Grafting: Decoupled Scale Factors and Modulus in RNS-CKKS

Jung Hee Cheon
Seoul National University
Seoul, Republic of Korea
CryptoLab Inc.
Seoul, Republic of Korea
jhcheon@snu.ac.kr

Hyeongmin Choe
CryptoLab Inc.
Seoul, Republic of Korea
hyeongmin.choe528@gmail.com

Minsik Kang
Seoul National University
Seoul, Republic of Korea
kaiser351@snu.ac.kr

Jaehyung Kim
Stanford University
Stanford, California, USA
jaehk@stanford.edu

Seonghak Kim
CryptoLab Inc.
Seoul, Republic of Korea
ksh@cryptolab.co.kr

Johannes Mono
Ruhr University Bochum
Bochum, Germany
CryptoLab Inc.
Seoul, Republic of Korea
jmono@cryptolab.co.kr

Taeyeong Noh
CryptoLab Inc.
Seoul, Republic of Korea
tynoh0219@cryptolab.co.kr

ABSTRACT

The CKKS Fully Homomorphic Encryption (FHE) scheme enables approximate arithmetic on encrypted complex numbers for a desired precision. Most implementations use RNS with carefully chosen parameters to balance precision, efficiency, and security. However, a key limitation in RNS-CKKS is the rigid coupling between the scale factor, which determines numerical precision, and the modulus, which ensures security. Since these parameters serve distinct roles—one governing arithmetic correctness and the other defining cryptographic structure—this dependency imposes design constraints, such as a lack of suitable NTT primes and limited precision flexibility, ultimately leading to inefficiencies.

We propose *Grafting*, a novel approach to decouple scale factors from the modulus by introducing *(universal) sprouts*, reusable modulus factors that optimize word-sized packing while allowing flexible rescaling. With the universal property, sprouts allow rescaling by arbitrary bit-lengths and key-switching at any modulus bit-length without requiring additional key-switching keys. Decoupling the scale factor from the modulus in Grafting yields significant efficiency gains: (1) Optimized RNS packing by decomposing the modulus into machine word-sized components, accelerating computations and reducing the ciphertext and encryption/evaluation key sizes; and (2) A freely adjustable scale factor independent of the modulus, unifying the ring structure across applications and reducing modulus consumption through adaptive scalings.

Our experiments demonstrate that Grafting improves performance across standard SHE/FHE parameter sets for ring dimensions 2^{14}-2^{16} by up to 1.83× and 2.01× for key-switchings and multiplications, respectively, and up to 1.92× for bootstrapping. Grafting also reduces public key and ciphertext sizes by up to 62% without compressions, maintaining the same number of public keys as before. As an application, we showcase the CKKS gate bootstrapping for bits (Bae et al.; Eurocrypt'24), achieving 1.89× speed-up due to the reduced number of RNS factors. Finally, we revisit the homomorphic comparison (Cheon et al.; Asiacrypt'20), evaluating it with carefully chosen scale factors for each iteration, reporting up to 204-bit fewer modulus consumption (27% reduction) in the standard parameter set, without precision loss.

KEYWORDS

Homomorphic Encryption, CKKS, Residue Number System, Bootstrapping

1 INTRODUCTION

The Cheon–Kim–Kim–Song (CKKS) scheme [CKKS17] is a Fully Homomorphic Encryption (FHE) scheme that allows approximate computation over real/complex-valued encrypted data. For real/complex-valued computations, the precision is determined by the *scale factors* used for each homomorphic operation. It is multiplied with the clear-text messages and rounded, encoding the real/complex messages into integers via Discrete Fourier Transform (DFT). Therefore, the sequence of scale factors becomes a key parameter to ensure arithmetic correctness at the desired precision level.

CKKS bases its IND-CPA security on the hardness of the Ring-Learning-With-Errors (RLWE) problem, where the modulus budget is limited for security. As in other RLWE-based FHE schemes, a huge modulus budget of hundreds to thousands of bits is used for its underlying cryptographic structure, the polynomial ring. To efficiently handle the huge modulus, every competitive library [Cry22, ABBB+22, SEA23, lat24] uses the Residue Number System (RNS) and Number Theoretic Transform (NTT) [CHK+19],

Part of this work was done while Hyeongmin Choe was affiliated with Seoul National University, and while Jaehyung Kim was affiliated with CryptoLab Inc.

decomposing the huge modulus into RNS factors, each of which is a NTT prime. The use of RNS in CKKS enables significantly more efficient homomorphic operations but introduces some restrictions: the linkage between the scale factors and the RNS factors. As the scale factors are multiplied during homomorphic multiplications, it necessitates *rescaling* to revert its size. This procedure not only scales down the scale factors but also the ciphertext modulus, consuming the modulus budget. Thus, an RNS factor should exist in the modulus, which has a size similar to the scale factor; otherwise, it introduces a larger error that sometimes exponentially explodes [CHK+19, KPP22, AAB+23] necessitating a careful selection of the RNS factors.

In RNS-CKKS, the two key parameters–the scale factor, which governs computational accuracy, and the ciphertext modulus, which provides cryptographic structure–are tightly coupled, significantly restricting flexibility in parameter selection. Ideally, the scale factor should be adjusted according to the application's required precision to minimize modulus consumption. However, since the scale factor is intrinsically linked to the modulus structure, any change requires regenerating not only the ciphertext modulus but also the key modulus and the public keys accordingly, adding substantial management overhead. As a result, most deployments rely on general-purpose or pre-configured parameter sets.

Recent advances, however, have explored application-dependent parameter optimization by fine-tuning scale factors to match precision needs more precisely. In bootstrapping, for instance, performance can be improved by assigning different scale factors to each step based on individual precision requirements, which results in faster execution and more efficient use of the available modulus [Cry22, lat24]. This leads to better throughput in terms of amortized bootstrapping time per available level. Also, the use of fine-grained scale factors for different purposes reduces the overall modulus consumption [KPK+22, CCKS23, SSKM24], resulting in better running time.[1] It is also demonstrated that highly efficient low-precision operations using small scale factors (e.g., 25–35 bits) can be designed [BCKS24, BKSS24], enabling the use of smaller ring dimensions, yielding benefits in both speed and memory footprint.

However, due to its bonded nature, it is unable to construct parameters with arbitrary scale factors for RNS-CKKS. In particular, there are not enough small NTT primes for typical CKKS ring dimensions of 2^{14} to 2^{17}, limiting the parameter designs. Furthermore, when it comes to hardware implementation, we suffer from the smaller word size of the machine, introducing another restriction to the size of the scale factors to be less than the machine's word size, say, 32 bits.

Furthermore, the RNS factors following the scale factor sizes significantly deteriorate performance and memory usage–it hinders optimal utilization of the machine's computational resources. Since CKKS scale factors typically range from 30 to 60 bits, the ciphertext modulus must be decomposed into RNS primes of similar sizes. When these scale factors are utilized, the number of RNS factors significantly increases compared to the optimal case. Given that the performance of homomorphic operations scales linearly (and sometimes quadratically) with the number of RNS factors, this increase leads the overall performance to be far from optimal.

[1]via decreasing the number of levels used, or the number of required bootstrappings.

Until now, decoupling the scale factors and the ciphertext modulus has been a challenging problem. Precisely, the CKKS ciphertext is a tuple (a, b) over a polynomial ring $\mathcal{R}_Q = \mathbb{Z}[X]/(X^N+1)$, where N is the ring dimension, which is a power-of-two integer and Q is a positive integer modulus. Each ciphertext has its multiplicative level ℓ, corresponding modulus Q_ℓ, and scale factor Δ_ℓ. Rescaling at level ℓ essentially divides the squared scale factor Δ_ℓ^2 by q_ℓ, resulting in a scale factor $\Delta_{\ell-1} = \Delta_\ell^2/q_\ell$. To slightly detour this relationship, Agrawal et al. [AAB+23] introduced the composite rescaling using the composition of the RNS factors instead of a single RNS factor q_ℓ, to make the moderate precision RNS-CKKS available in the hardware implementation. However, this does not actually decouple the scale factors and the modulus; it allows the modulus factors to be crafted to compose them roughly the size of the scale factor. As a side effect, the number of RNS factors increases, leading to a slower running time in the larger word-size machines, e.g., 64-bit CPUs, but a better performance in the machines with smaller word sizes, e.g., 32-bit GPUs.

Another approach of Mono et al. [MG23] was to adopt the nature of using mostly word-sized primes from the Double-CRT format in BGV [HS20] to RNS-CKKS. The ciphertext modulus can be switched from one to another having a similar size while ensuring that the new modulus has a similar size to control the size of the error newly introduced by the modulus switching. For example, when the scale factor is 36-bit, one can construct a ciphertext modulus with 54-bit NTT primes, and replace two 54-bit primes into three 36-bit primes before rescaling. This allows for a smaller number of RNS factors in the ciphertext modulus and decreases the overall computation time. Despite that, this approach reduced the number of RNS factors and improved the homomorphic computation efficiency; however, it was restricted to particular cases and pre-determined scale factors.

Samardzic and Sanchez [SS24] extended this approach, suggesting a similar concept of modulus managing technique, focusing on designing a hardware accelerator. Fusing the modulus switching with the CKKS rescaling,[2] the ciphertext modulus can be rescaled with a rational factor, from Qa to Qb, for some greatest common divisor $Q \in \mathbb{Z}$ of the moduli and the integer factors a and b, where $a/b \approx \Delta$. Given the largest possible modulus and a sequence of desired rescaling factors, one may find a descending chain of ciphertext moduli, where each modulus is a composition of the machine's word-size NTT primes with possibly several smaller NTT primes. Again, the application-dependent scale factors must be pre-determined for public key generation. Also, it is not compatible with the RNS-CKKS key-switching algorithm, the most frequent and heavy operation in HE, which is used for homomorphic multiplication or rotation. For key-switching, the ciphertext moduli need to be divisible by one another so that the least common multiple (LCM) of the ciphertext moduli becomes a factor of the key-switching key modulus. Without divisibility, the LCM becomes too large, pushing the key-switching key modulus beyond the secure budget. To stay within the secure modulus budget, multiple key-switching keys must be generated for each operation type–e.g., several relinearization keys for different multiplicative levels or multiple rotation keys for each rotation amount. This significantly increases the size of the public key, which already

[2]This concept was concurrently introduced in an earlier version of this work.

ranges from hundreds of megabytes to tens of gigabytes, amplifying the communication overhead in the typical secure outsourced computation scenarios.

None of the prior approaches efficiently resolve the tight coupling between the scale factors and the ciphertext modulus, and the following question arises:

*C*an we completely decouple the scale factors from the modulus, increasing the efficiency of the homomorphic computations while preserving the key-switching key sizes?

1.1 Our Contribution

We introduce *Grafting*, a modulus management framework for RNS-CKKS that completely decouples scale factors from the ciphertext modulus. This separation enables full utilization of the machine's computational and memory capabilities by allowing an optimal number of RNS factors. As a consequence, Grafting yields both a runtime speed-up and a significant reduction in the size of public keys proportional to the decrease in the number of RNS factors. Furthermore, the full decoupling allows application-independent parameterization–a single parameter set can universally support a wide range of applications. Moreover, unlike conventional schemes where scale factors must be rigidly determined during public key generation, Grafting supports flexible selection of scale factors even after the public parameters and the keys are fixed. This flexibility leads to substantial savings in modulus consumption, further improving overall efficiency.

High-level Overview of Grafting. Grafting is built on top of the RNS-CKKS multiplications and key-switchings but manages the ciphertext and switching key moduli differently. As a useful tool, we first introduce the rational version of the CKKS rescale operation, rescaling by a rational number instead of an integer similar to [SS24], namely the *rational rescale*. It is a composition of integer multiplication, and the usual CKKS rescale, i.e., modulus changing from Qa to Qab, then rescaled down to Qb, at the same time, the scale factor is also updated from Δ to $\Delta \cdot b$, then $\Delta \cdot b/a$ for some integers Q, a, and b. We proved that the error introduced by rational rescaling is the same amount as the usual rescaling, allowing the ciphertext modulus to be decomposed with the machine's word-sized factors as much as possible. However, this is not enough for efficient key-switching, requiring the divisibility between different ciphertext moduli. We introduce *(universal) sprout*, a small and reusable part of the ciphertext modulus, so that it can introduce divisibility to the ciphertext moduli. For instance, we can have a power-of-two integer 2^α as an RNS factor, composing the modulus $Q = q_0 q_1 \cdots q_\ell \cdot 2^\alpha$ with the word-size primes q_i for an integer α smaller than the machine word's bit-length, ω. Any two moduli can divide one another, allowing key-switchings as in RNS-CKKS. Based on this, we revisit the RNS-CKKS key-switching algorithms to allow the RNS factors of small power-of-two integers while keeping the NTT states. We illustrate how the modulus changes in Grafting in Figure 1.

Applications. As Grafting breaks the linkage between the scaling factors and the RNS factors, it can solve many restrictions that previous implementations had. Grafting optimally packs the modulus using mostly machine word-sized factors, resulting in fewer RNS components. This reduction not only accelerates homomorphic operations but also decreases the ciphertext and public key sizes, roughly in proportion to the reduced number of factors. In this respect, we revisit the Somewhat HE (SHE) and Fully HE (FHE) parameters in the literature [BCC+24, Cry22, lat24, ABBB+22] and re-design their grafted variants with reduced RNS factors by up to 41%.

