiner@intel.com
Intel Labs

Wen Wang
wen.wang@intel.com
Intel Labs

ABSTRACT

A prevalent issue in the residue number system (RNS) variant of the Cheon-Kim-Kim-Song (CKKS) homomorphic encryption (HE) scheme is the challenge of efficiently achieving high precision on hardware architectures with a fixed, yet smaller, word-size of bit-length W, especially when the scaling factor satisfies $\log \Delta > W$.

In this work, we introduce an efficient solution termed composite scaling. In this approach, we group multiple RNS primes as $q_\ell := \prod_{j=0}^{t-1} q_{\ell,j}$ such that $\log q_{\ell,j} < W$ for $0 \leq j < t$, and use each composite q_ℓ in the rescaling procedure as $\mathrm{ct} \mapsto \lfloor \mathrm{ct}/q_\ell \rceil$. Here, the number of primes, denoted by t, is termed the composition degree. This strategy contrasts the traditional rescaling method in RNS-CKKS, where each q_ℓ is chosen as a single $\log \Delta$-bit prime, a method we designate as single scaling.

To achieve higher precision in single scaling, where $\log \Delta > W$, one would either need a novel hardware architecture with word size $W' > \log \Delta$ or would have to resort to relatively inefficient solutions rooted in multi-precision arithmetic. This problem, however, doesn't arise in composite scaling. In the composite scaling approach, the larger the composition degree t, the greater the precision attainable with RNS-CKKS across an extensive range of secure parameters tailored for workload deployment.

We have integrated composite scaling RNS-CKKS into both OpenFHE and Lattigo libraries. This integration was achieved via a concrete implementation of the method and its application to the most up-to-date workloads, specifically, logistic regression training and convolutional neural network inference. Our experiments demonstrate that single and composite scaling approach are functionally equivalent, both theoretically and practically.

CCS CONCEPTS

• **Security and privacy** → **Domain-specific security and privacy architectures**;

KEYWORDS

Fully Homomorphic Encryption; High-precision CKKS; Fixed-word Size Architecture; Composite Scaling

1 INTRODUCTION

CKKS [13] is a fully homomorphic encryption (FHE) scheme that supports fixed-point arithmetic operations on real and complex numbers while they are encrypted. It is widely adopted in various fields, especially for real-world privacy-preserving applications including, but not limited to machine learning. Leading FHE libraries [2, 28, 33] implement a full-RNS variant of CKKS (referred to as RNS-CKKS) [12]. This variant expresses each polynomial over super-large modulus as a vector of multiple polynomials over small moduli. Since all operations in RNS-CKKS are done with these small modulo spaces, it offers a significant computational efficiency advantage compared over the basic CKKS scheme.

In RNS-CKKS, several algorithmic modifications differentiate it from the basic CKKS, leading to a distinct precision analysis compared to basic CKKS version. Specifically, the basic CKKS features a homomorphic operation called *Rescaling*. This operation divides

*This research was, in part, funded by the Defense Advanced Research Projects Agency (DARPA) through contract HR0011-21-3-0003. The views, opinions, and findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited).

the ciphertext by $1/\Delta$, where the CKKS parameter $\Delta > 0$ - known as scaling factor - determines the initial bit precision. In contrast, RNS-CKKS replaces this procedure with division by a prime q_i instead of Δ. To attain precision comparable to the basic CKKS, Cheon et al. [12] suggests to set q_i to closely approximate Δ. This approach is particularly logical when the scaling factor Δ is smaller than the word size (W). In such cases, each q_i can be set as an appropriate $\log \Delta$-bit prime. However, for high-precision arithmetic in RNS-CKKS, where Δ may exceed the word size, the solution proposed in [12] might fall short. This is because each prime would surpass the given word size.

As diverse RNS-CKKS applications are being actively developed, both in the literature and practice, progressively higher precision is required for RNS-CKKS over time to support applications such as machine learning training and multi-resolution imaging. There exist two different approaches to implement high-precision RNS-CKKS on a fixed, small word-size architecture. The first approach is in software [20], wherein we can use multi-precision arithmetic with the base of a given word size (say 2^{32}) to emulate an implementation with a larger word size (say 2^{64}). The second approach is to leverage the modular arithmetic circuit design principles, wherein one can compose the existing functional units to larger word-size functional units by assuming that composition arithmetic units are possible in the target architecture [1]. For example, a 64-bit modular multiplier can be realized by composing two 32-bit modular multipliers.

In this work, we pose and solve the following problem, which is a common issue for all FHE libraries implementing RNS-CKKS: *How can we efficiently achieve higher-precision fully homomorphic encryption (FHE) for fixed and smaller bit-length architectures?* Moreover, the precision vs. performance trade-off has never been fully investigated until now. To succinctly perform this investigation, we first need to describe a way to implement high-precision RNS-CKKS on a given fixed but small word-size architecture. We choose $W = 32$ bit, although our technique holds for any fixed bit-length, including classic power of two, e.g., $W = 64$ and crafted word sizes [23]. To this end, we first describe a more general approach, a.k.a. *composite scaling*. Then we address the challenges algorithmically while continuing to use the fixed, small word-size architectures. In composite scaling, we can group multiple RNS primes and use their product as scaling factor Δ in RNS-CKKS. We define the number of primes that are grouped together as the *composition degree*. The achievable precision in RNS-CKKS is directly proportional to the composition degree, as higher the composition degree, higher is the precision that can be achieved. Moreover, the composite scaling approach allows to trade-off precision for performance on fixed word-size architectures for a range of applications with different precision requirements without compromising security.

We enable composite scaling RNS-CKKS in the OpenFHE [2] and LattiGo [28] libraries through the concrete implementation of the method and known-to-date workloads, namely logistic regression training using the 2014 US Infant Mortality dataset, and convolutional neural network inference using the CIFAR-10 dataset. We illustrate through experiments that single and composite scaling are equivalent in terms of precision loss during homomorphic computation including bootstrapping, not only in theory but also in practice.

Table 1: Key CKKS notations and parameters.

Notation	Description
ct	Ciphertext.
swk	Key-switching key.
n	Length of a vector message, $n \leq \frac{N}{2}$.
N	Degree of power-of-two, # (number) of coefficients in a polynomial.
$\mathcal{R}_q$	$\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$ for an integer q, where $\mathcal{R}$ denotes the cyclotomic ring $\mathbb{Z}[X]/\langle X^N + 1\rangle$.
$\|\cdot\|_\infty^{\mathsf{can}}$	Canonical embedding norm for elements in $\mathbb{R}[X]/\langle X^N+1\rangle$. A formal definition is provided in [13].
h	Hamming weight of the ternary secret $s \in \{0, \pm 1\}^N[X]$ from a secret key distribution χ_{key} over $\mathcal{R}$.
σ	Standard deviation of an error distribution χ_{err} over $\mathcal{R}$.
Δ	Scale multiplied by the message during encryption.
ℓ	(Current) ciphertext level for $0 \leq \ell \leq L$, where L denotes the largest level.
Q_ℓ	A modulus at the ℓ-level with $Q_\ell = \prod_{i=0}^{\ell} q_i$, where each q_i corresponds to one level. A top-level modulus Q_L is simply denoted as Q.
q_i	One level in modulus $Q = \prod_{i=0}^{L} q_i$.
$q_{\ell,j}$	Each RNS prime composing one level $q_\ell = \prod_{j=0}^{t-1} q_{\ell,j}$.
t	# of RNS primes in a composite level q_ℓ.
P	Auxiliary modulus used for key-switching in $\mathcal{R}_{PQ}$.
p_i	Each RNS prime in modulus $P = \prod_{i=0}^{K-1} p_i$.
K	# of RNS primes p_i's composing P.
dnum	Decomposition number of the modulus $Q = \prod_{i=0}^{\mathsf{dnum}-1} \tilde{Q}_j$.
α	# of the factors in each partial product $\tilde{Q}_j = \prod_{i=j\alpha}^{(j+1)\alpha-1} q_i$ satisfying $\alpha = (L+1)/\mathsf{dnum}$.
Q'	Modulus left after bootstrapping.
W	Word size (bits).

2 BACKGROUND

We introduce the basic formulation of CKKS. Key notations and parameters are tabulated in Table 1.

2.1 CKKS Basics

In CKKS, a complex message vector $\mathbf{z} \in \mathbb{C}^{N/2}$ is first encoded into a plaintext polynomial $m \in \mathbb{Z}[X]/\langle X^N + 1\rangle$ for a power-of-two N. To be precise, a variant of inverse discrete Fourier transform is applied to $\mathbf{z}$ and result in a real-coefficient polynomial. Then, this polynomial is multiplied with a scaling factor Δ (typically in the range $[2^{30}, 2^{60}]$) and rounded to the closest integer-coefficient polynomial m, denoted by $m \leftarrow \mathsf{Ecd}(\mathbf{z}; \Delta)$. The decoding process is exactly the reverse, which is to apply a variant of the discrete Fourier Transform and divide by the scaling factor $1/\Delta$, which is denoted by $\mathbf{z}' \leftarrow \mathsf{Dcd}(m; \Delta)$. Note that $\mathsf{Dcd}(\mathsf{Ecd}(\mathbf{z}; \Delta); \Delta) \simeq \mathbf{z}$.

The multiplication of encoded polynomials approximately preserves the Hadamard multiplication of input message vectors, while the corresponding scaling factor is increased into a quadratic scale: For $m \leftarrow \mathsf{Ecd}(\mathbf{z}; \Delta)$ and $m' \leftarrow \mathsf{Ecd}(\mathbf{z}'; \Delta)$, the multiplication $m \cdot m'$ will approximately correspond to $\mathsf{Ecd}(\mathbf{z} \odot \mathbf{z}'; \Delta^2)$, where $\odot$ represents Hadamard product between the two vectors.