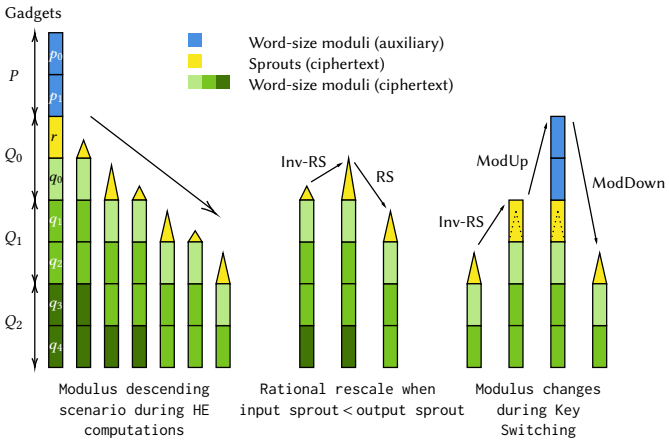

Figure 1: Modulus changes in Grafting.

Also, we do not need to suffer from the lack of small NTT primes or scale factors larger than the machine's word size, which allows optimized performance for lower—and/or higher-precision computation. In addition, Grafting is particularly effective in applications that use smaller-than-usual scale factors [CCKS23, BCKS24, BKSS24] where the original number of RNS factors is far from optimal. In this regard, we revisit the Bit-CKKS [BCKS24], which uses the scale factors of sizes down to 26 bits. The grafted implementation can reduce the number of RNS factors by up to 46%. Further exploiting the benefits of Grafting, we introduce an additional parameter set in $\log_2 N = 14$, that cannot be implemented in RNS-CKKS–3 available levels after binary gate bootstrapping instead of 2 available levels.

On the other hand, because of the decoupled nature, the modulus can be independent of the application. In this respect, in Grafting, a single underlying parameter can be used universally in different applications and/or precisions. Further, the scale factors can be freely changed for each sub-procedure of homomorphic computations. Like adaptive-precision techniques in clear-text computation, such as those used in machine learning training/inference or Newton-like iterative methods, the encrypted counterpart also enhances performance by reducing overall modulus consumption through carefully tuned scale factors at each sub-procedure. As an example, we revisit the iteration-based homomorphic comparison technique [CKK20], utilizing fine-grained scale factors for each iteration, ensuring the accuracy of the homomorphic comparison. We theoretically demonstrate that the scale factors can be adjusted smaller in the early iterations, reducing the modulus consumption, possibly reducing the number of bootstrappings in wider applications such as the ReLU function evaluation for neural networks [LLL+22, LLKN22, RKP+25] or lookup table evaluations [DMPS24, CCP24].

Parameter	Speed-ups			Key-size
	KeySwitch	Mult	BTS	reduction
SHE15	1.29×	1.42×	-	42% ↓
FHE15s	1.72×	1.78×	1.92×	62% ↓
BinFHE16	1.92×	2.07×	1.81×	25% ↓

Table 1: Gains obtained in the standard SHE [BCC+24], FHE [Cry22], and Bit-CKKS [BCKS24] parameter sets using Grafting.

Implementation and Experimental Results. We implemented Grafting with RNS-CKKS in C++, relying on most of the well-known optimization techniques. To compare the performance, we also implemented the non-grafted variant; we will refer to it as a *simple* variant. Based on the re-designed parameters for SHE and FHE, we achieved speed-ups of up to 2.01× for multiplications and 1.92× for bootstrappings, and reduced the key sizes by up to 62% without compression. Note that the available modulus after bootstrapping and the precisions of the homomorphic operations remain the same as before.

For Bit-CKKS [BCKS24], we achieved speed-ups of 1.89× and 1.81× for ring dimensions 2^{14} and 2^{15}. The additional parameter set reports similar timings with the original $\log_2 N = 14$ parameter set, but one more available level.

We summarize the main results in Table 1. Please see Section 5 for more details.

For the homomorphic comparison, we analyze the error behaviors during the polynomial evaluations and pick the scale factors that guarantee the precision as high as before. With the smaller scale factors, we achieved 10-27% less modulus consumption compared to the prior art for evaluating two different homomorphic comparison functions without precision loss. We also demonstrate the reduction in the number of bootstrapping for homomorphic comparison evaluation, thanks to the reduced modulus consumption, naturally accelerating the execution time by $3/2 = 1.5\times$.

Additional Contributions. We provide a few more interesting contributions in the Appendices. First of all, we revisit the Tuple-CKKS multiplication [CCKS23] with Grafting, introducing the grafted variant of the tuple rescaling, relinearization, and multiplication, in Appendix D. Based on our analysis, we expect around 2.4× speed-up for double-CKKS multiplication and enable larger tuple sizes, which was previously unrealistic due to the lack of NTT primes.

Secondly, in Appendix E, we show Grafting's improved flexibility from the decoupled nature: arbitrary-precision homomorphic multiplication up to 113-bit precision and homomorphic linear transformations up to 107-bit precision, based on a single underlying parameter set. This is intrinsically made by utilizing a quadruple-precision library for the scale factors.

Next, we revisit the CKKS bootstrapping in Appendix F and showcase some experimental results: 1) ModRaise to arbitrary modulus, innately supported in Grafting, and 2) EvalMod with reduced modulus consumption, by utilizing fine-tuned scale factors for sub-procedures of EvalModevaluations.

Finally, in Appendix G, we investigate the parameters in the literature and provide their grafted variants with expected speed-ups.

1.2 Additional Related Works

In the introduction, we already mentioned the relevant related works to our work. Following, we present some additional related works; the rest can be found in Appendix A.

RNS-CKKS. The original CKKS [CKKS17] scheme used the power-of-two integers for its ciphertext modulus; however, due to its limited efficiency, the RNS was introduced for CKKS implementations [CHK+19], as in other RLWE-based HE schemes like BGV [BGV12, GHS12] or B/FV [FV12, Bra12, BEHZ16] schemes. Currently, most of the available homomorphic encryption libraries, indeed, every competitive library, are based on RNS for efficiency [HHS+21, ABBB+22, Cry22, SEA23, lat24]. In RNS-CKKS, the ciphertexts and the switching keys are decomposed into integers smaller than the word size; each is a remainder modulo an RNS factor. The computation is done in a decomposed manner, thanks to the CRT. From its RNS-friendly nature, Han and Ki [HK20] introduced an advanced key-switching technique for RNS-CKKS, adopted from [BEHZ16, GHS12], which gave the flexibility of choosing the usable ciphertext modulus compensating the size of the switching keys and the key switching running times. Using the so-called RNS gadget decomposition, the ciphertexts are decomposed into several blocks and multiplied with the corresponding switching keys. By elaborately designing the modulus, its RNS factors, and the gadgets, more efficient bootstrappings can be achieved, accelerating the overall homomorphic computations [KPK+22, KPP22, BMTH21] and libraries [lat24, Cry22]. However, as mentioned earlier, the relationship forces the RNS factors to be set far smaller than the machine's word size, making it hard to fully utilize the machine's computation and memory budget.

2 PRELIMINARIES

2.1 Notations

Polynomials are denoted in bold font and lowercase letters. We let $\lfloor y \rceil$ be a rounding of $y \in \mathbb{R}$ to the nearest integer. We naturally extend the rounding notation to vectors and polynomials by applying it component-wise. For an integer n, we denote a set of non-negative integers smaller than n as $[n]$, i.e., $[n] = \{0, 1, \cdots, n-1\}$.

We let $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ be a polynomial ring where N is a power-of-two integer. For any positive integer Q, let the quotient ring $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R} = \mathbb{Z}_Q[X]/(X^N + 1)$. We let $\mathrm{ct} = (\mathbf{b}, \mathbf{a}) \in \mathcal{R}_Q^2$ be a ciphertext with ciphertext modulus Q with respect to a secret key $\mathbf{s} \in \mathcal{R}$ if it satisfies $\langle \mathrm{ct}, (1, \mathbf{s}) \rangle \approx \mathbf{m} \mod Q$ for some message $\mathbf{m} \in \mathcal{R}$ with respect to a target message precision. For a positive integer q and a polynomial $\mathbf{a} \in \mathcal{R}$ (or $\mathcal{R}_Q$ for some $q \mid Q$), we let $[\mathbf{a}]_q$ be a representative of $\mathbf{a} \bmod q$ in $\mathcal{R}_q$. For positive integer r and Q, we denote $\mathrm{ord}_r Q$ as the largest integer k such that r^k divides Q.

The definition and properties of Number Theoretic Transform (NTT) can be found in Appendix B.1.

2.2 Computation in RNS-CKKS

RNS-CKKS scheme is an RNS variant of the CKKS scheme, first introduced in [CHK+19]. The ciphertext and the switching key modulus comprise NTT primes, constituting the RNS factors. Specifically, $Q_{\max} = q_0 \cdots q_L$ be the maximum ciphertext modulus, where q_i are relatively prime NTT primes. The ciphertext modulus is $Q = q_0 \cdots q_\ell$ for some $\ell \in [L+1]$. The switching key modulus is $PQ_{\max}$, where $P = p_0 \cdots p_{K-1}$, where p_j's are relatively prime NTT primes. In addition, q_i's and p_j's are relatively prime. The polynomials in $\mathcal{R}_Q$ are stored and computed in RNS, i.e., for $a \in \mathcal{R}_Q$, we indeed have $[\mathbf{a}]_{q_i} \in \mathcal{R}_{q_i}$ for every RNS factor $q_i | Q$. We note that the CRT decomposition is homomorphic.

We now recall some main features of the RNS-CKKS scheme. We include detailed explanations for some basic operations (Fast Basis Conversion (FBC), Inv-RS, ModUp, ModDown, and RS, and the level adjustment techniques) in Appendix B.2.

2.2.1 Gadget Decomposition and Key Switching. When the ring dimension N, the hamming weight of the secret key, and the target security are chosen, the maximum possible modulus $PQ_{\max}$ can be decided based on the estimated attack costs of the known attacks via Lattice estimator [APS15].

The switching key is generated in the largest modulus $PQ_{\max}$, possibly using gadget decomposition. For RNS gadget decomposition, for instance, each gadget Q_i is composed of some RNS factors, and the largest ciphertext modulus $Q_{\max}$ is composed into

$$\begin{aligned} Q_{\max} &= Q_0 \cdots Q_{\mathsf{dnum}-1} \\ &= (q_0 \cdots q_{\alpha-1}) \cdot (q_\alpha \cdots q_{2\alpha-1}) \cdots (q_{\alpha(\mathsf{dnum}-1)} \cdots q_{\mathsf{dnum}\cdot\alpha-1}), \end{aligned}$$

where $\alpha = \lceil (L+1)/\mathsf{dnum} \rceil$ for a pre-fixed parameter $\mathsf{dnum} \in \mathbb{N}$. Note that the relinearization key rlk is a special switching key that switches the secret key from $\mathbf{s}$ to $\mathbf{s}' = \mathbf{s}^2$. We call dnum the gadget rank. We choose P to satisfy $P \gtrsim Q_i$ for all i, so the maximum possible modulus $PQ_{\max}$ should be split into P and Q with roughly $P \approx Q^{1/\mathsf{dnum}}$.

Let $P = p_0 \cdots p_{K-1}$. Then, the switching keys are defined as

$$\{\mathsf{swk}_i\}_{i\in[\mathsf{dnum}]} = \{(\boldsymbol{\beta}_i, \boldsymbol{\alpha}_i)\}_{i\in[\mathsf{dnum}]} \in \mathcal{R}_{PQ}^{2\times\mathsf{dnum}}, \text{ where}$$
$$\boldsymbol{\beta}_i = -\boldsymbol{\alpha}_i \cdot \mathbf{s} + P \cdot \hat{Q}_i \cdot [\hat{Q}_i^{-1}]_{Q_i} \cdot \mathbf{s}' + \mathbf{e}_i \in \mathcal{R}_{PQ},$$

for $\hat{Q}_i = Q_{\max}/Q_i$, and $\mathbf{e}_i \leftarrow \chi$ be errors. We note that a larger dnum results in a larger usable ciphertext modulus $Q_{\max}$ and a slower key switching operation also with a larger switching key size.

Key switching. Key switching of a ciphertext $\mathsf{ct} \in \mathcal{R}_Q^2$, denoted as $\mathsf{KeySwitch}_{\mathsf{swk}}(\mathsf{ct})$, involves the following procedures.[3].

(1) Regard the ciphertext in modulus $Q_0 \cdots Q_{d-1}$ instead of $Q = q_0 \cdots q_\ell$, for some α and d satisfying $\alpha(d-1) \le \ell < \alpha d$.
(2) ModUp each components of the ciphertext in modulus Q_i, to $PQ_0 \cdots Q_{d-1}$ for $i \in [d]$, resulting in d ciphertexts in modulus $PQ_0 \cdots Q_{d-1}$.
(3) External product part of each ciphertext with swk_i for $i \in [d]$, and add the remaining parts of the ciphertext.
(4) ModDown from modulus $PQ_0 \cdots Q_{d-1}$ to Q.

[3]It requires $(d+2)(\ell+K+1)$ (i)NTT operations, and in particular, $(d+3+2/d)(L+1)$ (i)NTTs at the top level $\ell = L$ [MG23].