After the encoding process, a plaintext polynomial can be securely encrypted into a ciphertext ct under the ring learning with errors (RLWE) assumption as follows: For a large modulus $Q \gg \Delta$, we denote the residue ring $\mathcal{R}/Q\mathcal{R}$ as $\mathcal{R}_Q$. For a (small) secret polynomial $s \in \mathcal{R}$, the public key is generated as $\mathsf{pk} := (b = -a \cdot s + e, a) \in \mathcal{R}_Q^2$, where a is uniformly sampled from $\mathcal{R}_Q$, and e is sampled from a Gaussian distribution over $\mathcal{R}$. Then, with randomly chosen small polynomials $r, e_0, e_1 \in \mathcal{R}$, the message vector $\mathbf{z}$ is encoded into a plaintext polynomial and then encrypted into $\mathsf{ct} = (c_0, c_1) \in \mathcal{R}_Q^2$:

$$\mathsf{Enc}_{\mathsf{pk}}(\mathbf{z}; \Delta) : \mathbf{z} \mapsto r \cdot (b, a) + (\mathsf{Ecd}(\mathbf{z}; \Delta) + e_0, e_1) \pmod{Q}.$$

The decryption of $\mathsf{ct} = (c_0, c_1) \pmod{Q}$ is simply done by computing $m \leftarrow c_0 + c_1 \cdot s \pmod{Q}$ through the secret key s. Note that the CKKS decryption is not exact, but *approximate*. For example, when $\mathsf{ct} = (c_0, c_1)$ is the encryption of the message $\mathbf{z}$, it holds that $c_0 + c_1 \cdot s = m + e' \pmod{Q}$ for $m \leftarrow \mathsf{Ecd}(\mathbf{z}; \Delta)$ and $e' := e + e_0 + e_1 \cdot s$, and the final output of the decryption would be $\mathsf{Dcd}(m + e'; \Delta) \simeq \mathsf{Dcd}(m; \Delta) \simeq \mathbf{z}$.

Various operations can be evaluated on ciphertexts, but we only explain addition and multiplication here. Please refer to [12, 13] for more details of applicable operations. Let $\mathsf{ct} := (c_0, c_1)$ and $\mathsf{ct}' := (c_0', c_1')$ be the ciphertexts of the message $\mathbf{z}$ and $\mathbf{z}'$, respectively. Then, the homomorphic addition and multiplication are described as follows:

Addition: $\mathsf{ct}_{\mathsf{add}} \leftarrow \mathsf{ct} + \mathsf{ct}'$ is a valid ciphertext of $\mathbf{z} + \mathbf{z}'$. Namely, $\mathsf{ct}_{\mathsf{add}} = (c_0, c_1)$ satisfies

$$c_0 + c_1 \cdot s \simeq \mathsf{Ecd}(\mathbf{z} + \mathbf{z}'; \Delta).$$

Multiplication: To multiply ct with ct′, the following computation $(c_{\times,0}, c_{\times,1}, c_{\times,2}) = (c_0 \cdot c_0', c_0 \cdot c_1' + c_1 \cdot c_0', c_1 \cdot c_1')$ is done first. This procedure is referred to as tensoring. From tensoring, the ciphertext degree is increased from 2 to 3, and this new degree-3 ciphertext $(c_{\times,0}, c_{\times,1}, c_{\times,2})$ satisfies

$$c_{\times,0} + c_{\times,1} \cdot s + c_{\times,2} \cdot s^2 \simeq \mathsf{Ecd}(\mathbf{z}; \Delta) \cdot \mathsf{Ecd}(\mathbf{z}'; \Delta) \simeq \mathsf{Ecd}(\mathbf{z} \odot \mathbf{z}'; \Delta^2).$$

Refer to Section 3.2 for a formal error analysis.

To prevent the exponential growth of the ciphertext degree, a procedure called relinearization is performed. It requires the use of a special public key called key-switching key (swk), which is an encryption of $P \cdot s^2$ under a larger modulus PQ for $P \simeq Q$. Then, by computing

$$\mathsf{ct}_{\mathsf{mult}} \leftarrow (c_{\times,0}, c_{\times,1}) + \left\lfloor 1/P \cdot c_{\times,2} \cdot \mathsf{swk} \right\rceil,$$

we can obtain a valid ciphertext $\mathsf{ct}_{\mathsf{mult}}$ whose decryption is approximately $\mathbf{z} \odot \mathbf{z}'$. The size of the special modulus P can be dynamically controlled by the use of hybrid key-switching technique [16].

Rescaling: The resulting ciphertext $\mathsf{ct}_{\mathsf{mult}} = (c_0, c_1)$ of homomorphic multiplication is actually associated with the squared scaling factor Δ^2, which means

$$c_0 + c_1 \cdot s \simeq \mathsf{Ecd}(\mathbf{z}_{\mathsf{mult}}; \Delta^2),$$

and the squared scaling factor Δ^2 must be reduced back to Δ to continue further operations with other ciphertexts. We simply compute $(\lfloor c_0/\Delta \rceil, \lfloor c_1/\Delta \rceil)$ to do so, and the procedure is called rescaling. The modulus is reduced from Q to Q/Δ by rescaling, such that $\lfloor c_0/\Delta \rceil + \lfloor c_1/\Delta \rceil \cdot s \simeq \mathsf{Ecd}(\mathbf{z}_{\mathsf{mult}}; \Delta) \pmod{Q/\Delta}$.

2.2 Full-RNS Instantiation of CKKS: RNS-CKKS

To efficiently handle the large modulus Q, RNS-CKKS [12] utilizes the residue number system (RNS) to split a large coefficient a of a polynomial into a tuple of multiple small residues modulo q_i primes $(a^{(0)}, a^{(1)}, \ldots, a^{(L)})$, where $Q = \prod_{i=0}^{L} q_i$, $a^{(i)} = a \pmod{q_i}$, and $a^{(i)} \in \{0, 1, \ldots, q_i - 1\}$. As it is difficult to evaluate the division by Δ (and rounding) in this form, RNS-CKKS adopts an alternative way of performing rescaling, where it chooses q_i $(i = 1, 2, \ldots, L)$ to be close to Δ and instead compute $(\lfloor c_0/q_i \rceil, \lfloor c_1/q_i \rceil)$. Then, we can perform a total of L rescaling operations on a ciphertext, which we refer to as the maximum level. When $\ell + 1$ number of primes are utilized in the modulus $Q_\ell = \prod_{i=0}^{\ell} q_i$, the (current) level of the ciphertext is ℓ. The ciphertext level is reduced by 1 through the rescaling procedure $(c_0, c_1) \pmod{Q_\ell} \mapsto (\lfloor c_0/q_\ell \rceil, \lfloor c_1/q_\ell \rceil) \pmod{Q_{\ell-1}}$.

Contrary to the basic CKKS scheme, RNS-CKKS has a precision issue in homomorphic addition, since the input ciphertexts might have different scaling factors because of the difference between Δ and q_i's. In [12], they suggested to set each q_i to be very close to Δ so that the gaps between different scaling factors become very small. However, this method results in non-negligible precision loss from homomorphic addition if we are not able to find enough number of such q_i's with the closeness property. Recently, a new technique resolving this precision issue called exact scaling [22] has been proposed, which does not require such closeness property for each prime. The exact scaling method defines the scaling factor Δ_ℓ per level, which is defined as $\Delta_{\ell-1} := \Delta_\ell^2/q_\ell$ from $\ell = L$ to $\ell = 1$. The input ciphertexts of homomorphic operations are manipulated to have the same level and the same scaling factor before performing the homomorphic operations, and hence we are able to fundamentally avoid the precision issue of homomorphic addition. This manipulation process before the homomorphic operations is done by the algorithm called Adjust.

Let ct and ct′ be ciphertexts with level ℓ and ℓ' $(\ell > \ell')$, and scaling factors Δ_ℓ and $\Delta_{\ell'}$, respectively. To perform homomorphic addition or multiplication over ct and ct′, we adjust the level and the scaling factor of ct through $\mathsf{Adjust}(\mathsf{ct}, \ell')$ which works as follows: First, drop the level of the ciphertext ct from ℓ to $\ell' + 1$. Then, multiply ct by a constant $\left\lceil \frac{\Delta_{\ell'} \cdot q_{\ell'+1}}{\Delta_\ell} \right\rfloor = \frac{\Delta_{\ell'} \cdot q_{\ell'+1}}{\Delta_\ell} + \delta$ with $\delta \in [-1/2, 1/2]$. Finally, rescale the result by $q_{\ell'+1}$, and the final result belongs to level ℓ' with the appropriate scaling factor $\Delta_{\ell'}$. As a result, ct and ct' eventually have the same level and the same scaling factor.

RNS-CKKS also provides a method to efficiently change the modulus (e.g., between Q and P during key switching) referred to as fast base conversion (FBC). For example, the modulus can be converted from $Q_\ell = \prod_{i=0}^{\ell} q_i$ to $P = \prod_{i=0}^{K-1} p_i$ using the following equation for every p_i $(i = 0, 1, \ldots, K-1)$:

$$\left[\sum_{j=0}^{\ell} \left(\left[a^{(j)} \cdot \left(\frac{Q_\ell}{q_j} \right)^{-1} \right]_{q_j} \cdot \left[\frac{Q_\ell}{q_j} \right]_{p_i} \right) \right]_{p_i}$$

Here, $[\cdot]_q$ denotes a modular reduction into the range $[0, q-1]$. $\left[\left(\frac{Q_\ell}{q_j} \right)^{-1} \right]_{q_j}$ and $\left[\frac{Q_\ell}{q_j} \right]_{p_i}$ are precomputed constant values. Each

$\left[a^{(j)} \cdot \left(\frac{Q_\ell}{q_j}\right)^{-1}\right]_{q_j}$ value can be computed only once and reused for every p_i. Also, the final modular reduction by p_i can be done after the summation, which is called lazy modular reduction in FBC [3].

Another important performance optimization is using number-theoretic transform (NTT), a finite-field variant of Fourier transform, for polynomial multiplications modulo q. As a multiplication between two polynomials in $\mathcal{R}_q$ is equivalent to a convolution between their coefficients, NTT can reduce the computational complexity from $O(N^2)$ to $O(N \log N)$ by converting the convolution into a simple Hadamard product; $NTT(m) \odot NTT(m') = NTT(m \cdot m')$ holds and the complexity of NTT is $O(N \log N)$ when using fast Fourier transform algorithms. Computationally, FBC and NTT are the two most dominant operations in the computation of RNS-CKKS, and other element-wise modular multiplication and addition operations account for the rest.