2.2.2 Homomorphic Multiplication. Homomorphic multiplication consists of the tensor product of two ciphertexts in the same modulus $Q = q_0 \cdots q_\ell$, then key switch s^2 to s via the rlk, then RS by q_ℓ. Note, the tensor of ciphertexts ct_1 and ct_2 satisfies $\langle \mathsf{ct}_1 \otimes \mathsf{ct}_2, (1, s, s^2) \rangle = \langle \mathsf{ct}_1, (1, s) \rangle \cdot \langle \mathsf{ct}_2, (1, s) \rangle$, where $\mathsf{sk} = (1, s)$.

2.3 CKKS Bootstrapping

As CKKS multiplication consumes the modulus by an amount of the scale factor sizes, CKKS bootstrapping is used to lengthen the multiplicative budget for further computations when the ciphertext modulus reaches the minimum. During bootstrapping, the ciphertext is first homomorphically decoded, then its modulus is raised, and the ciphertext is homomorphically encoded. Then, a reduction modulo, the base modulus, is applied homomorphically. Each sub-procedures are referred to as SlotToCoeff, ModRaise, CoeffToSlot, and EvalMod.

Basically, for given an input ciphertext in $\mathcal{R}_{q_0}^2$ encrypting a plaintext pt, ModRaise generates a ciphertext in $\mathcal{R}_{Q_{\max}}^2$ encrypting a plaintext $\mathsf{pt} + q_0\mathbf{I}$, requiring moduluo q_0 operation on the plaintext. The homomorphic Discrete Fourier Transforms (DFT) SlotToCoeff and CoeffToSlot appropriately switch the coefficient and the slot of the ciphertext so that the real-valued messages $\mathbf{m}$ can be mapped into $\mathbf{m} + q_0\mathbf{I}$ through the sequence of procedures SlotToCoeff, ModRaise, and CoeffToSlot. During EvalMod, a polynomial approximating the function $x \mapsto (x \mod q_0)$ is homomorphically evaluated. It is approximated as a sine function near the zeros, which is sometimes decomposed into several functions, namely the cosine, double angle formula, and arcsine functions with proper approximations [BTH22].

3 GRAFTING: FILLING-UP MACHINE WORDS IN RNS

All the computations in RNS-CKKS are done in the RNS format, i.e., every ciphertext or key is decomposed with respect to RNS factors, and each component is computed with the machine's word size. Therefore, if we can reduce the number of RNS components representing the ciphertext, the whole FHE computations will benefit from a straightforward speed-up by roughly the reduced ratio, depending on the ciphertext modulus.

An appealing approach is to use the word size primes for the RNS factors. However, it compromises the available multiplicative depths, compared to the cases when the moduli are set approximately the same as the scaling factors, say Δ, which varies (from 20 to 120 in general) on the target message precision.

In this section, we introduce *Grafting*, a method of using word-sized primes for RNS factors while allowing rescale by an independent factor. We first introduce a tool for switching the ciphertext modulus to an non-divisor modulus, namely, *Rational Rescale* in Section 3.1. We introduce a method to maintain the ciphertext modulus using word-sized NTT primes and a *universal sprout*, enabling optimal performance through *modulus resurrection*. This design supports rescaling to any scaling factor of arbitrary bit-length at any modulus, without incurring additional key generation costs compared to the original RNS-CKKS scheme. As detailed in

Section 3.2, this leads to a full decoupling between scale factors and moduli, fully realizing what was only partially addressed in prior works [AAB+23, MG23, SS24]. To support modulus switching for the universal sprout—central to achieving full decoupling—we extend the existing techniques to handle power-of-two factors in the RNS factor, as detailed in Section 3.3.

3.1 Rational Rescale: Rescale with non-divisor

In this section, we decouple the RNS factors from the scaling factors and propose a method to rescale a ciphertext from a modulus Q to another modulus Q', where Q' does not necessarily divide Q, which we call the *Rational Rescaling*. The rational rescale procedure rescales an input ciphertext in modulus Q by a rational number $Q/Q' \in \mathbb{Q}$ to an output ciphertext modulo $Q/(Q/Q') = Q'$, rescaling the scaling factor as well. We note that the same notion was introduced in [SS24], but without a formal definition or correctness proof. Concretely, we define the rational rescale operation and its correctness as follows.

Definition 3.1 (Rational Rescale). For given a polynomial $\mathbf{a} \in \mathcal{R}_Q$ and $Q' \nmid Q$, we define rational rescaling as the rescaling by a rational factor Q/Q', which can be computed as

$$\mathsf{RS}_{Q/Q'}(\mathbf{a}) = \mathsf{RS}_S(\mathsf{Inv\text{-}RS}_R(\mathbf{a}) \in \mathcal{R}_{\mathrm{lcm}(Q,Q')}) \in \mathcal{R}_{Q'},$$

where $R = \mathrm{lcm}(Q, Q')/Q \in \mathbb{Z}$ and $S = \mathrm{lcm}(Q, Q')/Q' \in \mathbb{Z}$.

Here, the operation RS_S in Definition 3.1 corresponds to the $\mathsf{ModDown}_{\mathrm{lcm}(Q,Q')\to Q'}$ in the original scheme, which rescales by multiple prime factors. In our grafted RNS-CKKS framework, we abuse the notation RS to encompass both the conventional *integer* rescaling—either by a single factor (i.e., the original RS) or multiple factors (i.e., ModDown)—and our extended notion of *rational* rescaling by arbitrary rational factors.

The following theorem shows that the error introduced by rational rescaling is of the same nature as the errors from ModDown and RS operations, i.e., it grows linearly with the number of eliminated RNS factors. The proof of the theorem can be found in Appendix C.2.

Theorem 3.2 (Rational Rescale Correctness). *For given a ciphertext* $\mathsf{ct} \in \mathcal{R}_Q^2$ *and* $Q' \nmid Q$*, it holds that* $\left[\langle \mathsf{RS}_{Q/Q'}(\mathsf{ct}), \mathsf{sk}\rangle\right]_{Q'} = (Q'/Q) \cdot [\langle \mathsf{ct}, \mathsf{sk}\rangle]_Q + e_{res}$ *for some rescale error* e_{res} *satisfying* $\|e_{res}\|_\infty \le \ell/2 \cdot (\|s\|_1 + 1)$*, where* ℓ *is the number of RNS blocks in* $\mathrm{lcm}(Q, Q')/Q'$*, and* s *be a secret key.*

Rational rescale maps a ciphertext including a message with a scaling factor Δ^2 (after tensor) to a ciphertext whose message has a scaling factor $\Delta^2 \cdot Q'/Q \approx \Delta$, instead of $\Delta^2/q_\ell \approx \Delta$. We describe homomorphic multiplication with rational rescale in Algorithm 1. We also note that the rational scaling factor can be tracked as in [KPP22].

We also propose modulus adjustment that adjusts two ciphertexts with different moduli and scaling factors before adding or multiplying them, as shown in Algorithm 2. We note that our modulus adjustment is a counterpart of the level adjustment in RNS-CKKS [KPP22], while introducing more flexibility by allowing one ciphertext to be adjusted to both the modulus and scaling factor of another. The detailed correctness proof can be found in Appendix C.3.

Algorithm 1: Homomorphic multiplication

Input: $\mathsf{ct}_i = (\mathbf{b}_i, \mathbf{a}_i; \Delta) \in \mathcal{R}_Q^2$ for $i = 0, 1$ for the modulus $Q = q_0 \cdots q_{\ell-1} \cdot r$, the scaling factor Δ, and the relinearization key $\mathsf{rlk} \in \mathcal{R}_{PQ_{\max}}^{2\cdot \mathsf{dnum}}$.

Output: $\mathsf{ct} = (\mathbf{b}, \mathbf{a}; \Delta') \in \mathcal{R}_{Q'}^2$ where $Q' = q_0 \cdots q_{\ell'-1} \cdot r'$ and $\Delta' = \Delta^2/(Q/Q')$.

1 $(\mathbf{b}, \mathbf{a}, \mathbf{d}) \leftarrow \mathsf{ct}_1 \otimes \mathsf{ct}_2$ ▷ $\mathbf{b} + \mathbf{as} + \mathbf{ds}^2 \approx \Delta^2 \mathbf{m}_1 \mathbf{m}_2$

2 $(\mathbf{b}', \mathbf{a}') \leftarrow \mathsf{KeySwitch}_{\mathsf{rlk}}((0, \mathbf{d}))$ ▷ $\mathbf{b}' + \mathbf{a}'\mathbf{s} \approx \mathbf{ds}^2$

3 $\mathsf{ct} \leftarrow (\mathbf{b}, \mathbf{a}) + (\mathbf{b}', \mathbf{a}')$

4 $\mathsf{ct} \leftarrow \mathsf{RS}_{Q/Q'}(\mathsf{ct})$ ▷ Rational rescale by $Q/Q' \approx \Delta$

return ct

Algorithm 2: Modulus adjustment.

Input: $\mathsf{ct} \in \mathcal{R}_Q^2$ with scaling factor Δ.

Output: $\mathsf{ct}' \in \mathcal{R}_{Q'}^2$ with scaling factor Δ', where $Q > Q'\Delta$.

1 Choose the smallest $Q_{\mathrm{mid}} \mid Q$ such that $Q \ge Q_{\mathrm{mid}} \gtrsim Q' \cdot \Delta$.

2 $\mathsf{ct} \leftarrow (\mathsf{ct} \mod Q_{\mathrm{mid}})$ ▷ Modulo reduction to Q_{mid}

3 $\mathsf{ct} \leftarrow \mathsf{ct} \cdot \lceil (Q_{\mathrm{mid}} \cdot \Delta')/(Q' \cdot \Delta) \rfloor$ ▷ Integer multiplication

4 $\mathsf{ct}' = \mathsf{RS}_{Q_{\mathrm{mid}}/Q'}(\mathsf{ct})$ ▷ Rational rescale to Q'

return ct'

Once we utilize rational rescaling to manage the scaling factor, however, the resulting ciphertext modulus Q' may not be a divisor of the switching key modulus $PQ_{\max}$. In this case, the original key-switching procedure does not apply to the ciphertext due to the changed modulus, which may not be compatible with the gadget decomposition.

For instance, highlighting the difficulty regarding key-switching, [SS24] introduces level-specific moduli, called *terminal residues*, which capture the possible bit size of the ciphertext modulus after rational rescaling for each ciphertext level. One might consider preparing keys for all possible ciphertext moduli to support key switching in this setting. However, this will increase the number of switching keys, which degrades the communication cost for key transmissions. When no specific circuits are predetermined, a huge number of keys need to be transmitted for general circuit evaluations. Having multiple copies is too much because the switching keys are already hundreds to thousands of megabytes for FHE parameters.[4]

3.2 Modulus Resurrection with Universal Sprouts

We propose the *Modulus Resurrection* technique, which enables the use of Rational Rescale without generating additional switching keys or incurring extra costs for key switching, by reusing factors of the ciphertext modulus. As a primary condition for the RNS factors is being relatively prime, we *resurrect* some factors of the top ciphertext modulus, which was scaled out previously. By doing so, the RNS factors remain relatively prime, while the ciphertext modulus is kept to be a divisor of the switching key modulus.

[4] See Table 5 for practical example. We also propose another solution that can be applied to this problem in Appendix C.1

The special resurrecting part of the ciphertext modulus is called *sprout*, which is flexible and possibly small. For all possible sprouts, we define a common multiple r_{top} of the sprouts, which we call the *top sprout*. In other words, we set a maximal sprout r_{top} and choose sprouts from its divisors. Each ciphertext modulus is a product of sprout and distinct word-sized NTT primes, which we call *unit moduli*. In this case, we call the ciphertext modulus *grafted with the sprout*.

The maximum ciphertext modulus is $Q_{\max} = q_0 \cdots q_{L-1} \cdot r_{\text{top}}$ where q_i's be the unit moduli approximately of the machine word size $w = 2^\omega$. Each ciphertext modulus is $Q = q_0 \cdots q_{\ell-1} \cdot r$ for some sprout r. The switching key modulus is $PQ_{\max}$, where P is set conventionally, i.e., $P = p_0 \cdots p_{K-1}$ for relatively prime unit moduli p_i's with $\gcd(P, Q_{\max}) = 1$. As each sprout r is a divisor of r_{top}, it satisfies $r \mid r_{\text{top}}$ and thus $Q \mid Q_{\max} \mid PQ_{\max}$. Hence, we can key switch using the switching keys in modulus $PQ_{\max}$, as many as in the conventional key-switching.

We describe how to construct the sprout r_{top} in an optimal way, in the sense that for any ciphertext modulus Q with an arbitrary integer bit-length, one can always find $\ell \in [L]$ and $r \mid r_{\text{top}}$ such that $Q \approx q_0 \cdots q_{\ell-1} \cdot r$. Then, for any modulus $Q \mid Q_{\max}$ and scaling factor Δ of arbitrary bit-length, we can find $Q' \mid Q_{\max}$ such that $Q' \approx Q/\Delta$. We refer to this property as *universality* and call such a sprout a *universal sprout*, formally defined as follows:

Theorem 3.3 (Universal Rescalability). *Let the ciphertext modulus $q_i \in [2^\omega(1-\eta), 2^\omega(1+\eta)]$ for some $\eta > 0$ for $i \in [L-1]$. Let the maximum sprout modulus $r_{top} = r_0 \cdots r_s$. Assume for any positive integer $\gamma \le w$, there exist $r | r_{top}$ such that $r \in 2^\gamma \cdot [1-\epsilon, 1+\epsilon]$ for some $\epsilon > 0$. Then, for any ciphertext in any possible ciphertext modulus Q, one can (rational) rescale by $2^\delta \cdot \left(1 \pm (n\eta + 2\epsilon) + O\left(\eta^2 + \epsilon^2\right)\right)$ for any positive integer $\delta < \log_2 Q$, where $n = \lceil \delta/w \rceil$.*

We refer to Appendix C.4 for the proof. A simple example of a universal sprout is $r_{\text{top}} = 2^\omega$, as illustrated below:

Example 3.1 (`sprout-60`). *Let q_i's be the 60-bit [5] NTT primes and $r_{top} = 2^{60}$. Each sprout r is a power of two integers dividing $r_{top} = 2^{60}$. Thus, any ciphertext modulus of approximately an integer bit can be represented:*

$$Q = q_0 \cdots q_{\ell-1} \cdot 2^\gamma \approx (60 \cdot \ell + \gamma)\text{-bit modulus},$$

where the top modulus is $Q_{max} = q_0 \cdots q_{L-1} \cdot 2^{60}$. From a ciphertext modulus of approximately $(60 \cdot \ell + \gamma)$ bits, one can rational rescale by $q_{\ell'} \cdots q_{\ell-1} \cdot 2^{(\gamma-\gamma')}$ to obtain a ciphertext modulus of approximately $(60 \cdot \ell' + \gamma')$ bits, regardless of whether $\gamma \ge \gamma'$.

For instance, if we want to rescale by $\approx$ 36 bits from a modulus $Q = q_0 \cdots q_{\ell-1} \cdot 2^{35}$, we can rational rescale by $\left(q_{\ell-1}/2^{24}\right) \approx 2^{36}$, resulting in a ciphertext modulus of $Q' = q_0 \cdots q_{\ell-2} \cdot 2^{59}$.

As in the above example, the top sprout r_{top} is universal if and only if its divisors can approximately represent all the bit lengths from 1 to ω. A universal sprout enables rescaling by nearly any bit-length, allowing it to be used universally, regardless of the target circuit.

[5] We take 60-62 bits for primes instead of 64 bits for some efficiency reasons even in the 64-bit machines, e.g., due to key switching with dnum > 1 or lazy rescale.

We note that efficiently handling a ciphertext modulo 2^{60}, however, is difficult on a 64-bit machine. It requires embedding $\mathcal{R}_{2^{60}}$ into a larger modulus ring $\mathcal{R}_B$, where B must satisfy $B > N \cdot 2^{120}$ to prevent reductions modulo B during multiplication.[6] In practice, such a B involves at least three word-sized moduli, making the computation roughly three times more expensive than using a single 60-bit NTT prime. Instead, we propose another universal sprout in Example 3.2, whose handling cost is roughly twice that of a word-sized NTT prime.

Example 3.2 (`sprout-15-16-30`). *Let q_i's be 61-bit NTT primes and $r_{top} = 2^{15} \cdot r_1 \cdot r_2$, where r_1 is a 16-bit NTT prime, and r_2 is a 30-bit NTT prime. Typically, we can choose $r_1 = 2^{16}+1$ and $r_2 = 2^{30} - 2^{18} + 1$, which are NTT primes for ring dimension $N \le 2^{15}$.[7] Each sprout r can be represented as $r = 2^\alpha \cdot r_1^{\beta_1} \cdot r_2^{\beta_2}$, where $0 \le \alpha \le 15$, $\beta_i \in \{0, 1\}$. We note that the sprouts can represent any bit length from 1 to 61 as*

$$\{2^1, \cdots, 2^{15}, r_1, r_1 \cdot 2^1, \cdots, r_1 \cdot 2^{13}, r_2, r_2 \cdot 2^1, \cdots, r_2 \cdot r_1 \cdot 2^{15}\}.$$

Hence, it forms a universal sprout that supports rational rescale with arbitrary bit-length.

For instance, from a modulus $Q = q_0 \cdots q_{\ell-1} \cdot r$, where $r = 2^{13} \cdot r_2 \approx 2^{43}$, let assume we want to rescale by $\approx$ 15 bits. We can rational rescale by $(r_2/2)$, resulting in a ciphertext modulus $Q' = q_0 \cdots q_{\ell-1} \cdot r'$, where $r' = \left(2^{13} \cdot r_2\right)/(r_2/2) \approx 2^{14}$. If we want to rescale by 34 bits in addition, we can rational rescale by $\left(2^3 \cdot q_{\ell-1}/r_2\right)$, resulting in a ciphertext modulus $Q'' = q_0 \cdots q_{\ell-2} \cdot r''$, where $r'' = \left(q_\ell \cdot 2^{14}\right)/\left(2^3 \cdot q_\ell/r_2\right) = 2^{11} \cdot r_2 \approx 2^{41}$. Note that a larger amount of rational rescale can be done as well.

In Example 3.2, arithmetic over both $\mathcal{R}_{2^{15}}$ and $\mathcal{R}_{r_1 r_2}$ can be efficiently performed within a single word-sized modulus. In the latter case, this is achieved using composite NTT [CHK+21]. We refer the reader to Appendix C.6 for details on optimized arithmetic over universal sprouts, and to Appendix C.5 for additional candidate `sprout-19-20-23`.

Once a universal sprout is chosen, Algorithm 3 describes how to derive an output modulus $Q' \approx Q/\Delta$ from the input modulus Q and the rescaling factor Δ via rational rescaling. In the algorithm, the universality condition ensures that a suitable sprout r' can be found for Q'.

Algorithm 3: Selecting grafted ciphertext moduli after rational rescale.

Input: $Q = q_0 \cdots q_{\ell-1} \cdot r$ for input ciphertext with scaling factor Δ and word size $w = 2^\omega$.
Output: $Q' = q_0 \cdots q_{\ell'-1} \cdot r'$ for output ciphertext with $Q' \approx Q/\Delta$.

1 Choose $r' \mid r_{\text{top}}$ such that $|\log_2 r' - (\log_2(Q/\Delta) \bmod \omega)|$ achieves the minimum.
2 $\ell' \leftarrow \left\lfloor (\omega \cdot \ell + \lfloor \log_2 Q - \log_2 \Delta \rceil)/\omega \right\rfloor$
return $Q' = q_0 \cdots q_{\ell'-1} \cdot r'$

[6] One may consider using a real/complex-valued Discrete Fourier Transform (DFT) instead of the NTT. However, it introduces implementation difficulties due to the need for higher precision than the native word size.
[7] Even for larger dimensions, we can use them by utilizing incomplete NTTs instead of complete NTTs.

Using Algorithm 3, one can always select an output modulus $Q' \approx Q/\Delta$ in Algorithm 1 as a grafted modulus dividing $Q_{\max}$, thereby enabling key switching from ciphertexts in arbitrary moduli during homomorphic computations. We also note that only a few unit moduli are eliminated during rational rescaling, and the resulting error remains roughly of the same magnitude as that of standard rescaling.

Now, we describe the overall process of the key switching operation, as well as the relinearization, in the Grafted RNS-CKKS. We decompose the largest ciphertext modulus $Q_{\max}$ into dnum number of similar-sized blocks $Q_0, \cdots, Q_{\mathsf{dnum}-1}$, known as the gadget blocks, where $P \gtrsim \max_i Q_i$. Although the sprout can be placed in any gadget block, the gadget block containing the sprout must always participate in key switching. Therefore, we place the sprout in the bottom gadget block Q_0 so that key switching under the grafted moduli incurs no computational compromise compared to the original key switching procedure. The gadget blocks are defined as:

$$\begin{aligned} Q_0 &= r_{\text{top}} \cdot q_0 \cdots q_{\alpha-2}, \\ Q_i &= q_{\alpha i-1} q_{\alpha i} \cdots q_{\alpha(i+1)-2} \text{ for } 1 \le i \le \mathsf{dnum} - 1, \end{aligned}$$

where $L + 1 = \mathsf{dnum} \cdot \alpha$. Then, a grafted ciphertext modulus with sprouts can be represented as

$$\begin{aligned} Q = r \cdot q_0 q_1 \cdots q_{\ell-1} &= (r \cdot q_0 \cdots q_{\alpha-2}) \cdot (q_{\alpha-1} \cdots q_{2\alpha-2}) \\ &\quad \cdots (q_{\alpha(d-1)-1} \cdots q_{\ell-1}) \\ &= Q'_0 \cdot Q_1 \cdots Q_{d-2} \cdot Q'_{d-1}, \end{aligned}$$

for some $1 \le \ell \le L$, $Q'_0 \mid Q_0$, and $Q'_{d-1} \mid Q_{d-1}$ where $d = \lceil (\ell+1)/\alpha \rceil$. This choice clearly minimizes the number of gadgets for each ciphertext modulus Q.

Following the original RNS-CKKS key-switching procedures described in Section 2.2.1, we present its extension to grafted ciphertext moduli in Algorithm 4.

Algorithm 4: Key switching in grafted moduli.

Input: $\mathsf{ct} = (\mathbf{b}, \mathbf{a}) \in \mathcal{R}_Q^2$ for $Q = Q'_0 \cdot Q_1 \cdots Q_{d-2} \cdot Q'_{d-1}$ with a switching key $\mathsf{swk} = \{\mathsf{swk}_i\} \in \mathcal{R}_{PQ_{\max}}^{2\times d}$.

Output: $\mathsf{ct}' \in \mathcal{R}_Q^2$ with a switched secret key.

$\mathbf{a} \leftarrow \mathsf{Inv\text{-}RS}_{Q_{\text{inter}}/Q}(\mathbf{a})$ ▷ Inv-RS to $Q_{\text{inter}} = Q_0 \cdots Q_{d-1}$

$\mathbf{a}_i \leftarrow \mathsf{ModUp}_{Q_i \to PQ_{\text{inter}}}(\mathbf{a} \mod Q_i)$ for $i \in [d]$;

$\mathsf{ct} \leftarrow \sum_i \mathbf{a}_i \cdot (\mathsf{swk}_i \mod PQ_{\text{inter}})$;

$\mathsf{ct}' \leftarrow \mathsf{RS}_{PQ_{\text{inter}}/Q}(\mathsf{ct})$ ▷ RS to Q

$\mathsf{ct}' \leftarrow \mathsf{ct}' + (\mathbf{b}, 0)$;

return ct'

3.3 Modulo Arithmetic with Power-of-two Sprouts

The correctness of multiplication in Algorithm 1 and key switching in Algorithm 4 under grafted ciphertext moduli follows directly from the correctness of modulus switching, including rational rescale, ModUp, and ModDown processes. However, when a power-of-two modulus in the universal sprout r_{top} is involved, a new type of issue arises during modulus switching. The original RNS-CKKS scheme does not cover such cases, as modulus switching was originally defined only between relatively prime bases.

In this section, we extend modulus switching algorithms so that grafted RNS-CKKS supports RNS factors with power-of-two factors. We begin by identifying cases where such extensions apply naturally.

If the exponent of the power-of-two component remains unchanged between the input and output moduli, then all CRT factors—including the power-of-two modulus—remain pairwise relatively prime. In this case, the fast basis conversion (FBC) and its applications—RS (by an integer factor), Inv-RS, and ModUp —can be used without modification, as in the original RNS-CKKS.

When the exponent differs, we handle the switching in two stages using an intermediate modulus Q_{inter}. First, we apply standard modulus switching from Q to Q_{inter}, ensuring that their power-of-two exponents are equal. Then, we use Inv-RS or RS from Q_{inter} to Q' to adjust the exponent. This order can also be reversed.

This two-step approach introduces negligible overhead, as shown in Appendix B.2.2, and often improves error control by separating exponent adjustment from standard switching.

Accordingly, we present new algorithms for Inv-RS and RS (including ModDown) to support such cases, as shown in Algorithms 5 and 6.

Algorithm 5: Inv-RS with power-of-two sprouts.

Input: $\mathbf{a} \in \mathcal{R}_Q$ for $Q = \tilde{Q} \cdot 2^\beta$ and $R = \tilde{R} \cdot 2^\gamma$, where $\tilde{Q}$ and $\tilde{R}$ are odd integers and $\beta, \gamma \in \mathbb{Z}_{\ge 0}$.

Output: $\mathsf{Inv\text{-}RS}_R(\mathbf{a}) \in \mathcal{R}_{QR}$.

$\mathbf{b} \leftarrow \begin{cases} [2^\gamma]_{q_i} \cdot [\mathbf{a}]_{q_i} & (\mod q_i) \text{ for } i \in [\ell], \\ 2^\gamma \cdot [\mathbf{a}]_{2^\beta} & (\mod 2^{\beta+\gamma}), \end{cases}$ ▷ $\mathbf{b} \in \mathcal{R}_{\tilde{Q} \cdot 2^{\beta+\gamma}}$

$\mathbf{c} \leftarrow \mathsf{Inv\text{-}RS}_{\tilde{R}}(\mathbf{b})$ ▷ $\mathbf{c} \in \mathcal{R}_{QR}$

return $\mathbf{c}$

In Algorithm 5, the intermediate polynomial satisfies $\mathbf{b} \equiv 2^\gamma \cdot \mathbf{a} \pmod{\tilde{Q} \cdot 2^{\beta+\gamma}}$. Then, the Inv-RS with the integer factor $\tilde{R}$ can be applied since $\tilde{R}$ is an odd integer, resulting in an output $\mathbf{c} \equiv \tilde{R} \cdot (2^\gamma \cdot \mathbf{a}) \equiv R \cdot \mathbf{a} \pmod{QR}$ as desired.