2.3 Bootstrapping

After repeated rescaling, the modulus eventually becomes too small to perform more rescaling operations. To continue operations when the modulus is fully dissipated, bootstrapping is required to restore the modulus. CKKS bootstrapping is performed in four steps of `ModRaise`, `CoeffToSlot`, `EvalMod`, and `SlotToCoeff`. `ModRaise` increases the current modulus of ciphertext to Q by increasing the number of primes utilized from one (i.e., q_0) to the maximum possible. In this process, multiples of q_0 are added to the coefficients of the polynomials composing the ciphertext. The following bootstrapping steps remove these multiples by moving coefficients to slot positions (`CoeffToSlot`), homomorphically evaluating the mod q_0 function with an approximate polynomial (`EvalMod`), and moving the coefficients back to their original positions (`SlotToCoeff`). These steps include dozens of rescaling operations, and dissipate a large portion of the modulus, leaving only Q' ($< Q$) as the modulus for further operations.

Enhancing bootstrapping performance is crucial to the practical use of FHE as bootstrapping is the most dominant procedure accounting for most of the computation time in CKKS-based FHE applications [14, 21]. In addition, contrary to the other FHE schemes, the CKKS bootstrapping results in an additional error attached to the message, so the output precision is another critical factor of bootstrapping performance. Abundant studies have proposed methods for enhancing the performance and precision of bootstrapping [8–10, 18], which we selectively adopt in each of our implementations of FHE CKKS applications.

3 CKKS OVER SMALL WORD-SIZE BASED ON COMPOSITE SCALING

3.1 Composite Scaling

As described in Section 2.2, rescaling in RNS-CKKS is done as $ct \pmod{Q_\ell} \mapsto \lfloor 1/q_\ell \cdot ct \rceil \pmod{Q_{\ell-1}}$, where each prime q_i is chosen to be approximately $\log \Delta$-bit. We refer to this approach as *single scaling*. However, this *single scaling* approach is only possible when $\log \Delta$ *is smaller than the word size* of the given architecture, unless we use multi-precision arithmetic which accompanies large computational overhead [35]. In other words, if $\log \Delta$ is larger than the word size, we need to take another approach for rescaling, which we call *composite scaling*.

In composite scaling, each modulus q_ℓ is not a prime anymore, but a product of multiple primes $q_{\ell,j}$'s for $0 \le j < t$, i.e., $q_\ell := \prod_{j=0}^{t-1} q_{\ell,j}$. The number of primes t in each q_ℓ is determined by the gap between $\log \Delta$ and the word-size, i.e., $t := \lceil \frac{\log \Delta}{\text{word-size}} \rceil$. For the functionality of rescaling, each q_ℓ needs to be set as $\log \Delta$-bit, and hence we set each prime $q_{\ell,j}$ to be approximately $\frac{\log \Delta}{t}$-bit.

Mathematically, all the homomorphic operations in CKKS based on the composite scaling approach are expressed in exactly the same way as those in Section 2.1. However, since each modulus q_i is now a composition of multiple primes, there exist algorithmic differences between composite and single scaling. For example, rescaling in single scaling is to divide by a single prime and take rounding. However, in composite scaling, one needs to divide by *multiple primes* instead as follows:

$$\textit{Comp-RS}(\text{ct}) : \text{ct} \pmod{Q_\ell} \to \left\lfloor \frac{1}{\prod_{j=0}^{t} q_{\ell,j}} \cdot \text{ct} \right\rceil \pmod{Q_{\ell-1}}.$$

In the following subsections, we provide the precision analysis for each homomorphic operation based on composite scaling and show that there is basically no difference in terms of precision loss compared to the previous approach.

3.2 Precision Analysis for Mult and Rescale

We first analyze the precision of homomorphic operations between two ciphertexts with the same level. The CKKS algorithms that do not contain the division procedure, including encryption, decryption, addition, and tensoring, result in the same error growth for single and composite scaling methods. However, the other operations such as rescale and (hybrid) key-switching include the division-by-q_ℓ computation, which might result in different error growth between single and composite scaling.

Since each modulus q_ℓ is a product of multiple primes in composite scaling, the division-by-q_ℓ computation can be done by applying either a single FBC from a big base $q_\ell = \prod_{j=0}^{t-1} q_{\ell,j}$ or multiple FBCs from a small base $q_{\ell,j}$ for $0 \le j < t$ successively. In this paper, we apply the latter strategy, which results in almost the same rescaling error for the single scaling approach. With this strategy, Theorem 3.1 and 3.2 below show the precision loss from rescaling and key switching in composite scaling.

Theorem 3.1 (Composite Rescale). *For a level-ℓ ciphertext* $\text{ct} = (c_0, c_1)$*, the algorithm* Comp-RS(ct) *returns a ciphertext* ct′ *of level* $\ell - 1$*, along with additional error* e_{rs} *satisfying*

$$||e_{\text{comp-rs}}||_\infty^{\text{can}} \le \left(\frac{1}{q_{l,1} q_{l,2} \cdots q_{l,t-1}} + \cdots + \frac{1}{q_{\ell,t-1}} + 1\right) B_{rs},$$

where $B_{rs} := \sqrt{3N} + 8\sqrt{\frac{hN}{3}}$ *denotes the upper bound of the rescaling error in the single scaling approach.*

Proof. We follow the heuristic assumption about the canonical embedding norm in [12, 13, 22]. The given ciphertext $\text{ct} = (c_0, c_1) \in \mathcal{R}_{Q_\ell}^2$ encrypting m with error e satisfies the following relation:

$$c_0 + c_1 \cdot s \equiv \Delta_\ell \cdot m + e \ (mod\ Q_\ell).$$

To perform ModDown about one level q_ℓ, we first perform a rescaling procedure with one RNS-prime $q_{\ell,0}$. Then it follows that

$$\left\lceil \frac{c_0}{q_{\ell,0}} \right\rfloor + \left\lceil \frac{c_1}{q_{\ell,0}} \right\rfloor \cdot s \equiv \frac{1}{q_{\ell,0}}(\Delta_\ell \cdot m + e) + (r_0 + r_1 \cdot s) \ \left(mod\ \frac{Q_\ell}{q_{\ell,0}}\right),$$

where r_0, r_1 denote the rounding parts by one RNS prime $q_{\ell,0}$ with coefficients bounded by $1/2$. From lemma 2 of [13], it follows that $e_{rs}^{(0)} := r_0 + r_1 \cdot s$ is bounded by $\|e_{rs}^{(0)}\|_\infty^{\mathsf{can}} \le B_{rs} = \sqrt{3N} + 8\sqrt{\frac{hN}{3}}$. We rewrite the modular equation as follows:

$$c_0^{(0)} + c_1^{(0)} \cdot s \equiv \frac{1}{q_{\ell,0}}(\Delta_\ell \cdot m + e) + e_{rs}^{(0)} \ \left(mod\ \frac{Q_\ell}{q_{\ell,0}}\right).$$

We then perform rescaling with next RNS-prime $q_{\ell,1}$ to obtain

$$c_0^{(1)} + c_1^{(1)} \cdot s \equiv \frac{1}{q_{\ell,0}q_{\ell,1}}(\Delta_\ell \cdot m + e) + \left(\frac{e_{rs}^{(0)}}{q_{\ell,1}} + e_{rs}^{(1)}\right) \ \left(mod\ \frac{Q_\ell}{q_{\ell,0}q_{\ell,1}}\right),$$

where $e_{rs}^{(1)}$ represents the new error added during rescaling with $q_{\ell,1}$. We repeat this procedure t times for each RNS-prime $q_{\ell,j}$ composing one level q_ℓ, resulting in the following:

$$c_0^{(t-1)} + c_1^{(t-1)} \cdot s \equiv \frac{1}{q_\ell}(\Delta_\ell \cdot m + e) + e_{\text{comp-rs}} \ \ (mod\ Q_{\ell-1}),$$

where the final rescaling error $e_{\text{comp-rs}}$ is of the form:

$$e_{\text{comp-rs}} = \frac{e_{rs}^{(0)}}{q_{l,1}q_{l,2}\cdots q_{l,t-1}} + \cdots + \frac{e_{rs}^{(t-2)}}{q_{\ell,t-1}} + e_{rs}^{(t-1)}.$$

We note that all error terms $e_{rs}^{(j)}$'s have the same upper bound B_{rs}. Therefore, we obtain the following inequality for error $e_{\text{comp-rs}}$ as follows:

$$\begin{aligned}\|e_{\text{comp-rs}}\|_\infty^{\mathsf{can}} &\le \frac{\|e_{rs}^{(0)}\|_\infty^{\mathsf{can}}}{q_{l,1}q_{l,2}\cdots q_{l,t-1}} + \cdots + \frac{\|e_{rs}^{(t-2)}\|_\infty^{\mathsf{can}}}{q_{\ell,t-1}} + \|e_{rs}^{(t-1)}\|_\infty^{\mathsf{can}} \\ &\le \left(\frac{1}{q_{l,1}q_{l,2}\cdots q_{l,t-1}} + \cdots + \frac{1}{q_{\ell,t-1}} + 1\right) B_{rs}.\end{aligned}$$

□

Remark. The approach to reduce one level q_ℓ at once requires to perform ModDown (i.e., a big FBC) for t RNS primes $q_{\ell,0}, q_{\ell,1}, ..., q_{\ell,t-1}$. This operation induces the error whose upper bound is $\sqrt{t}$ times larger compared to the error bound B_{rs}. We refer to the proof of Lemma 1 in [12] for more details.

A similar argument can be applied to the key-switching procedure as well.

Theorem 3.2 (Composite KeySwitching). *Let* $\mathsf{ct} = (c_0, c_1)$ *be a level-*ℓ *ciphertext under the secret* s'*. The composite key-switching algorithm* $\mathsf{Comp\text{-}KS}_{s'\to s}(\mathsf{ct})$ *outputs a ciphertext* $\mathsf{ct}_{\text{comp-ks}}$ *under the secret* s *with additional error* $e_{\text{comp-ks}}$ *satisfying*

$$\begin{aligned}||e_{\text{comp-ks}}||_\infty^{\mathsf{can}} \le{}& \frac{8\beta \cdot \sqrt{\alpha} \cdot \max_i(\tilde{Q}_i) \cdot \sigma N}{\sqrt{3}P} \\ &+ \left(\frac{1}{p_1p_2\cdots p_{K-1}} + \cdots + \frac{1}{p_{K-1}} + 1\right) B_{rs},\end{aligned}$$

where $B_{rs} := \sqrt{3N} + 8\sqrt{\frac{hN}{3}}$ *is the upper bound of the key-switching error in the single scaling approach, and* $\beta := \lceil (\ell+1)/\alpha \rceil$.