Algorithm 6: RS with power-of-two sprouts.

Input: $\mathbf{a} \in \mathcal{R}_{QR}$ for $Q = \tilde{Q} \cdot 2^\beta$ and $R = \tilde{R} \cdot 2^\gamma$, where $\tilde{Q}$ and $\tilde{R}$ are odd integers and $\beta, \gamma \in \mathbb{Z}_{\ge} 0$.

Output: $\mathsf{RS}_R(\mathbf{a}) \in \mathcal{R}_Q$.

$\mathbf{b} \leftarrow \mathsf{ModDown}_{QR \to \tilde{Q} \cdot 2^{\beta+\gamma}}(\mathbf{a})$ ▷ $\mathbf{b} \in \mathcal{R}_{\tilde{Q} \cdot 2^{\beta+\gamma}}$

$\mathbf{c} \leftarrow \begin{cases} [2^{-\gamma}]_{q_i} \left([\mathbf{b}]_{q_i} - [\mathbf{b}]_{2^\gamma} \right) & (\mod q_i), \\ \left([\mathbf{b}]_{2^{\beta+\gamma}} - [\mathbf{b}]_{2^\gamma} \right) / 2^\gamma & (\mod 2^\beta), \end{cases}$ ▷ $\mathbf{c} \in \mathcal{R}_Q$

return $\mathbf{c}$

In Algorithm 6, the intermediate polynomial $\mathbf{b}$ satisfies $\|\mathbf{b} - \tilde{R}^{-1} \cdot \mathbf{a}\|_\infty \le k/2$, where k is the number of decomposed factors of $\tilde{R}$. We note that the last division for modulo 2^β can be replaced by shifting γ bits. The final output $\mathbf{c}$ satisfies $\mathbf{c} = (\mathbf{b} - [\mathbf{b}]_{2^\gamma})/2^\gamma$, leading to the

bound $\|\mathbf{c} - 2^{-\gamma} \cdot \mathbf{b}\|_\infty \le 1/2$. This implies the overall error satisfies $\|\mathbf{c} - R^{-1} \cdot \mathbf{a}\|_\infty \le 1/2 + 2^{-\gamma} \cdot k/2$, as desired.

Lastly, we note that ModUp is used only within the key switching operation, where the power-of-two component is already adjusted by the preceding Inv-RS (line 1) of Algorithm 4. As a result, ModUp (line 2) can be applied without modification.

With the above extensions in place, we can now apply the key switching in Algorithm 4 and the homomorphic multiplication in Algorithm 1 directly under the universal sprout setting. We remark that the key switching procedure can be slightly modified to omit the Inv-RS by the power-of-two factors. That is, key switching in Algorithm 4 for the power-of-two modulus part can be performed modulo $2^{\mathrm{ord}_2(Q)}$ by reducing the modulus, rather than modulo $2^{\mathrm{ord}_2(Q_{\max})}$ with an additional Inv-RS.

4 APPLICATIONS

Grafting optimally packs the modulus using mostly machine word-sized factors, resulting in fewer RNS components. This reduction not only accelerates homomorphic operations but also decreases the ciphertext and public key sizes, roughly in proportion to the reduced number of factors. Also, when using Grafting, we do not need to suffer from the lack of small NTT primes or scale factors larger than the machine's word size, which naturally allows lower—and/or higher-precision computation. In addition, Grafting is particularly effective in applications that use smaller-than-usual scale factors where the original number of RNS factors is far from optimal.

In this section, we revisit the CKKS applications with Grafting. In Section 4.1, we first suggest grafted parameters for the standard RNS-CKKS parameters for SHE and FHE, showcasing the expected speedups gained from Grafting. Then, in Section 4.2, we revisit the Bit-CKKS gate bootstrappings of Bae et al. [BCKS24] with Grafting, which benefit from the small-sized scale factors $\approx$ 28-30 bits. Finally, in Section 4.3, we revisit homomorphic comparison [CKK20] using Grafting as an example of Newton-like methods accelerated via adaptively chosen scale factors.

4.1 Grafted SHE and FHE Parameters

We revisit various CKKS parameters of SHE and FHE in the literature standard, specifically from the white paper of Bossuat et al. [BCC$^+$24], which specifies the guidelines for securely implementing HE schemes and the libraries such as HEaaN [Cry22] and OpenFHE [ABBB$^+$22]. In Table 2, we summarize the details of the parameter sets, including SHE parameters in $\log_2 N = 14$ and 15 and FHE parameters in $\log_2 N = 15$ and 16. The security guideline white paper introduces several parameters basically based on Lattigo [lat24] and OpenFHE, but with full ternary secret keys, meaning that the coefficients of the secret keys are randomly sampled from $\{-1, 0, 1\}$. The security parameters are set as $\lambda \gtrsim 128$ for all the parameter sets, and a parameter set from [BCC$^+$24] has $\lambda \approx 192$. The sizes of the RNS factors consisting of the ciphertext modulus are shown for each subprocedure as $X \times Y$, implying that it can be decomposed into Y RNS factors of size X bits each. The total number of the RNS factors (denoted as #) and the rank for gadget decompositions (denoted as d) are given in advance. We note that for simSHE15s, the larger than word-size moduli are used for the RNS factors requiring multiple-precision integer arithmetic [ABBB$^+$22]. We, instead, decompose them into factors smaller than word-size and use composite rescaling [AAB$^+$23], which shows better performance in general.

In the table, we also introduce their adaptation to Grafting using the universal sprouts `sprout-15-16-30` and `sprout-19-20-23`. The maximum modulus (denoted as $\log_2 PQ$) and the modulus for multiplication (shown in parenthesis for grafted parameters) in the grafted adaptation are maintained similarly to the original parameters. As an exception, the FHE parameter for $\log_2 N = 16$ has a much smaller maximum modulus than the original parameter but offers more available modulus for multiplications between the two consecutive bootstrappings. This is because the moduli for ciphertexts and for switching keys can be adjusted in a more efficient way.

Speed-up Expectations. The speed-up when using Grafting compared to the prior state-of-the-art can be expected via the ratio of the number of RNS factors. The run-time of the tensor product at level L is proportional to the number of RNS factors, $L + 1$. The run-time of the key-switching can be roughly expected as $(\mathrm{dnum} + 2)(L + K + 1)$ for K, the number of RNS factors for temporary modulus (P) and the gadget rank dnum, when focusing on the number of NTT/iNTT operations. The speed-ups for homomorphic multiplications are expected to be similar to those of the key-switchings, as they are the dominating part of the multiplications. However, when the overall speed is very fast, e.g., at the bottom level, the ratio may differ due to slightly more expensive rescaling in Grafting. For instance, when comparing simSHE14 and graSHE14, we expect $8/7 = 1.14\times$ speed-up for tensor product and $(6 \cdot 10)/(5 \cdot 9) = 1.33\times$ speed-up for key-switchings at level $L = 7$.

For SlotToCoeff, CoeffToSlot, and EvalMod in FHE parameters, the run-time is significantly affected by the key-switching and multiplication timings. They will have a similar ratio to the key-switchings. However, as the levels are changing during the procedures, it is not easy to expect the gains. We roughly estimate the speed-up as the weighted mean of the ratios for SlotToCoeff, CoeffToSlot, and EvalMod, where each ratio is estimated as the key-switching ratio at the middle levels. For instance, when comparing simFHE15 and graFHE15, we expect $(12 \cdot 21)/(8 \cdot 14) = 2.25\times$ speed-up for key-switchings at level $L = 18$, $(9 \cdot 15)/(7 \cdot 12) = 1.61\times$ at level $L = 13$, and $(4 \cdot 6)/(4 \cdot 6) = 1\times$ at level $L = 2$. Assuming 50% of CoeffToSlot, 40% of EvalMod, and 10% of SlotToCoeff for the total timings, we can expect the bootstrapping speed-up as $1.87\times$. We note that the ratio can differ in the levels, and we expect almost no gain in the bottom levels.

4.2 Bit-CKKS, Revisited

The binary bootstrapping of Bae et al. [BCKS24] assumes a ciphertext encrypts only a bit in each slot. Thus, the scale factors can be smaller than usual. Specifically, the BinBoot algorithm in [BCKS24] assumes each complex message $b+\epsilon$ for a bit $b \in \{0, 1\}$ with an error $|\epsilon| \ll 1$ and a fine-grained scale factor $q_0/2$. The EvalMod algorithm is then adapted to approximate $(b + \epsilon)/2 + I \mapsto b + o(\epsilon^2)$ via

$$f_{\mathsf{BinBoot}}(x) = \frac{1 - \cos(2\pi x)}{2},$$

Table 2: Various HE parameters for CKKS from the security guidelines white paper [BCC+24], the HEaaN library [Cry22], and the OpenFHE library [ABBB+22], along with their grafted adaptations. All the parameters have a security parameter of $\lambda \gtrsim 128$, and specifically for simple/graSHE15, $\lambda = 192$**. The last** s **indicates using sparse ternary secret.**

Parameters		N	$\log PQ$	h	$\log q_i$					$\log p_i$	#	d
					Base	StC	Mult	EvalMod	CtS			
simSHE14	[BCC+24]	2^{14}	427	full	40	38×7				60×2	10	4
graSHE14					51×6					60×2	8	3
simSHE15	[BCC+24]	2^{15}	592	full	43	41×9				60×3	13	4
graSHE15					$58 + 59 \times 6$					60×3	10	3
simSHE15s	[ABBB+22]	2^{15}	675	192	$105\ (52 + 53)$	$90 \times 5\ (45 \times 10)$				60×2	14	6
graSHE15s			679		$61 \times 3 + 62 \times 6$					62×2	11	5
simFHE15s	[Cry22]	2^{15}	777	192	38	$32 + 28 \times 2$	28×5	38×8	41×3	42×2	22	10
graFHE15s			780		$38 + 61 \times 5 + 62 \times 5 + 4^*$ (137)					$61 + 62$	13	6
simFHE16	[BCC+24]	2^{16}	1734	full	45	30×3	35×10	60×12	56×4	61×5	35	6
graFHE16					$59 \times 11 + 60 \times 13$ (361)					61×5	29	5
simFHE16s	[Cry22]	2^{16}	1555	192	58	42×3	42×9	58×9	58×3	$59 \times 3 + 60 \times 2$	30	5
graFHE16s					$36 + 58 \times 20 + 62$ (378)					$59 \times 3 + 60 \times 2$	27	5

so that $f_{\mathsf{BinBoot}}((b+\epsilon)/2+I) = (1-\cos(\pi b+\pi\epsilon))/2 = b+o(\epsilon^2)$ for $b \in \{0,1\}$ and $|\epsilon| \ll 1$.[8] Similar approaches apply to boolean gates, e.g., $f_{\mathsf{nandBoot}}(x_0, x_1) = 2(1+\cos(2\pi(x_0+x_1)+\pi/6))/3$, sharing most of the parameters.

In Table 3, we provide the original and grafted variants of the Bit-CKKS [BCKS24] parameters utilizing `sprout-15-16-30` and `sprout-19-20-23` for ring dimensions $N = 2^{14}$ and 2^{16}, respectively. All the parameters use sparse secret encapsulation [HS21, BTH22] between a normal secret key with a Hamming weight h and a sparser secret key with a Hamming weight h', to avoid costly EvalMod and to lower the failure probability. The grafted adaptation follows the same strategy as in Table 2. As an exception, we could reduce the maximum modulus and the Hamming weight h for $N = 2^{16}$ while keeping the security parameter $\lambda \gtrsim 128$.[9]

In addition to the parameters in the prior art, we suggest another parameter set for $\log_2 N = 14$, namely graBinFHE14Opt, with optimizations for more available levels after bootstrapping: 3 available levels instead of 2, which was unavailable due to the lack of the NTT primes, but enabled by Grafting.

4.3 Homomorphic Comparison, Revisited

Iterative methods for finding a specific real number, such as Newton's iterative methods, benefited from adaptive-precision computation in cleartext. Its encrypted counterpart can be the homomorphic computation with adaptive scale factors, changing corresponding to the iterations.

This also applies to homomorphic comparison methods [CKK20, LLNK22, LLKN22], which is basically an iterative homomorphic evaluation of low-degree polynomials. The comparisons are frequently required for applications, such as machine learning, but which consume a lot of moduli, even requiring a few numbers of bootstrappings.

The homomorphic comparison function proposed in [CKK20] is constructed of two polynomials, f and g. The composited polynomial maps $[-1,-\epsilon]$ and $[\epsilon,1]$ to $[-1,-1+2^{-\alpha}]$ and $[1-2^{\alpha},1]$, respectively, which approximates the sign function in the range $[-1,1]$. In more detail, composing g multiple times maps $[\epsilon,1]$ to $[p,1]$, where $p \approx 0.8$ is widely chosen. The more g is composed, the less ϵ becomes. After that, $[p,1]$ is mapped into $[1-2^{-\alpha},1]$ by composing f multiple times. The more f is composed, the larger α becomes.

Regarding the homomorphic computation errors, indeed, composing g multiple times maps $[\epsilon,1]$ to the interval $[p-e_g,1]$, where e_g is the error introduced during composing g multiple times. During computing f, the error e_g can vanish. With this heuristic property, we can adjust the scaling factors per step using grafting while maintaining the final precision.