Proof. We follow the key switching procedure as described in [18]. We first perform RNS-Decompose and ModUp on polynomial $c_1 \in \mathcal{R}_{Q_\ell}$ to obtain $\tilde{c}_1^{(i)} \in \mathcal{R}_{PQ_L}$ for $0 \le i < \beta$, satisfying

$$\tilde{c}_1^{(i)} \equiv c_1 \cdot \left(\frac{Q_\ell}{\tilde{Q}_i}\right)^{-1} \ (mod\ \tilde{Q}_i) \quad and \quad \|\tilde{c}_1^{(i)}\|_\infty \le \frac{\alpha}{2} \cdot \tilde{Q}_i$$

where $Q_\ell = \tilde{Q}_0 \cdot \tilde{Q}_1 \cdots \tilde{Q}_{\beta-1}$. We note that each polynomial $\tilde{c}_1^{(i)}$ behaves like the sum of α independent and uniform random variables over $\mathcal{R}_{\tilde{Q}_i}$, so its variance is $V = \alpha \cdot \tilde{Q}_i^2 N/12$. For a given key-switching key $\mathsf{swk} = (\mathsf{swk}_0, \mathsf{swk}_1) = \left(\{b_i'\}_{i=0}^{\mathsf{dnum}-1}, \{a_i'\}_{i=0}^{\mathsf{dnum}-1}\right) \in \mathcal{R}_{PQ_L}^{2\times \mathsf{dnum}}$, we denote the error terms in swk_1 as $\{e_i'\}$. Then the error after the Inner Product is bounded by

$$\begin{aligned}\|\sum_{i=0}^{\beta-1} \tilde{c}_1^{(i)} \cdot e_i'\|_\infty^{\mathsf{can}} &\le \sum_{i=0}^{\beta-1} \|\tilde{c}_1^{(i)} \cdot e_i'\|_\infty^{\mathsf{can}} \\ &\le \sum_{i=0}^{\beta-1} 16 \cdot \sqrt{\frac{\alpha \cdot \tilde{Q}_i^2 N}{12}} \cdot \sqrt{\sigma^2 N} \\ &\le \frac{8\beta \cdot \sqrt{\alpha} \cdot \max_i(\tilde{Q}_i) \cdot \sigma N}{\sqrt{3}} := B_{\mathsf{ks}},\end{aligned}$$

adopting the heuristic bound for canonical embedding norm [12, 13, 22]. Finally, we perform ModDown to reduce K RNS primes in modulus $P = \prod_{i=0}^{K-1} p_i$. We note that we can adopt the same technique in Theorem 3.1 by applying rescaling K times for each RNS primes p_i. Hence, we obtain the following error bound for $e_{\text{comp-ks}}$ in ciphertext $\mathsf{ct}_{\text{comp-ks}}$.

$$\|e_{\text{comp-ks}}\|_\infty^{\mathsf{can}} \le \frac{1}{P} \cdot B_{\mathsf{ks}} + \left(\frac{1}{p_1p_2\cdots p_{K-1}} + \cdots + \frac{1}{p_{K-1}} + 1\right) B_{rs},$$

where $B_{\mathsf{rs}} = \sqrt{3N} + 8\sqrt{\frac{hN}{3}}$ as in Theorem 3.1. □

To put this all together, we can analyze the precision for the multiplication (with rescaling) for composite scaling. The multiplication consists of three steps: (1) tensoring, (2) key-switching, and (3) rescaling. For tensoring, the error terms are the same as the single scaling approach. For the last two steps, we can use Therom 3.2 and 3.1 respectively to analyze the error. Also, note that the tensoring and key-switching errors are significantly reduced during the rescaling procedure; hence, the rescaling error becomes the dominant term for the entire multiplication error.

Let $\mathsf{ct}_1, \mathsf{ct}_2$ be two ciphertexts at the same level encrypting m_1, m_2 respectively. The multiplication for ciphertexts $\mathsf{ct}_1, \mathsf{ct}_2$ at the same level ℓ is done in three steps: tensoring, linearization, and rescaling. For tensoring, we tensor the ciphertexts as follows.

$$\begin{aligned}&c_{\times,0} + c_{\times,1} \cdot s + c_{\times,2} \cdot s^2 \\ &\equiv (\Delta_\ell \cdot m_1 + e_1) \cdot (\Delta_\ell \cdot m_2 + e_2) \\ &= \Delta_\ell^2 \cdot m_1 m_2 + \underbrace{\Delta_\ell \cdot (m_1 e_2 + m_2 e_1) + e_1 e_2}_{\text{tensoring error } e_\times} \ (mod\ Q_\ell).\end{aligned}$$

After tensoring, to remove the s^2 term, we relinearize the tensored ciphertext using key-switching as follows.

$$(c_{\text{relin},0}, c_{\text{relin},1}) \leftarrow (c_{\times,0}, c_{\times,1}) + \mathsf{Comp\text{-}KS}_{s^2\to s}(0, c_{\times,2}) \ (mod\ Q_\ell).$$

Then, we yield the ciphertext encrypting the product of the inputs, but with an inappropriate scaling factor. That means,

$$c_{\text{relin},0} + c_{\text{relin},1} \cdot s \equiv \Delta_\ell^2 \cdot m_1 m_2 + e_\times + e_{\text{comp-ks}} \ (mod\ Q_\ell).$$

Finally, we rescale the relinearized ciphertexts to have an appropriate scaling factor. To be more precise,

$$\begin{aligned} c_{\text{out},0} + c_{\text{out},1} \cdot s &\equiv \frac{\Delta_\ell^2}{q_\ell} \cdot m_1 m_2 + \frac{e_\times + e_{\text{comp-ks}}}{q_\ell} + e_{\text{comp-rs}} \\ &= \Delta_{\ell-1} \cdot m_1 m_2 + e_{\text{comp-mult}} \ (mod\ Q_{\ell-1}), \end{aligned}$$

where $\Delta_{\ell-1} = \frac{\Delta_\ell^2}{q_\ell}$ is the scaling factor at $(\ell-1)$-level [22]. We can estimate the error induced by composite multiplication as follows.

Corollary 3.3 (Composite Multiplication). *The error term $e_{\text{comp-mult}}$ induced from Mult (Tensoring + Relin. + Rescaling) is bounded as*

$$\|e_{\text{comp-mult}}\|_\infty^{\text{can}} \le \frac{\|e_\times\|_\infty^{\text{can}}}{q_\ell} + \frac{\|e_{\text{comp-ks}}\|_\infty^{\text{can}}}{q_\ell} + \|e_{\text{comp-rs}}\|_\infty^{\text{can}},$$

where $e_{\text{comp-rs}}$ and $e_{\text{comp-ks}}$ are described in Theorem 3.1 and 3.2.

Note that the Mult error from the single scaling approach has the same form of the upper bound $\|e_{\text{mult}}\|_\infty^{\text{can}} \le \frac{\|e_\times\|_\infty^{\text{can}}}{q_\ell} + \frac{\|e_{\text{ks}}\|_\infty^{\text{can}}}{q_\ell} + \|e_{\text{rs}}\|_\infty^{\text{can}}$, where e_{ks} and e_{rs} denote the key-switching and rescaling error in single scaling, respectively. From Theorem 3.1 and 3.2, we get $\|e_{\text{comp-rs}}\|_\infty^{\text{can}} \simeq \|e_{\text{rs}}\|_\infty^{\text{can}} + \frac{1}{q_{\ell,t-1}} \cdot \|e_{\text{rs}}\|_\infty^{\text{can}}$ and $\|e_{\text{comp-ks}}\|_\infty^{\text{can}} \simeq \|e_{\text{ks}}\|_\infty^{\text{can}} + \frac{1}{p_{K-1}}\|e_{\text{rs}}\|_\infty^{\text{can}}$, where both additional terms $\frac{1}{q_{\ell,t-1}} \cdot \|e_{\text{rs}}\|_\infty^{\text{can}}$ and $\frac{1}{p_{K-1}}\|e_{\text{rs}}\|_\infty^{\text{can}}$ are negligible compared to the dominant terms. As a result, single and composite scaling methods result in almost the same precision loss in homomorphic operations between the ciphertexts with the same level.

3.3 Precision Analysis for Add and Crossing levels

When we manipulate ciphertexts at different levels, we need to make their level the same before the subsequent homomorphic operation. If we lower the level, ciphertexts would have the same level but different scaling factors, resulting in an extra error after homomorphic addition due to the difference of the scaling factors.

To address this issue, we use the exact scaling approach as in Section 2.2, which was introduced in [18] and analyzed with detailed experiments in [22]. We utilize the Adjust function in [22] to adjust the scaling factors of ciphertexts that are at different levels. At a high level, to adjust the ciphertext at level ℓ to lower level ℓ', we first multiply the ratio between the scaling factors (i.e., $\lfloor q_{\ell'+1}\Delta_{\ell'}/\Delta_\ell \rceil$) and lower the level to adjust the scaling factors (see Section 2.2 for the details). Since Adjust also crosses multiple moduli, we need to analyze the precision for composite scaling.

To adopt the Adjust technique to the composite scaling approach, we should replace the last rescaling procedure with the composite rescaling procedure. Except for the last rescaling procedure, all other processes can be done the same as the single scaling approach. The last rescaling error is the dominant term for the error from Adjust , as in the multiplication and rescaling procedure in Section 3.2. By applying the similar argument in Section 3.2, Theorem 3.4 represents the Adjust error of the composite scaling approach.

Theorem 3.4 (Composite Adjust). *Let* ct *be an encryption of m having an error e, along with level $\ell > \ell'$ and scaling factor Δ_ℓ. The algorithm* Comp-Adj(ct, ℓ') *returns a ℓ'-level ciphertext* $\text{ct}_{comp\text{-}adj}$ *having error $e_{comp\text{-}adj}$ denoted by*

$$e_{comp\text{-}adj} = \frac{\Delta_{\ell'}}{\Delta_\ell} \cdot e + \frac{\delta(\Delta_\ell \cdot m + e)}{q_{\ell'+1}} + e_{comp\text{-}rs},$$

where $\delta = \left\lceil \frac{\Delta_{\ell'} \cdot q_{\ell'+1}}{\Delta_\ell} \right\rfloor - \frac{\Delta_{\ell'} \cdot q_{\ell'+1}}{\Delta_\ell} \in [-1/2, 1/2]$ and $e_{comp\text{-}rs}$ denotes the error during Comp-RS to reduce one level $q_{\ell'}$.

Proof. The algorithm Comp-Adj(ct, ℓ') replaces the original rescaling in Adjust [22] with our Comp-RS and the rest of the process is the same. We refer to [22, pg.18] for detailed proof. □

In conclusion, we can use composite scaling with almost the same precision loss to single scaling. Note that we can further improve the speed for rescaling, adjusting, and key-switching from the hybrid use of FBC from a big base and a small base. For example, we can compute the division-by-q_ℓ with the following method instead: (a) Apply a single FBC from the big base $q_{\ell,0} \cdot q_{\ell,1} \cdots q_{\ell,t-2}$, and then (b) apply a single FBC from the small base $q_{\ell,t-1}$. The rescaling error would be slightly increased but not significantly.

3.4 Moduli Generation for Composite Scaling

In the exact scaling approach, the closeness of q_ℓ's does not affect the precision of homomorphic addition anymore, but each prime $q_{\ell,i}$ (for $1 \le \ell \le L$, $0 \le j < t$) still needs to be carefully chosen to prevent the divergence of the scaling factor Δ_ℓ as ℓ increases [22]. We follow a similar methodology to the moduli generation algorithms given in [22], which adaptively choose $q_{\ell,0}, q_{\ell,1}, ..., q_{\ell,t-1}$ from $\ell = L$ to 1 based on the value of the previous scaling factor $\Delta_{\ell+1}$. To be precise, we choose each $q_{\ell,j}$ to be close to $(\Delta_{\ell+1}^2/\Delta_L)^{1/t}$ so that $\Delta_\ell = \Delta_{\ell+1}^2/\prod_{j=0}^{t-1} q_{\ell,j} \simeq \Delta_L$. Since the level-0 modulus q_0 is irrelevant to the divergence issue of the scaling factors, the primes $q_{0,j}$ for $0 \le j < t$ which determines q_0 is determined independently.

The baseline algorithm of RNS-CKKS moduli generation for composite scaling is given as Algorithm 1. The max level L, the target bit length p of the scaling factor, and the target bit length $p_0 (> p)$ of the level-0 modulus q_0 are given as input.

Algorithm 1 RNS-CKKS Moduli Generation for Composite Scaling; `ClosestPrimes`$(x, 2N, t)$ outputs t-closest primes of x which congruent to 1 modulo $2N$.

Require: Max level L, target scaling factor Δ, and target $\log q_0$ size p_0
Ensure: $\Delta_\ell = \Delta_{\ell+1}^2/q_{\ell+1} \simeq \Delta_L$ for $0 \le \ell < L$

```
(q_{L,0}, q_{L,1}, ..., q_{L,t-1}) ← ClosestPrimes(Δ^{1/t}, 2N, t)
for ℓ = L − 1 to 1 do
    Δ_{ℓ+1} ← ∏_{j=0}^{t−1} q_{ℓ+1,j}
    Δ_target ← Δ_{ℓ+1}^2/Δ_L
    (q_{ℓ,0}, q_{ℓ,1}, ..., q_{ℓ,t−1}) ← ClosestPrimes(Δ_target^{1/t}, 2N, t)
end for
(q_{0,0}, q_{0,1}, ..., q_{0,t−1}) ← ClosestPrimes(2^{p_0/t}, 2N, t)
return (q_{ℓ,j})_{0≤ℓ≤L, 0≤j<t}
```

Table 2: Empirical bootstrapping precision bits ($\log \bar{\epsilon}^{-1}$) of 64-bit (baseline) and 32-bit (composite scaling) parameter sets implemented on Lattigo [28]. $\bar{\epsilon}$ denotes the mean magnitude of bootstrapping error averaged over 100 bootstrapping trials. h and $\tilde{h}$ are the secret key sparsity factors (Hamming weights) defined in [9].

Param	N	Δ	$\log QP$	$\log Q'$	h	$\tilde{h}$	# of primes	$\log \bar{\epsilon}^{-1}$
64-bit	2^{16}	2^{48}	1519	487	192	32	$Q:24, P:5$	22.771
32-bit	2^{16}	2^{48}	1519	487	192	32	$Q:48, P:10$	22.770

Table 3: Empirical bootstrapping precision bits ($\log \bar{\epsilon}^{-1}$) of 64-bit (baseline) and 32-bit (composite scaling) parameter sets implemented on OpenFHE [2]. LevelBudget denotes the number of FFT layers used in `CoeffToSlot` and `SlotToCoeff`. Secret key is sampled from uniform ternary distribution in both cases.

Param	N	Δ	$\log PQ$	levelBudget	# of primes	$\log \bar{\epsilon}^{-1}$
64-bit	2^{16}	2^{58}	1580	(2,2)	$Q:20, P:7$	12
32-bit	2^{16}	2^{58}	1580	(2,2)	$Q:40, P:14$	12

4 BOOTSTRAPPING ANALYSIS

As described in Section 2.3, bootstrapping is a crucial procedure that determines the performance and precision of FHE applications. Therefore, in this section, we analyze in detail how composite scaling affects the precision and execution cost of bootstrapping.

4.1 Bootstrapping Precision

First, we analyze how the bootstrapping precision bits ($\log \bar{\epsilon}^{-1}$) are affected when applying composite scaling. We developed proof-of-concept implementations of 32-bit composite scaling and the corresponding bootstrapping algorithm in both Lattigo [28] and OpenFHE [2]. The implemented 32-bit bootstrapping algorithm only differs from the baseline algorithm in that composite scaling is used in rescaling and that `ModRaise` starts with multiple primes left in the modulus q_0. Therefore, we are able to reuse the majority of the publicly available bootstrapping codes in Lattigo and OpenFHE. Table 2 and Table 3 show the baseline 64-bit parameter set along with the 32-bit parameter set we designed. Two parameter sets are equivalent in terms of applied algorithms and usability. They only differ in the number of primes: the 32-bit parameter set utilizes a doubled number of primes due to composite scaling, but both procedures consume the same amount of levels.

Despite using different scaling methods, the bootstrapping precision measured for the two parameter sets does not show a significant difference with merely 0.001 bits of precision difference (see $\log \bar{\epsilon}^{-1}$ in Table 2). Also, for 100 bootstrapping trials, the empirical worst-case precision bits, each of which is measured for the message slot with the maximum error among all slots and trials, were 20.66 bits for the 64-bit parameter set and 20.67 bits for the 32-bit set. As the precision difference is negligible for bootstrapping, which is the dominant source of error, we expect composite scaling to not require any additional measures to ensure the correctness of practical FHE applications compared to when using single scaling.

Table 4: Operation counts of the major operations and their weighted sum (WSum) for 64-bit (baseline) and 32-bit (composite scaling) parameter sets in Lattigo. Each operation's portion (ratio) in the weighted sum is also shown. For base conversion, it is assumed that lazy modular reduction (red) is performed after the accumulation of multiplication (mult) results.

Op (weight)	(I)NTT ($\frac{2W^2+W}{2}\log N$)	Base conversion mult (W^2)	Base conversion red (W^2+W)	ModMult ($2W^2+W$)	WSum
64-bit	10,685	39,420	9,083	47,513	1,297M
(ratio)	(54.4%)	(12.4%)	(2.9%)	(30.2%)	
32-bit	21,652	161,410	21,452	95,550	747M
(ratio)	(48.2%)	(22.1%)	(3.0%)	(26.6%)	

4.2 Execution Cost

To estimate the hardware execution cost of bootstrapping for each parameter set, we count the number of major operations that require multiplicative operations: number-theoretic transform and its inverse ((i)NTT), fast base conversion [5], and general modular multiplication (ModMult). We calculate the weighted sum of these operations considering the computational complexity of each operation; for word size of bit-length W, we set the weight of multiplication as W^2 and (Montgomery) modular reduction as $W^2 + W$ based on [7]. Then, the weight of each operation and the weighted sum of operations for the two parameter sets in Table 2 can be calculated as in Table 4.

As we utilize twice more primes for the 32-bit set, the number of (i)NTT and general modular multiplications increase linearly by 2.03× and 2.01×. Meanwhile, as the computational complexity of base conversion is quadratic to the number of primes, the number of base conversion multiplication increases by 4.09×. The number of modular reductions in base conversion does not increase quadratically (2.36×) due to the lazy modular reduction technique [3].

However, as the computational complexity of each operation is quadratic to the word size W, the 32-bit set substantially lowers the weighted sum (WSum), which represents the total computational cost of bootstrapping, by 42.4%. Also, we observe that base conversion accounts for a larger 25.1% (vs. 15.4%) portion of the total complexity for the 32-bit set, which is due to the quadratic complexity growth of base conversion in proportion to the number of primes.

Finally, we stress that the same amount of memory space is required for both parameter sets except for the precomputed constants required for base conversion, which occupy memory space of only few dozens of KB. The size of other major contributors to the working set, namely ciphertexts, plaintexts, and evaluation keys, does not change because (the number of primes)×(word size) stays the same for both parameters.