5 EXPERIMENTAL RESULTS

We implement Grafting using C++ for the non-grafted (simple) and grafted variant of RNS-CKKS. In this section, we report and compare execution times, ciphertext and key sizes, and the bootstrapping precision of our simple and grafted RNS-CKKS implementations, following the previous section's parameters.

We run our experiments on Ubuntu 24.04.1 LTS with an AMD Ryzen 7 3700X CPU and 32 GB of available memory. We disable CPU scaling, use only a single thread, and pin execution to one CPU core.

In the following, we define the *precision* of each experiment as the negative base-2 logarithm of the maximum error obtained from over 100 runs. Note that in [BCC+24], the precision is defined differently–the average of the negative base-2 logarithm of the error, i.e., $-\log_2|\text{error}|$ averaged over the slots and executions. We will refer this measure to *L1-precision*, mainly to compare with the precision given in [BCC+24]. The timings are averaged from over 100 runs.

[8] Since $\cos(\pi\epsilon) = 1+o(\epsilon^2)$ and $1+\cos(\pi+\pi\epsilon) = 2\sin^2(\pi\epsilon/2) = o(\epsilon^2)$.
[9] Estimated for grafted variants via lattice estimator [APS15].

Table 3: Parameters for Bit-CKKS [BCKS24] and their grafted variants with security parameter $\lambda \gtrsim 128$. Note that for the additional parameter graBinFHE14Opt**, the modulus consumption for each sub-procedure is given in advance.**

Parameters		N	$\log PQ$	h (h')	$\log q_i$					$\log p_i$	#	dnum
					Base	StC	Mult	EvalMod	CtS			
simBinFHE14	[BCKS24]	2^{14}	424	256 (32)	32	$28(=14+14)$	26×2	32×7	29×2	33	14	13
graBinFHE14			426		$28+56\times 5+62$ (52)					56	8	7
graBinFHE14Opt	Ours	2^{14}	424	256 (32)	26	$24(=12+12)$	24×3	28×7	25×2	56	8	7
					$26+56\times 5+62$ (72)							
simBinFHE16	[BCKS24]	2^{16}	1598	256 (32)	32	$32(=16+16)$	30×28	32×7	32×2	58×7	46	3
graBinFHE16			1522	192 (32)	$34+62\times 19$ (840)					62×5	25	4

5.1 SHE and FHE with Grafting

We display the experimental results for SHE and FHE operations in Table 4 based on the parameter designed in Table 2. The timings for tensoring, key-switching, homomorphic multiplications, and bootstrappings are shown in seconds, and the ratios between the simple and the grafted variants are given in the parenthesis. The timings are measured at the top levels for evaluations, i.e., top levels after bootstrapping for the FHE parameters. Our experiments report up to 1.58×, 1.83×, 2.01× speed-ups for homomorphic evaluations for available levels after bootstrapping, and up to 1.92× speed-up for bootstrapping. Note that for sim/graFHE16s, the speed-up is low as expected–due to the larger scale factors. The experiments reported precisions at least higher than or similar to the non-grafted cases, for instance, multiplication precision of 22.8 bits (L1-precision of 26.2 bits) for graSHE14, and bootstrapping precision for graFHE16 of 13.2 bits. Note that the L1-precision is similar to or larger than that given in [BCC+24].

Table 4: Execution timings for SHE and FHE operations, given in milliseconds. The bootstrapping (BTS**) timings are given in seconds. Note that** SHE **and** FHE **parameters are abbreviated as** S **and** F**, respectively. The speed-up is given in the parenthesis.**

Params	Tensor	KeySwitch	Mult	BTS
simS14	1.94	32.22	41.61	-
graS14	1.51 (1.3×)	24.16 (1.3×)	36.52 (1.1×)	-
simS15	4.55	93.39	118.86	-
graS15	3.53 (1.3×)	65.61 (1.4×)	91.95 (1.3×)	-
simS15s	5.56	127.38	171.64	-
graS15s	4.74 (1.2×)	93.42 (1.4×)	128.44 (1.3×)	-
simF15s	4.2	69.16	102.2	14.5
graF15s	2.78 (1.5×)	40.29 (1.7×)	57.28 (1.8×)	7.6 (1.9×)
simF16	12.96	248.47	360.84	86.5
graF16	8.21 (1.6×)	136.07 (1.8×)	179.87 (2.0×)	71.7 (1.2×)
simF16s	12.46	227.68	329.38	37
graF16s	9.88 (1.3×)	186.81 (1.2×)	247.45 (1.3×)	35.5 (1.1×)

In addition to the performance gain in the execution time, we also present the sizes of the public keys under various parameters in the basic RNS-CKKS case, in BitPacker [SS24], and Grafting. The sizes are the same for every key-switching key, we focus on the relinearization key and estimate its sizes in Table 2, required for each level for multiplication. For better performance, each key component is in CRT format during key switching. We note that in simple and grafted, the keys are naturally packed in the machine's word size, and the size of each key is already near optimal.[10] However, in BitPacker, the number of required keys is different. In general, each level requires keys in the different moduli, which should be generated independently for each level.

In Table 5, we compare each parameter set's relinearization key sizes to allow multiplications at each level. The simple and grafted cases are assumed to generate only one key at modulus $PQ_{\max}$ for whole levels, which can be reused. They differ by the ratio between the number of primes consisting of the modulus $PQ_{\max}$. In the case of BitPacker [SS24], we assume one key for each level (except for the SlotToCoeff and CoeffToSlot levels), but the gadget decompositions are optimized for better sizes. Note the key sizes are computed based on the levels of the corresponding simple SHE/FHE parameters. Thus, if one wants to use ciphertext modulus other than the levels at the simple parameter, the key sizes will increase for BitPacker but not for Grafting.

Table 5: Size estimations for the relinearization key in SHE and FHE parameters, in MB. The changes in the key sizes in percentile for each parameter over the simple parameters are given in parentheses.

Parameters	simple	BitPacker [SS24]	grafted
SHE14	10.49	29.88 (185%↑)	7.08 (32%↓)
SHE15	27.26	88.60 (225%↑)	17.30 (37%↓)
SHE15s	44.04	97.00 (122%↑)	31.46 (29%↓)
FHE15	115.34	313.00 (171%↑)	44.04 (62%↓)
FHE16	220.20	1394.61 (533%↑)	157.29 (29%↓)
FHE16s	157.29	981.47 (524%↑)	146.80 (7%↓)

5.2 Bit-CKKS with Grafting

We display the experimental results for Bit-CKKS parameters in Table 6, based on the parameter designed in Table 3. The message space for the input messages is binary, $\{0,1\}$, and the

[10] For transmission, half of the key's polynomials (say, a-parts) can be replaced by the seed when stored and transformed on the fly via an extendable output function (XOF).

messages are randomly chosen. We use real slots for $\log_2 N = 14$ parameters (i.e., 2^{13} binary messages) and full complex slots for $\log_2 N = 16$ parameters (i.e., 2^{16} binary messages), as in [BCKS24]. Our experiments report 1.81-1.89× speed-up while reporting the moderate precisions of 6.6 and 6.1 bits for sim/graBinFHE14 and sim/graBinFHE16, respectively. Note that for the additional parameter set with optimized available levels, 3.5 bits of precision is reported. The precisions are again similar for both non-grafted and grafted variants. We additionally report the multiplication timings with speed-ups of up to 2.07×. For $\log_2 N = 14$, the measurement was done at the near-bottom modulus; it shows the same as timings. The relinearization key sizes are given in advance.

Table 6: Execution times for Bit-CKKS binary NAND gate bootstrapping (in seconds), multiplication (in milliseconds), and size estimations for the relinearization key (rlk, in MB).

Parameters	nandBoot	Mult	rlk size
simBinFHE14	5.18	16.1	47.71
graBinFHE14	2.74 (1.9×)	16.2 (1.0×)	16.52 (65%↓)
graBinFHE14Opt	2.75 (1.9×)	15.7 (1.0×)	16.52 (65%↓)
simBinFHE16	102.1	884.8	144.70
graBinFHE16	56.41 (1.8×)	428.1 (2.1×)	109.05 (25%↓)

5.3 Homomorphic Comparison with Grafting

We display the experimental results for the homomorphic comparison in Tables 7 and 8, based on the parameters in Table 2. We provide implementations that mainly reduce the modulus consumption and execution timings of various homomorphic comparison functions proposed in [CKK20] while maintaining precision. Specifically, we used seven-degree polynomials f_3 and g_3 from [CKK20] to construct homomorphic comparison functions.

We use two different homomorphic comparison functions, comp1 and comp2, both composing the f and g functions but by different numbers targeting different input ranges and precisions. The input range of the homomorphic comparison function can be denoted by $[-1, -\epsilon] \cup [\epsilon, 1]$ where $\epsilon = 2^{-8}$ for comp1 and $\epsilon = 2^{-16}$ for comp2. The function f_3 is composed twice for both comparison functions. The function g_3 is four times composed for comp1 and eight times for comp2.

In the following tables, the amounts of modulus consumed per step are denoted as $X \times Y$, where X represents the number of bits consumed when computing a single polynomial (either f_3 or g_3) and Y is the number of such polynomials. Note that computing a seven-degree polynomial requires three multiplicative depths, so the scale factor used during computing a polynomial is $X/3$ bits.

Table 7 reports 10% to 27% reduction in modulus consumption compared to the simple(non-grafted) case. In practice, computing homomorphic comparison requires a few bootstrappings. Therefore, reducing modulus consumption can be directly converted into significant speed-up by reducing the number of bootstrapping. Additionally, adjusting the available modulus for each bootstrapping using the flexible output modulus bootstrapping described in the Appendix F can further enhance the speed up of each bootstrapping. Table 8 reports 1.24× to 1.56× speed up compared to the simple (non-grafted) case.

Table 7: Modulus consumption for homomorphic comparison. The per-step modulus consumption is denoted by $X \times Y$, where X represents the modulus consumed for computing a single polynomial (either f_3 or g_3) and Y represents the number of such polynomials. An additional parameter is introduced in advance.

Configs	Parameters	Modulus Consumption		Prec
		Total	Per-Step	
comp1	simFHE16	630	105×6	16.3
	graFHE16	531 (16%↓)	$84 \times 4 + 90 + 105$	16.5
comp2	simFHE16	1050	105×10	16.4
	graFHE16	942 (10%↓)	$93 \times 9 + 105$	16.4
comp1	simFHE16s	756	126×6	23.1
	graFHE16s	552 (27%↓)	$84 \times 4 + 90 + 126$	23.1
comp2	simFHE16s	1260	126×10	23.5
	graFHE16s	963 (24%↓)	$93 \times 9 + 126$	23.3

Table 8: Execution timings for homomorphic comparison functions with bootstrapping. For graFHE16s, we additionally applied the flexible output modulus bootstrapping described in Appendix F.

Configs	Parameters	Bootstrapping		Time(s)
		#BTS	Avail. Modulus	
comp1	simFHE16s	1	378× 1	42.7
	graFHE16s	1	216	34.3 (1.24)
comp2	simFHE16s	3	378×3	114.5
	graFHE16s	2	372 + 219	73.5 (1.56)

REFERENCES

[AAB+23] Rashmi Agrawal, Jung Ho Ahn, Flavio Bergamaschi, Ro Cammarota, Jung Hee Cheon, Fillipe D. M. de Souza, Huijing Gong, Minsik Kang, Duhyeong Kim, Jongmin Kim, Hubert de Lassus, Jai Hyun Park, Michael Steiner, and Wen Wang. High-precision RNS-CKKS on fixed but smaller word-size architectures: theory and application. In *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '23, page 23–34, New York, NY, USA, 2023. Association for Computing Machinery.

[ABBB+22] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'22, pages 53–63, New York, NY, USA, 2022. Association for Computing Machinery.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[BCC+24] Jean-Philippe Bossuat, Rosario Cammarota, Ilaria Chillotti, Benjamin R. Curtis, Wei Dai, Huijing Gong, Erin Hales, Duhyeong Kim, Bryan Kumara, Changmin Lee, Xianhui Lu, Carsten Maple, Alberto Pedrouzo-Ulloa, Rachel Player, Yuriy Polyakov, Luis Antonio Ruiz Lopez, Yongsoo Song, and Donggeon Yhee. Security guidelines for implementing homomorphic encryption. Cryptology ePrint Archive, Paper 2024/463, 2024.

[BCG+23] Mariya Georgieva Belorgey, Sergiu Carpov, Nicolas Gama, Sandra Guasch, and Dimitar Jetchev. Revisiting key decomposition techniques for FHE: Simpler, faster and more generic. Cryptology ePrint Archive, Paper 2023/771, 2023.

[BCH+24] Youngjin Bae, Jung Hee Cheon, Guillaume Hanrot, Jai Hyun Park, and Damien Stehlé. Plaintext-ciphertext matrix multiplication and FHE bootstrapping: Fast and fused. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part III*, volume 14922 of *LNCS*, pages 387–421. Springer, Cham, August 2024.

[BCK+23] Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, Jai Hyun Park, and Damien Stehlé. HERMES: Efficient ring packing using MLWE ciphertexts and application to transciphering. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 37–69. Springer, Cham, August 2023.

[BCKS24] Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, and Damien Stehlé. Bootstrapping bits with CKKS. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part II*, volume 14652 of *LNCS*, pages 94–123. Springer, Cham, May 2024.

[BEHZ16] Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. A full RNS variant of FV like somewhat homomorphic encryption schemes. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, pages 423–442. Springer, Cham, August 2016.

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

[BKSS24] Youngjin Bae, Jaehyung Kim, Damien Stehlé, and Elias Suvanto. Bootstrapping small integers with CKKS. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part I*, volume 15484 of *LNCS*, pages 330–360. Springer, Singapore, December 2024.

[BMTH21] Jean-Philippe Bossuat, Christian Mouchet, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 587–617. Springer, Cham, October 2021.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 868–886. Springer, Berlin, Heidelberg, August 2012.

[BTH22] Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation. In Giuseppe Ateniese and Daniele Venturi, editors, *ACNS 22International Conference on Applied Cryptography and Network Security*, volume 13269 of *LNCS*, pages 521–541. Springer, Cham, June 2022.

[CCKS23] Jung Hee Cheon, Wonhee Cho, Jaehyung Kim, and Damien Stehlé. Homomorphic multiple precision multiplication for CKKS and reduced modulus consumption. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 696–710. ACM Press, November 2023.

[CCP24] Jung Hee Cheon, Hyeongmin Choe, and Jai Hyun Park. Tree-based lookup table on batched encrypted queries using homomorphic encryption. Cryptology ePrint Archive, Report 2024/087, 2024.

[CHK+19] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In Carlos Cid and Michael J.: Jacobson, Jr., editors, *SAC 2018*, volume 11349 of *LNCS*, pages 347–368. Springer, Cham, August 2019.

[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT multiplication for NTT-unfriendly rings: New speed records for saber and NTRU on Cortex-M4 and AVX2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, Feb. 2021.

[CKK20] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. Efficient homomorphic comparison methods with optimal complexity. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*, pages 221–256. Springer, Cham, December 2020.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 409–437. Springer, Cham, December 2017.

[Cry22] CryptoLab. HEaaN library, 2022. Available at https://heaan.it/.

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

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 850–867. Springer, Berlin, Heidelberg, August 2012.

[HHS+21] Shai Halevi, Hamish Hunt, Victor Shoup, Oliver Masters, Flavio Bergamaschi, Jack Crawford, Fabian Boemer, et al. HElib (version 2.2.1), October 2021. Available at https://github.com/homenc/HElib.

[HK20] Kyoohyung Han and Dohyeong Ki. Better bootstrapping for approximate homomorphic encryption. In Stanislaw Jarecki, editor, *CT-RSA 2020*, volume 12006 of *LNCS*, pages 364–390. Springer, Cham, February 2020.

[HS20] Shai Halevi and Victor Shoup. Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive, Paper 2020/1481, 2020.

[HS21] Shai Halevi and Victor Shoup. Bootstrapping for helib. *J. Cryptol.*, 34(1), January 2021.

[KLSS23] Miran Kim, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. Accelerating HE operations from key decomposition technique. *CRYPTO 2023.*, Aug 2023.

[KPK+22] S. Kim, M. Park, J. Kim, T. Kim, and C. Min. EvalRound algorithm in CKKS bootstrapping. In *ASIACRYPT*, 2022.

[KPP22] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. Approximate homomorphic encryption with reduced approximation error. In Steven D. Galbraith, editor, *CT-RSA 2022*, volume 13161 of *LNCS*, pages 120–144. Springer, Cham, March 2022.

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

[LLK+23] Seewoo Lee, Garam Lee, Jung Woo Kim, Junbum Shin, and Mun-Kyu Lee. Hetal: efficient privacy-preserving transfer learning with homomorphic encryption. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[LLKN22] Eunsang Lee, Joon-Woo Lee, Young-Sik Kim, and Jong-Seon No. Optimization of homomorphic comparison algorithm on rns-ckks scheme. *IEEE Access*, 10:26163–26176, 2022.

[LLL+22] Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 12403–12422. PMLR, 17–23 Jul 2022.

[LLNK22] Eunsang Lee, Joon-Woo Lee, Jong-Seon No, and Young-Sik Kim. Minimax approximation of sign function by composite polynomial for homomorphic comparison. *IEEE Transactions on Dependable and Secure Computing*, 19(6):3711–3727, 2022.

[MG23] Johannes Mono and Tim Güneysu. A new perspective on key switching for bgv-like schemes. *Cryptology ePrint Archive*, 2023.

[Par25] Jai Hyun Park. Ciphertext-ciphertext matrix multiplication: Fast for large matrices. Cryptology ePrint Archive, Paper 2025/448, 2025.

[RKP+25] Donghwan Rho, Taeseong Kim, Minje Park, Jung Woo Kim, Hyunsik Chae, Ernest K. Ryu, and Jung Hee Cheon. Encryption-friendly llm architecture, 2025.

[SEA23] Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, January 2023. Microsoft Research, Redmond, WA.

[SS24] Nikola Samardzic and Daniel Sanchez. Bitpacker: Enabling high arithmetic efficiency in fully homomorphic encryption accelerators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 137–150, New York, NY, USA, 2024. Association for Computing Machinery.

[SSKM24] Hyewon Sung, Sieun Seo, Taekyung Kim, and Chohong Min. EvalRound+ bootstrapping and its rigorous analysis for CKKS scheme. Cryptology ePrint Archive, Paper 2024/1379, 2024.

A MORE RELATED WORKS

Approaches Filling the Machine Word Sizes. The idea to use word-sized primes mostly and a few small primes in the RNS factors was first proposed by Gentry, Halevi, and Smart [GHS12] from

their BGV implementation [HS20]. They introduce a method of choosing word-size primes and some smaller primes, enabling the ciphertext modulus to be fairly close to any desired target value. For key-switching, they switch the modulus by putting more word-sized primes and dropping non-word-sized primes, maintaining the same message. However, this approach is not applicable to CKKS since the message is not well preserved as in BGV. This induces an absolutely larger error during modulus-switching unless the input/output ciphertext moduli are extremely close.

In [KLSS23], the machine word-sized moduli were partly utilized for the key-switching operation. However, their technique is advantageous only for sufficiently large gadget ranks and applies only to the key-switching operation. Grafting, however, accelerates the whole homomorphic computations with no such restrictions on gadget ranks.

A following work by Belorgey et al. [BCG+23] extended their technique to digit-based gadget decompositions and proposed to implement FHE using binary modulus with Discrete Fourier Transform (DFT) instead of NTT. However, this requires higher precision and increases the number of DFT units, leading to less efficient implementation.

Machine-dependent Approaches. Agrawal et al. [AAB+23] proposed an implementation depending on the machine word size to design a 32-bit hardware implementation with the RNS-CKKS parameters. As the scaling factors range from 48 to 58 bits, two NTT primes of 24 to 29 bits were required for each rescale operation. It is worth noting that the smaller primes are hard to use since there are fewer NTT primes, such as 24 to 29 bits, lacking NTT primes for larger ring degrees. This was also the case in Tuple-CKKS [CCKS23], where the performance was restricted due to the non-existence of NTT primes of appropriate sizes.

B MISSING PRELIMINARIES

B.1 Number Theoretic Transform

For a polynomial $\mathbf{a} = a_0 + a_1 x + \cdots + a_{N-1}x^{N-1} \in \mathcal{R}_q$, we let $\mathrm{NTT}(\mathbf{a}) = (\mathbf{a}(\zeta^0), \mathbf{a}(\zeta^1), \cdots, \mathbf{a}(\zeta^{N-1})) \in \mathbb{Z}_q^N$ be a number theoretic transform (NTT) of $\mathbf{a}$ in modulus q, where $\zeta \in \mathbb{Z}_q$ be a primitive N-th root of unity in $\mathbb{Z}_q$, which only exist when $2N|(q-1)$. We call primes q satisfying the condition $2N|(q-1)$ the NTT primes (with respect to ring dimension N and modulo q). For a vector $\mathbf{b} = (b_0, \cdots, b_{N-1}) \in \mathbb{Z}_q^N$, we let $\mathrm{iNTT}(\mathbf{b}) = \sum_{i=0}^{N-1} \tilde{b_i} x^i$ be an inverse NTT transfrom (iNTT), where $\tilde{b_i} = n^{-1} \cdot \sum_{j=0}^{N-1} b_j \cdot \zeta^{-ij} \in \mathbb{Z}_q$. Note that NTT and iNTT commutes, i.e.,

$$\mathrm{NTT}(\mathrm{iNTT}(\mathbf{b})) = \mathbf{b}, \text{ and } \mathrm{iNTT}(\mathrm{NTT}(\mathbf{a})) = \mathbf{a},$$

and that NTT and iNTT are homomorphic. We let the coefficient vector $(a_0, a_1, \ldots, a_{N-1}) \in \mathbb{Z}_q^N$ be an *NTT-coefficient* format of $\mathbf{a}$, and $\mathrm{NTT}(\mathbf{a}) \in \mathbb{Z}_q^N$ be an *NTT-evaluated* format of $\mathbf{a}$.

B.2 Basic Computations in RNS-CKKS

B.2.1 Fast Basis Conversion in [CHK+19]. Let $\mathcal{B} = \{p_0, \ldots, p_{k-1}\}$ and $C = \{q_0, \ldots, q_{l-1}\}$ be the bases for moduli $P = \prod_{i=0}^{k-1} p_i$ and $Q = \prod_{j=0}^{l-1} q_j$, respectively, where the base moduli are pairwise relatively prime. A RNS representation of an element $\mathbf{a} \in \mathbb{Z}_Q$ is denoted by

$$[\mathbf{a}]_C = (\mathbf{a}^{(0)}, \ldots, \mathbf{a}^{(l-1)}) \in \mathbb{Z}_{q_0} \times \cdots \times \mathbb{Z}_{q_{l-1}} \ .$$

One can convert such $\mathbf{a}$ into its RNS representation with respect to $\mathbb{Z}_P$ as

$$\mathrm{Conv}_{C\to\mathcal{B}}([\mathbf{a}]_C) = \left(\sum_{j=0}^{\ell-1} [\mathbf{a}^{(j)} \cdot \hat{q}_j^{-1}]_{q_j} \cdot \hat{q}_j \pmod{p_i} \right)_{0\le j<k},$$

where $\hat{q}_j = Q/q_j$. Note that $\tilde{\mathbf{a}} := \sum_{j=0}^{\ell-1} [\mathbf{a}^{(j)} \cdot \hat{q}_j^{-1}]_{q_j} \cdot \hat{q}_j = \mathbf{a} + Qe$ for some small $e \in \mathbb{Z}$ satisfying $|\tilde{\mathbf{a}}| \le (\ell/2) \cdot Q$. Let us use the notation $Q \to P$ instead of $C \to \mathcal{B}$ if there is no confusion.

B.2.2 Modulus Switching. For a polynomial $\mathbf{a} \in \mathcal{R}_Q^2$, we define the ModUp procedure so that the resulting polynomial is in $\mathcal{R}_{PQ}$, but with the same value in modulus Q and not too large. ModDown reduces the modulus from PQ to Q. It reduces the size of the polynomial and the modulus with the same factor, i.e., by a factor of $Q/PQ \sim P^{-1}$. RS is the same as ModDown but is rescaled by fewer moduli factors than ModDown. It reduces the size of the polynomial and the modulus by q_ℓ. Let us borrow the notations from the previous Section and let $\mathcal{D} = \mathcal{B} \cup C$. Precisely,

$$\begin{aligned}
\mathrm{ModUp}_{C\to\mathcal{D}}(\cdot) : \prod_{j=0}^{\ell-1} \mathcal{R}_{q_j} &\to \prod_{i=0}^{k-1} \mathcal{R}_{p_i} \times \prod_{j=0}^{\ell-1} \mathcal{R}_{q_j} \\
[\mathbf{a}]_C &\to (\mathrm{Conv}_{C\to\mathcal{B}}([\mathbf{a}]_C), [\mathbf{a}]_C), \\
\mathrm{ModDown}_{\mathcal{D}\to C}(\cdot) : \prod_{i=0}^{k-1} \mathcal{R}_{p_i} \times \prod_{j=0}^{\ell-1} \mathcal{R}_{q_j} &\to \prod_{j=0}^{\ell-1} \mathcal{R}_{q_j} \\
([\mathbf{a}]_{\mathcal{B}}, [\mathbf{b}]_C) &\to [P^{-1}]_C \cdot ([\mathbf{b}]_C - \mathrm{Conv}_{\mathcal{B}\to C}([\mathbf{a}]_{\mathcal{B}})),
\end{aligned}$$

and $\mathrm{RS}_{q_{\ell-1}}(\cdot) = \mathrm{ModDown}_{C\to C'}(\cdot)$, where $C' = C \setminus \{q_{\ell-1}\}$.

We note that the ModUp operation maps $\mathbf{a} \in \mathcal{R}_Q$ to $\mathbf{a}+Q\mathbf{e} \in \mathcal{R}_{PQ}$, where $|\mathbf{e}| \le \ell/2$ from the fast basis conversion. The ModDown operation maps $\mathbf{a} = ([\mathbf{a}]_P, [\mathbf{a}]_Q) \in \mathcal{R}_{PQ}$ to $\mathbf{a}' = P^{-1} \cdot (\mathbf{a} - \tilde{\mathbf{a}}) \in \mathcal{R}_Q$, where $\tilde{\mathbf{a}} \equiv \mathbf{a} \pmod{P}$ and $\|\tilde{\mathbf{a}}\|_\infty \le (k/2) \cdot P$, resulting $\|\mathbf{a}' - P^{-1} \cdot \mathbf{a}\|_\infty = P^{-1} \cdot \|\tilde{\mathbf{a}}\|_\infty \le k/2$. RS introduces an error of size $\le 1/2$.