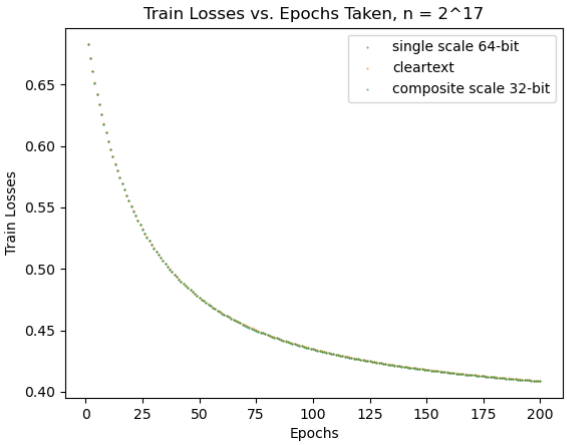


Figure 1: Comparison of training losses: 32-bit composite scale, 64-bit single scale, and cleartext computations

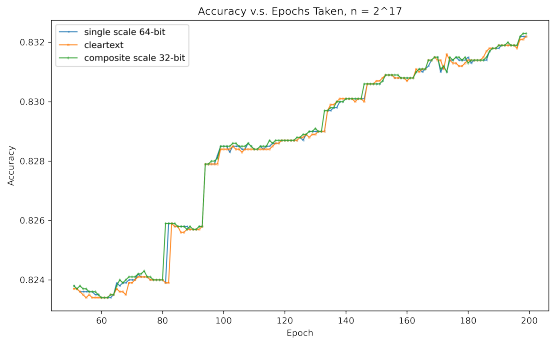


Figure 2: Comparison of accuracy: 32-bit composite scale, 64-bit single scale, and cleartext computations

4.3 Toward High-Precision Workloads

Most of real-world complex applications accompany the computation of large-depth circuits, and hence bootstrapping is required. The composite scaling method basically enables the use of large scaling factors even in small word-size architecture, but the precision of CKKS bootstrapping has an inherent upper bound regardless of the scaling factor size. Recently, there has been proposed a new method called META-BTS [4] that resolves this precision issue of CKKS bootstrapping. To be precise, META-BTS repeats the baseline CKKS bootstrapping multiple (r) times, which results in roughly r times higher precision. As a result, composite scaling with META-BTS finally enables to support high-precision workloads based on CKKS in arbitrary word-size architectures.

5 REAL-WORLD IMPLEMENTATION AND EVALUATION

5.1 Logistic Regression Model Training

In this section, we evaluate a logistic regression (LR) model trained using the composite scaling approach. During this training phase, the model receives encrypted training data as input and computes the weight vector of the LR model in an encrypted state. The prediction model can be expressed as follows:

$$\boldsymbol{y} = \text{Sigmoid}(\boldsymbol{X}\boldsymbol{w}) \tag{1}$$

where $\boldsymbol{w}$ denotes a vector of weights and Sigmoid is a function applied entry-wise defined as

$$\text{Sigmoid}(x) = \frac{1}{1 + \exp(-x)} \tag{2}$$

We leverage the code sourced from [19], which provides an example of LR model training using the OpenFHE library in CKKS with the 2014 US Infant Mortality dataset. This implementation uses a cross-entropy loss function and deploys the Nesterov Accelerated Gradient (NAG) as the optimization technique.

Table 5: Precision and average CPU execution time comparison of the logistic regression implementation based on OpenFHE. (∗) Refer to Section 5.4 on the CPU performance of composite scaling workloads.

Operation	Bit Precision Abs. Diff. (64-bit - 32-bit)	Time 64-bit	Time$^{(*)}$ 32-bit
MatVector	0.0002730	5.41s	12.16s
Sigmoid	0.0000599	9.02s	20.26s
Weights update	0.0002484	0.18s	0.46s
Epoch (loss)	0.00001585	14.83	34.01s

A comparison of LR models trained via OpenFHE CKKS composite scaling, OpenFHE CKKS single scaling, and cleartext is conducted across 200 epochs. Our model operates at a learning rate of 0.1 and uses 1024 training samples, each with 10 features. Both encrypted training processes set a ring dimension of 2^{17}, $\log_2(\text{scaling factor}) = 58$ and ensures a 128-bit security. Note that for encrypted training, a single bootstrapping (not META-BTS) is executed during each epoch excluding the first one.

Figure 1 provides a visual comparison of the loss values. The closely overlapped curves show very similar loss progression and convergence rates for the three training processes.

We assess and compare the accuracy of each epoch iteration in Figure 2. The data points are obtained by evaluating the prediction model against a test dataset of 46582 samples, based on the updated weights at the end of each epoch. The accuracy is defined by the percentage of correctly classified samples. Figure 2 shows the accuracy for each epoch in the encrypted training with composite scaling is closely aligned with that of single scaling and cleartext training, also converging at a similar rate. After 200 epochs, all three training processes obtain 83.2% accuracy. Overall, the experimental evidence in this section suggests that composite scaling exhibits functional capabilities that are sufficiently comparable to those of single scaling in the context of LR training.

Table 5 shows the mean absolute difference of the outputs of the Matrix-Vector multiplication, and the outputs of the Sigmoid collected over 200 epochs. We show also the absolute mean difference

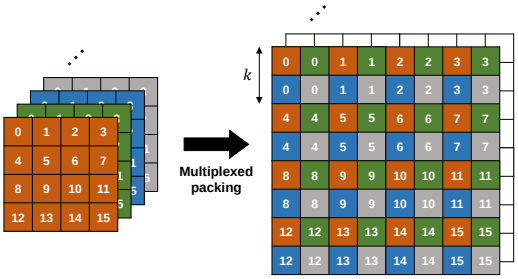


Figure 3: Multiplexed packing of a feature map with multiple channels (left side) into the slots (right side) for $k = 2$.

Table 6: The specification of our CNN model. Input and output feature map shapes are expressed as tuples of (channel, height, weight). All convolutional (conv) layers have strides of one. k_i and k_o are multiplexed packing parameters for input and output feature maps defined in [26]. After each conv layer, a precise polynomial approximation of ReLU activation based on [26] is evaluated.

Layer	Input shape	Output shape	k_i	k_o
Conv1 (3×3)	(3, 32, 32)	(64, 32, 32)	1	1
Conv2 (3×3)	(64, 32, 32)	(64, 32, 32)	1	1
AvgPool1 (2×2)	(64, 32, 32)	(64, 16, 16)	1	2
Conv3 (3×3)	(64, 16, 16)	(64, 16, 16)	2	2
Conv4 (3×3)	(64, 16, 16)	(64, 16, 16)	2	2
AvgPool2 (2×2)	(64, 16, 16)	(64, 8, 8)	2	4
Conv5 (3×3)	(64, 8, 8)	(64, 8, 8)	4	4
Conv6 (1×1)	(64, 8, 8)	(64, 8, 8)	4	4
Conv7 (1×1)	(64, 8, 8)	(16, 8, 8)	4	4
FC (1,024×10)	1,024 (flattened)	10	-	-

of the weights computed for each of the 200 epochs and the time it takes on CPU to execute these algorithms for each architecture. For the Epoch precision we report the mean absolute difference of the losses collected over 200 epochs. With 32-bit architecture in comparison to state-of-the-art implementations we expect to see a reduction in die area, silicon cost and power consumption with no loss of latency if the architecture allows parallel processing of the Residue Number System limbs. On CPU though, as the table 5 shows, the 32-bit architecture executed serially takes more execution time than 64-bit architecture.

5.2 CNN Inference with Multiplexed Packing

As a baseline, we implement a privacy-preserving Convolutional Neural Network (CNN) inference workload using the Lattigo FHE library [28] based on the multiplexed packing method [26], which is shown in Figure 3. Multiplexed packing fills the empty slots arising from strided convolution and pooling layers with other available channels from the feature map to utilize the slots in a ciphertext as much as possible. A detailed specification of our CNN model is summarized in Table 6. The model consists of seven convolutional layers and achieves an accuracy of 90.89% for 10,000 CIFAR-10 validation images when evaluated in the unencrypted (FP32) form.

Table 7: Accuracy and average single-thread CPU per-inference execution time (time / infer.) of CNN based on Lattigo with the parameter sets in Table 2. The values are measured with 3,000 random samples from the CIFAR-10 validation images. (∗) Refer to Section 5.4 on the CPU performance of composite scaling workloads.

64-bit set (single scaling)		32-bit set (comp. scaling)	
Accuracy	Time / infer.	Accuracy	Time / infer.
90.97%	376 s	90.97%	895 s$^{(*)}$

We test our implemented CNN model with the two parameter sets in Table 2: a 64-bit set using single scaling and a 32-bit set using composite scaling. Table 7 shows the accuracy and execution time of inference for each parameter set. Execution time is measured using a single CPU thread on an AMD EPYC 7473X system with 512GB of DDR4-3200 memory.

We observe that applying composite scaling does not degrade the accuracy of FHE CNN inference. In both parameter sets, our CNN model achieves the same accuracy of 90.97% (2,729 correctly inferred images out of 3,000). Observed accuracy values are similar to the unencrypted inference accuracy of 90.89% and do not differ depending on the scaling method.

5.3 CNN Inference with Longitudinal Packing

The data packing scheme proposed in CryptoNets [17] enables inference of multiple images in a single forward pass. This makes it suitable for batch processing and high-throughput applications, unlike the multiplexed packing described in Section 5.2. Henceforth, we interchangeably refer to longitudinal packing to the data packing technique used in CryptoNets [17]. Our choice is motivated by its implementation simplicity and friendliness to homomorphic encryption with the CKKS scheme. Regarding the latter, with respect to parameterization, the higher the ring size N, the more images we can process in a single evaluation. Additionally, the operations involved do not require rotations, except in the bootstrap layers; thus, effectively minimizing the number of rotations. We implement the longitudinal CNN inference as described in Table 8. Illustration of the longitudinal packing for the image data and convolution filter weights are shown in Figures 4 and 5, respectively.

5.3.1 Convolution Filters with Longitudinal Packing. Each convolution filter weight $f[j, i, x, y]$ is packed in a single ciphertext and replicated across all the available slots, as depicted in Figure 6(a). Then, the convolution reduces to element-wise multiplications and element-wise additions, as depicted in Figure 6(b). Algebraically, this can be simply expressed as Equation 5.3.1, where the image I is encrypted as a collection of ciphertexts $E(I[t])$ and the convolution weights f are encoded as a collection of plaintexts $P(f[t'])$, where

Table 8: The specification of our CryptoNets-like CNN inference model. Input and output feature map shapes are expressed as tuples of (channel, height, weight). All convolutional (Conv) layers have strides of one and average pool layer have stride 2. B is the number of input ciphertexts to each layer, which in turn is the output by previous layer. K denotes the number of ciphertexts required by to encrypt the weights of a convolution filter. After each Conv layer, bootstrapping is employed on its output which in turn is input to a precise polynomial approximation of ReLU activation (ReLU layer) based on 59-degree Chebyshev Polynomials with bounds [-13,13] is evaluated.

Layer	Input shape	Output shape	B	K
Input	(3, 32, 32)	-	3072	-
Conv1 (3×3)	(3, 32, 32)	(64, 32, 32)	3072	1728
Conv2 (3×3)	(64, 32, 32)	(64, 32, 32)	65536	36864
AvgPool1 (2×2)	(64, 32, 32)	(64, 16, 16)	65536	-
Conv3 (3×3)	(64, 16, 16)	(64, 16, 16)	16384	36864
Conv4 (3×3)	(64, 16, 16)	(64, 16, 16)	16384	36864
AvgPool2 (2×2)	(64, 16, 16)	(64, 8, 8)	16384	-
Conv5 (3×3)	(64, 8, 8)	(64, 8, 8)	4096	36864
Conv6 (1×1)	(64, 8, 8)	(64, 8, 8)	4096	4096
Conv7 (1×1)	(64, 8, 8)	(16, 8, 8)	4096	1024
FC (1024×10)	1024	10	1024	10240

$t \in \mathbb{N}^3$ and $t' \in \mathbb{Z}^4$ are indexing tuples.

$$E(I * f[j,:]) = \sum_{i=0}^{C} \sum_{u=-\frac{k}{2}}^{\frac{k}{2}} \sum_{v=-\frac{k}{2}}^{\frac{k}{2}} E(I[i, x+u, y+v]) \odot P(f[j,i,u,v]),$$

where $0 \le x+u < W$
and $0 \le y+v < H$.

Note that j spans the number of output feature maps, i spans the number of input feature maps, x and y iterate over the width and height of the image, u and v iterate over the convolution filter weights in a $k \times k$ window, where k is typically one of the odd numbers 3 or 1.

5.3.2 Average Pooling with Longitudinal Packing. Input ciphertexts to the average pooling layer are the results of convolution followed by bootstrapping and relu; thus, the input shape will be the same as the output shape of the previous immediate convolution layer. All average pooling layers have stride 2 and 2x2 window, effectively reducing the number of ciphertexts by a factor of 4. As can be expected, the computation from a single pooling window involves 3 element-wise additions and a single scalar multiplication with the constant 0.25.

5.3.3 The Implications of Longitudinal Packing for CNN Inference. We summarize below a series of implications that can be expected from using longitudinal packing as the choice of data encoding strategy for a convolution neural network:

(1) Implementation is simple and easy to understand.
(2) High throughput per single inference for larger ring sizes.

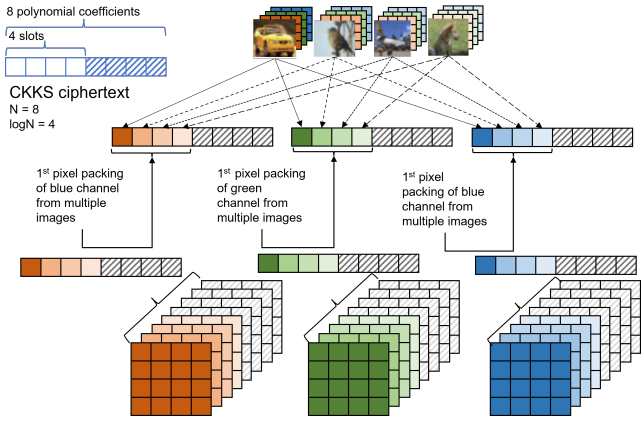


Figure 4: Longitudinal packing of image data per channel. Four 3x3 images encrypted into a collection of 27 ciphertexts, using 8 as the ring size, which leaves 4 slots available to store a pixel from each of the four images.

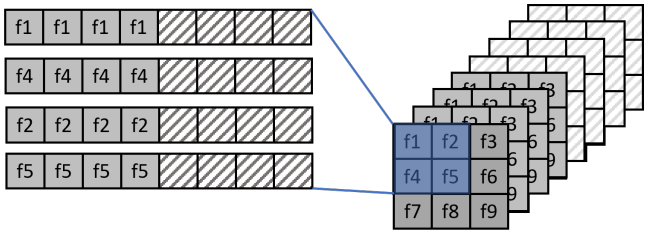
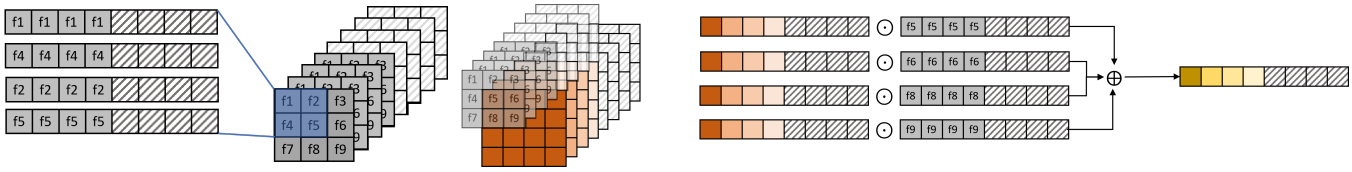


Figure 5: Longitudinal packing of 2D convolution filter of 3x3 size.

(3) It requires high RAM memory capacity, especially for large ring sizes.
(4) High memory footprint during execution.
(5) Each layer's computation can be reduced to element-wise or scalar multiplications and element-wise additions.
(6) No rotation is required for the main bulk of computation in convolution, except in bootstrapping.

On what concerns correctness validation, we provide an empirical analysis of the arithmetic precision robustness of 32-bit composite scaling with respect to 64-bit single scaling using several statistical hypothesis tests. To assess the closeness of data distributions of the outputs by the two approaches, we perform a two-tailed two-sample z-test to determine if the means of their populations are significantly different. We perform the z-test statistics for each layer. All the tests produce z-values ≤ 0.0003 and p-values ≥ 0.9997; thus, with confidence level of 99%, the tests fail to reject the null hypothesis, indicating that there is statistical significant relationship between the data distributions of all outputs.

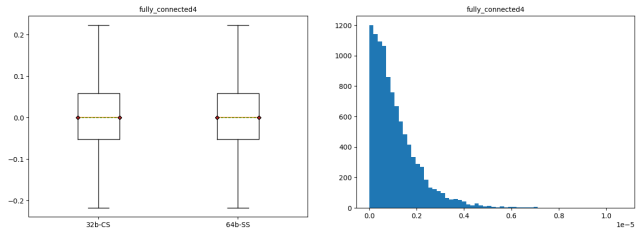
As for the arithmetic precision differences between the outputs of the 32-bit composite scaling and 64-bit single scaling methods, we perform a left-tailed one-sample z-test. The null hypothesis H_0 states that the absolute difference in precision has mean $\mu_0 \ge 10^{-5}$. The alternative hypothesis H_A states the opposite, *i.e.* $\mu_A <$



(a) Longitudinal packing of 2D convolution filter of 3x3 size.

(b) Illustration of how it convolution works with longitudinal packing. It reduces to element-wise operations, $\odot$ and $\oplus$.

Figure 6: Longitudinally-packed 2d convolution filter operating on a longitudinally-packed 3x3 red channel image. It consists of element-wise multiplications between corresponding weight and pixel ciphertexts followed by the element-wise summation of all ciphertexts resulted from the multiplications.



(a) Boxplot of data distribution of outputs.

(b) Histogram of absolute arithmetic precision differences.

Figure 7: Boxplots (on the left) and histogram of the precision differences (on the right) in the outputs of the fully connected (prediction) layer.

10^{-5}. We use the absolute difference of the outputs of the two methods as the sample data to perform the left-tailed paired z-test statistics. All the tests produce very small calculated z-values, tending to $-\infty$ and accordingly p-values = 0; thus, with confidence level of 99%, the tests successfully rejected the null hypothesis for all layers, including the prediction layer (the fully connected layer), confirming that most differences in arithmetic precision happens after the 5^{th} digit after the decimal point.

In particular, the fully connected layer had the following absolute precision difference statistics: sample mean $\bar{X} = 1.1097e^{-6}$, standard deviation $\sigma = 1.0029e^{-6}$, maximum absolute difference as $1.0669e^{-5}$, minimum absolute difference as 0, number of samples $N_{fc} = 10240$, calculated z-value as -897.0333 and p-value equal to 0. As can be observed, most differences in arithmetic precision occurs around the 6^{th} digit after the decimal point. In Figure 7, we show the boxplots comparing the data distribution statistics of the two approaches in the fully connected layer output, as well as the histogram of their absolute precision differences.

In summary, both hypothesis tests, for all layers, provide desirable outcomes that corroborate only negligible difference in arithmetic precision when running CNN-I workload on the CIFAR10 dataset using the 32-bit composite scaling strategy compared to the 64-bit single scaling's.

5.4 Workload Execution time

Because most modern CPUs utilize 64-bit word size, utilizing 32-bit composite scaling on CPUs has limited benefits. On CPUs, the execution times of the workloads when using 32-bit composite scaling increases. When using 32-bit composite scaling, logistic regression training takes 1.7× longer, CNN inference takes 2.38× (multiplexed packing) compared to when using 64-bit single scaling.

For CNN inference with multiplexed packing, the slowdown is mainly due to our 32-bit parameter set implementation being a proof-of-concept; our implementation still uses the 64-bit word size internally. Revisiting Table 4, if we regard that the word size W is fixed to 64 bits in both of the parameter sets, WSum of the 32-bit set will become 2.29× larger (2,969M) than that of the 64-bit set (1,297M), which explains the slowdown.

Although composite scaling may not be beneficial in terms of performance on CPUs, graphics processing units (GPUs) and custom hardware can substantially benefit from it. As a representative example, NVIDIA GPUs do not natively support 64-bit integer operations and emulate them by compositions of 32-bit (or 16-bit) integer operations. Therefore, by simply utilizing native 32-bit integer operations with 32-bit composite scaling, higher throughput can be achieved for GPUs. In custom ASIC or FPGA hardware, 32-bit datapath will reduce the hardware cost as analyzed in Section 4.2. In fact, SHARP [23] has shown the effectiveness of double scaling in reducing the chip area and power of an ASIC FHE accelerator.

We leave the implementation of 32-bit composite scaling on GPUs and design of 32-bit hardware architecture for FHE CKKS as promising future work.

6 RELATED WORK

The early implementation work of FHE begins in the form of various open-sourced software libraries [2, 11, 28, 33] targeting 64-bit CPU architecture. However, with high computational overhead and memory consumption, these software libraries cannot be leveraged as is on the fixed but small word-size architectures. Therefore, the performance of these software libraries (both in terms of achievable precision and execution time) on small word-size architecture remains unknown. As a result, Natarajan and Dai [29] proposed SEAL-Embedded, a 32-bit version of the SEAL library featuring the CKKS homomorphic encryption scheme. This library achieves impressive performance gains but is limited in terms of the circuit depth it can evaluate. To further improve the performance of SEAL-Embedded, Li et al. [27] proposed a combination of SEAL-Embedded and homomorphic encryption acceleration library (HEXL) [6] to

accelerate deep learning using 32-bit word-size operations. However, they do not implement bootstrapping or evaluate the precision achieved by the deep learning workloads with 32-bit word-size.

GPUs being the next natural choice for FHE implementation, there exists several works [15, 21, 34, 35] that perform either 32-bit or 64-bit GPU implementations for various FHE schemes. Despite the optimization efforts by these works, the existing 64-bit GPU implementations suffer from a significant performance bottleneck. This is because 64-bit integer operations are emulated using multiple 32-bit integer operations, giving rise to computational overhead. Moreover, the 32-bit GPU implementations still suffer from the precision that they can achieve for real-world privacy-preserving applications without the use of composite scaling. Our proposed composite scaling approach will be extremely beneficial to implementing real-world privacy-preserving applications requiring high-precision using GPUs having 32-bit word-size.

Similar to GPUs, FHE implementation on FPGAs can witness a significant performance improvement with our proposed composite scaling approach. FPGAs have small word-size fast multipliers and adders (27 bits in Intel FPGAs) within the DSP blocks. Typical approach to leverage these blocks for performing fast large word-size (54-64 bits) arithmetic operations is through multi-precision arithmetic approach [30]. However, using multi-precision arithmetic approach leads to higher latency on FPGA-based FHE implementations. Instead, with composite scaling approach, these FPGA implementations can have reduced latency while still achieving a higher operating frequency.

In the recent 2-3 years, there has been growing trend on designing custom hardware accelerators for FHE. The 64-bit architectures include Bootstrappable, Technology-driven, Secure accelerator (BTS) [25] and an Accelerator for FHE with Runtime data generation and inter-operation Key reuse (ARK) [24] designs while the 32-bit or below architectures include F1 [31] and CraterLake [32] designs. These state-of-the-art ASIC solutions rely on large amount of on-chip memory, computational units, and power to process the large working data sets (several 100 MBs) that can make them expensive to operate. To deliver high performance at a greatly reduced hardware cost, a Short-word Hierarchical Accelerator for Robust and Practical FHE (SHARP) [23] is the first work that explored a wide range of small word-sizes that can be practical for various privacy-preserving applications.

7 CONCLUSION

A prevalent issue in RNS-CKKS is the challenge of efficiently achieving high precision on hardware architectures with a fixed, yet smaller, word-size. In this work, we provide an efficient solution to the problem that we refer to as composite scaling. In this approach, we group multiple RNS primes as $q_\ell := \prod_{i=0}^{t-1} q_{\ell,j}$ such that $\log q_{\ell,j} < W$ for $0 \leq j < t$, and use each composite q_ℓ in the rescaling procedure as $\mathrm{ct} \mapsto \lfloor \mathrm{ct}/q_\ell \rceil$. The larger the composition degree t the higher is the precision that can be achieved with RNS-CKKS under a wide range of secure parameters for workload deployment.

Our approach makes RNS-CKKS more flexible by allowing to express precision requirements of applications and data sets at much finer grains, and for most real workloads for which bootstrapping is required, composite scaling combined with META-BTS finally enable to support high precision RNS-CKKS in arbitrary word-size architectures.

We enable composite scaling RNS-CKKS in the OpenFHE and the Lattigo libraries through concrete implementation of the method and known to date workloads, namely logistic regression training and convolutional neural network inference. Our experiments demonstrate that single and composite scaling are functionally equivalent, both theoretically and practically.

Finally, using composite scaling in RNS-CKKS can allow even more resource-constrained devices, e.g., IoT and Edge devices, typically driven by 32-bit microcontrollers, to achieve higher-precision efficiently in RNS-CKKS. Similar considerations hold for GPUs.

REFERENCES

[1] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2023. FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 882–895.

[2] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, et al. 2022. Openfhe: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 53–63.

[3] Ahmad Al Badawi, Yuriy Polyakov, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Kurt Rohloff. 2021. Implementation and Performance Evaluation of RNS Variants of the BFV Homomorphic Encryption Scheme. *IEEE Transactions on Emerging Topics in Computing* 9, 2 (2021), 941–956. https://doi.org/10.1109/TETC.2019.2902799

[4] Youngjin Bae, Jung Hee Cheon, Wonhee Cho, Jaehyung Kim, and Taekyung Kim. 2022. META-BTS: Bootstrapping Precision Beyond the Limit. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. ACM, New York, 223–234. https://doi.org/10.1145/3548606.3560696

[5] Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. 2016. A Full RNS Variant of FV like Somewhat Homomorphic Encryption Schemes. In *International Conference on Selected Areas in Cryptography (SAC 2016)*. Springer, Cham, 423–442. https://doi.org/10.1007/978-3-319-69453-5_23

[6] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, and Vinodh Gopal. 2021. Intel hexl: Accelerating homomorphic encryption with intel avx512-ifma52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 57–62.

[7] Antoon Bosselaers, René Govaerts, and Joos Vandewalle. 1994. Comparison of Three Modular Reduction Functions. In *Advances in Cryptology — CRYPTO' 93*. Springer, Berlin, Heidelberg, 175–186. https://doi.org/10.1007/3-540-48329-2_16

[8] Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2021. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In *Advances in Cryptology – EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part I*. Springer, Cham, 587–617. https://doi.org/10.1007/978-3-030-77870-5_21

[9] Jean-Philippe Bossuat, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2022. Bootstrapping for Approximate Homomorphic Encryption with Negligible Failure-Probability by Using Sparse-Secret Encapsulation. In *International Conference on Applied Cryptography and Network Security (ACNS 2022)*. Springer, Cham, 521–541. https://doi.org/10.1007/978-3-031-09234-3_26

[10] Hao Chen, Ilaria Chillotti, and Yongsoo Song. 2019. Improved bootstrapping for approximate homomorphic encryption. In *Advances in Cryptology – EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part II*. Springer, Cham, 34–54. https://doi.org/10.1007/978-3-030-17656-3_2

[11] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. Bootstrapping for approximate homomorphic encryption. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part I 37*. Springer, 360–384.

[12] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2019. A full RNS variant of approximate homomorphic encryption. In *Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25*. Springer, 347–368.

[13] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and*

Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23. Springer, 409–437.

[14] Leo de Castro, Rashmi Agrawal, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, Chiraag Juvekar, and Ajay Joshi. 2021. Does fully homomorphic encryption need compute acceleration? *arXiv preprint arXiv:2112.06396* (2021).

[15] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. 2023. Tensorfhe: Achieving practical computation on encrypted data using gpgpu. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 922–934.

[16] Craig Gentry, Shai Halevi, and Nigel P Smart. 2012. Homomorphic evaluation of the AES circuit. In *Annual Cryptology Conference*. Springer, 850–867.

[17] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*. PMLR, 201–210.

[18] Kyoohyung Han and Dohyeong Ki. 2020. Better bootstrapping for approximate homomorphic encryption. In *Topics in Cryptology–CT-RSA 2020: The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24–28, 2020, Proceedings*. Springer, 364–390.

[19] Ian Quah and Ahmad Al Badawi and David Bruce Cousins and Yuriy Polyakov. 2023. OpenFHE-Based Examples of Logistic Regression Training using Nesterov Accelerated Gradient Descent. https://github.com/openfheorg/openfhe-logreg-training-examples. (2023). Accessed: 2023-07-23.

[20] David Ireland. 2001. Google. https://code.google.com/archive/p/libmcrypto/. (2001).

[21] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 2021. Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 114–148.

[22] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. 2022. Approximate homomorphic encryption with reduced approximation error. In *Topics in Cryptology–CT-RSA 2022: Cryptographers' Track at the RSA Conference 2022, Virtual Event, March 1–2, 2022, Proceedings*. Springer, 120–144.

[23] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. 2023. SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–15.

[24] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. 2022. Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1237–1254.

[25] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. Bts: An accelerator for bootstrappable fully homomorphic encryption. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 711–725.

[26] Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. 2022. Low-Complexity Deep Convolutional Neural Networks on Fully Homomorphic Encryption Using Multiplexed Parallel Convolutions. In *Proceedings of the 39th International Conference on Machine Learning*, Vol. 162. PMLR, 12403–12422.

[27] Xinyi Li, Masaki Nishi, Teppei Shishido, and Keiji Kimura. 2022. Acceleration of HE-Transformer with bit reduced SEAL and HEXL. *IEICE Technical Report; IEICE Tech. Rep.* (2022).

[28] Christian Vincent Mouchet, Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2020. Lattigo: a Multiparty Homomorphic Encryption Library in Go. In *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography (WAHC '20)*. 64–70. https://doi.org/10.25835/0072999

[29] Deepika Natarajan and Wei Dai. 2021. Seal-embedded: A homomorphic encryption library for the internet of things. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 756–779.

[30] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. HEAX: An architecture for computing on encrypted data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1295–1309.

[31] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 238–252.

[32] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. 2022. Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 173–187.

[33] SEAL 2023. Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL. (Jan. 2023). Microsoft Research, Redmond, WA.

[34] Kaustubh Shivdikar, Gilbert Jonatan, Evelio Mora, Neal Livesay, Rashmi Agrawal, Ajay Joshi, José L. Abellán, John Kim, and David Kaeli. 2022. Accelerating Polynomial Multiplication for Homomorphic Encryption on GPUs. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. 61–72. https://doi.org/10.1109/SEED55351.2022.00013

[35] Yujia Zhai, Mohannad Ibrahim, Yiqin Qiu, Fabian Boemer, Zizhong Chen, Alexey Titov, and Alexander Lyashevsky. 2022. Accelerating encrypted computing on intel gpus. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 705–716.