When applied to ciphertext, the error becomes multiplied by the secret key $\mathbf{s}$ and thus has an infinity norm of $\le k/2 \cdot (\|\mathbf{s}\|_1 + 1)$ and $\le 1/2 \cdot (\|\mathbf{s}\|_1 + 1)$, respectively, for ModDown and RS. Let us use the notation $Q \to PQ$ (Resp. $PQ \to Q$) instead of $C \to \mathcal{D}$ (Resp. $\mathcal{D} \to C$) if there is no confusion. We note that in [AAB+23], it is demonstrated that rather than processing the ModDown at once, splitting the procedure into two or more steps – such as from PQ to p_0Q, and then Q – can reduce the output error by a factor of k with only small additional costs for Hadamard multiplications.

We additionally define the Inv-RS operation, which is sometimes called zero-padding. It multiplies a factor to both the ciphertext and its modulus and is used during key switching with gadget decomposition.

B.2.3 Inverse Rescale. For given a polynomial $\mathbf{a} \in \mathcal{R}_Q$, we define inverse rescaling by an integer factor R as

$$\text{Inv-RS}_R(\mathbf{a}) = R \cdot \mathbf{a} \in \mathcal{R}_{QR},$$

or in the RNS representation, one can write as:

$$\text{Inv-RS}_R(\mathbf{a}) \equiv \begin{cases} [R]_{q_i} \cdot [\mathbf{a}]_{q_i} & (\text{mod } q_i) \\ 0 & (\text{mod } r_j) \end{cases},$$

for $i \in [\ell]$, $j \in [k]$, where $Q = \prod_{i=0}^{\ell} q_i$ and $R = \prod_{j=0}^{k} r_j$ with co-prime NTT primes q_i's and r_j's. We note that can be naturally extended to a vector of polynomials or ciphertexts.

B.2.4 Level Adjustments. In the RNS setting, we rescale the ciphertexts during multiplication by one of the RNS factors, say Q_i, instead of the real scaling factor Δ. This yields an additional error after a series of rescaling. In [CHK+19], it is suggested that choosing each modulus q_i as close as possible to Δ to minimize the error, and later in [KPP22] the authors suggest using level-specific scaling factors. Specifically, the scaling factor Δ_ℓ for each level ℓ is defined as $\Delta_{\ell-1} \coloneqq \Delta_\ell^2/q_\ell$, iteratively from $\ell = L$ to 1. With different scaling factors in different levels, one may need to manipulate two input ciphertexts having different levels and, thus, different scaling factors. To adjust the ciphertext, the level adjusting technique is introduced [KPP22].

Let ct and ct′ be the ciphertexts with level ℓ and ℓ' ($\ell > \ell'$) and scaling factors Δ_ℓ and $\Delta_{\ell'}$, respectively. Before performing homomorphic operations over ct and ct′, we adjust ct to level ℓ' with the scaling factor $\Delta_{\ell'}$, by Adjust operation: For inputs ct in level $\ell > \ell'$, and the target level ℓ',

(1) Let $\text{ct} = [\text{ct}]_{q_0 \cdots q_{\ell'+1}} \in \mathcal{R}^2_{q_0 \cdots q_{\ell'+1}}$ by dropping the RNS factors $\{q_{\ell'+1}, \ldots, q_\ell\}$,
(2) Multiply a constant $\left\lceil \frac{\Delta_{\ell'} \cdot q_{\ell'+1}}{\Delta_\ell} \right\rfloor$ in $\mathcal{R}_{q_0 \cdots q_{\ell'+1}}$.
(3) RS by $q_{\ell'+1}$.

The resulting ciphertext is in $\mathcal{R}^2_{q_{\ell'}}$ and has a scaling factor $\Delta_{\ell'}$ with an additional error, which is approximately a rounding error.

C MISSING DETAILS FROM SECTION 3

C.1 Key Switching with an Intermediate Modulus Chain

This section presents an alternative approach for achieving full decoupling between the scaling factor and modulus, independent of grafting. Specifically, the idea is to temporarily switch to an intermediate modulus, chosen as a divisor of the switching key modulus, and to execute the key switching operation within this intermediate modulus. That is, for a ciphertext encrypting an encoded message $\Delta\mathbf{m} + \mathbf{e}$ in modulus Q, one can choose a modulus $Q_{\text{int}} \gtrsim Q$ satisfying $Q_{\text{int}} \mid PQ_{\max}$. Similar to the rational rescaling, one can switch the modulus of the ciphertext to Q_{int}, then apply key switching. The resulting ciphertext will encrypt an encoded message $(Q_{\text{int}}/Q) \cdot (\Delta\mathbf{m} + \mathbf{e}) + \mathbf{e}'$ with a switched secret key, where $\mathbf{e}'$ includes the rescale and the key switching errors. We can convert the modulus back to Q using the rational rescale procedure while maintaining the error within $\mathbf{e}$ plus the rescale error.

Extending the above method, one can prepare a switching key in a modulus Q_{swk} consisting mostly of word-sized RNS factors. Depending on the scaling factors, we can rationally rescale the ciphertext modulus from Q to $Q' \approx Q/\Delta$, while temporarily moving the ciphertext modulus to $Q_{\text{int}} \mid Q_{\text{swk}}$ for key switching.

$$\begin{array}{ccccc} Q_{\text{swk}} & & Q_{\text{swk}} & & Q_{\text{swk}} \\ \updownarrow \text{KeySwitch} & & \updownarrow \text{KeySwitch} & & \updownarrow \text{KeySwitch} \\ Q_{\text{int}} & & Q'_{\text{int}} & & Q''_{\text{int}} \\ \updownarrow & & \updownarrow & & \updownarrow \\ \cdots \to Q \to & & Q' \approx Q/\Delta \to & & Q'' \approx Q'/\Delta \to \cdots \end{array}$$

However, the temporary change in modulus from Q to Q_{int} introduces additional (i)NTT operations, even when fused with the ModUp operation. Specifically, ModUp requires the same number of (i)NTT operations for inputs of coefficients and for NTT-evaluated formats to output the results in an NTT-evaluated state since some NTT-evaluated inputs can be reused. Thus, we need additional $(\ell + 1)$ (i)NTT operations for the temporary modulus switching, where $\ell + 1$ is the number of RNS factors in Q.

In addition, the temporary modulus switching, as well as the rational rescaling, induce additional Hadamard multiplication costs. They are not small, especially when the factors of the two moduli do not overlap much. Precisely, when assuming $\ell = \ell_1 + \ell_2$ factors in Q and $\ell' = \ell_2 + \ell_3$ factors in Q' (or Q_{int}), where exactly the ℓ_2 factors are overlapped, the cost for changing the modulus from Q to Q' is $O\left(N(\ell + (\ell - \ell_2) \cdot \ell')\right)$, where N is the ring dimension.

C.2 Proof of Theorem 3.2

Proof. Let us follow the notations in Definition 3.1. Let $[\langle \text{ct}, \text{sk} \rangle]_Q = \Delta \cdot m + e$, or $\langle \text{ct}, \text{sk} \rangle = \Delta \cdot m + e + QI$ for some $I \in \mathcal{R}$. Then $\langle \text{Inv-RS}_R(\text{ct}), \text{sk} \rangle = R\Delta \cdot m + Re + QRI$. The final rescaling RS_S introduces an additional error e_{res}, where $\|e_{\text{res}}\|_\infty \le \ell/2 \cdot (\|s\|_1 + 1)$ for ℓ the number of RNS blocks in $S = \text{lcm}(Q, Q')/Q'$. Precisely, we have

$$\langle \text{RS}_{Q/Q'}(\text{ct}), \text{sk} \rangle = (Q'/Q) \cdot (\Delta \cdot m + e) + Q'I + e_{\text{res}},$$

which concludes the proof. □

C.3 Correctness of Modulus Adjustment

Suppose we have a ciphertext $\text{ct} = (\mathbf{b}, \mathbf{a}) \in \mathcal{R}^2_Q$ satisfying the following relation,

$$\mathbf{b} + \mathbf{a} \cdot \mathbf{s} = \Delta \cdot \mathbf{m} + \mathbf{e} \pmod{Q},$$

with a scaling factor Δ and the modulus $Q = q_0 \ldots q_{\ell-1} \cdot r$ for ℓ unit moduli q_i's and a sprout r. Note that the adjustments can be fused into a circuit, as in the level adjustment technique.

We first choose an appropriate modulus Q_{mid} which is a divisor of Q to satisfy $Q \ge Q_{\text{mid}} > Q'$ and $Q_{\text{mid}} > Q' \cdot \Delta$. We reduce the ciphertext modulus to modulus Q_{mid}, by dropping some components. We then multiply an integer constant and rational rescale the ciphertext to the final modulus Q'. Specifically, the integer constant can be represented as

$$\left\lceil \frac{Q_{\text{mid}}\Delta'}{Q'\Delta} \right\rfloor = \frac{Q_{\text{mid}}\Delta'}{Q'\Delta} + \delta,$$

for some $\delta \in (-1/2, 1/2]$. For the output ciphertext $\mathsf{ct}' = (\mathbf{b}', \mathbf{a}') \in \mathcal{R}^2_{Q'}$, it holds that

$$\begin{aligned}\mathbf{b}' + \mathbf{a}' \cdot \mathbf{s} &= \frac{Q'}{Q_{\mathrm{mid}}} \cdot \left(\frac{Q_{\mathrm{mid}}\Delta'}{Q'\Delta} + \delta \right) \cdot (\Delta \mathbf{m} + \mathbf{e}) + \mathbf{e}_{\mathrm{res}} \pmod{Q'} \\ &= \left(\frac{\Delta'}{\Delta} + \frac{Q'\delta}{Q_{\mathrm{mid}}} \right) \cdot (\Delta \mathbf{m} + \mathbf{e}) + \mathbf{e}_{\mathrm{res}} \pmod{Q'} \\ &= \Delta' \mathbf{m} + \mathbf{e}_{\mathrm{adj}} \pmod{Q'},\end{aligned}$$

where the $\mathbf{e}_{\mathrm{res}}$ is an error added during rescaling, and $\mathbf{e}_{\mathrm{adj}}$ is defined as

$$\mathbf{e}_{\mathrm{adj}} = \frac{\Delta' \mathbf{e}}{\Delta} + \frac{Q'\delta\Delta\mathbf{m}}{Q_{\mathrm{mid}}} + \frac{\delta Q' \mathbf{e}}{Q_{\mathrm{mid}}} + \mathbf{e}_{\mathrm{res}}.$$

The error is bounded by the scaled error, the rescale error, and the rounding error, where the rescale error becomes the most significant one. We note that the condition $Q_{\mathrm{mid}} > Q' \cdot \Delta$ allows us to manage the error $\mathbf{e}_{\mathrm{adj}}$ to be sufficiently small.

C.4 Proof of Theorem 3.3

Proof. Let ct be a ciphertext with modulus $Q = q_0 \cdots q_\ell \cdot r$, where r is a sprout satisfying $r | r_{\mathrm{top}}$, and $r \in 2^\gamma \cdot [1-\epsilon, 1+\epsilon]$, where $0 \le \gamma < w$. Let $w\ell + \gamma - \delta = w\ell' + \gamma'$, where $0 \le \gamma' < w$. Since $\delta = w(\ell - \ell') + \gamma - \gamma'$, it holds that $n = \lceil \delta / w \rceil = \lceil \ell - \ell' + (\gamma - \gamma')/w \rceil \ge \ell - \ell'$. Moreover, there exists $r' | r_{\mathrm{top}}$ such that $r' \in 2^{\gamma'} \cdot [1-\epsilon, 1+\epsilon]$ from the assumption. Therefore, for a modulus $Q' = q_0 \cdots q_{\ell'} \cdot r'$, we have

$$\frac{1-\epsilon}{1+\epsilon} \cdot (1-\eta)^{\ell - \ell'} \le \frac{Q/Q'}{2^\delta} = \frac{q_{\ell'+1}}{2^w} \cdots \frac{q_\ell}{2^w} \cdot \frac{r}{2^\gamma} \cdot \frac{2^{\gamma'}}{r'} \le \frac{1+\epsilon}{1-\epsilon} \cdot (1+\eta)^{\ell - \ell'},$$

which concludes the proof. □

Note that the choice of sprout in Examples 3.1 and 3.2 are universal sprouts satisfying the assumption of Theorem 3.3 with $\epsilon < 2^{-13}$. Also, there are plenty of 61-bit primes in $[2^\omega(1 - 2^{-20}), 2^\omega(1+2^{-20})]$. Thus, these sprouts have *universal rescalability*, i.e., the ciphertexts are rational rescalable by $2^\delta \cdot (1 \pm 2^{-12})$ for any $\delta \in \mathbb{N}$ smaller than the current ciphertext modulus, where the ciphertext moduli are grafted with the sprout.

C.5 Candidates for Universal Sprout

We propose another candidate for a universal sprout that is applicable to higher ring dimensions N, as follows:

Example C.1 (`sprout-19-20-23`). *Let q_i's be 62-bit NTT primes and $r_{top} = 2^{19} \cdot r_1 \cdot r_2$, where r_1 is a 20-bit NTT prime, and r_2 is a 23-bit NTT prime. Typically, we can choose $r_1 = 1{,}179{,}649$ and $r_2 = 8{,}519{,}681$, the NTT primes for ring dimension $N \le 2^{17}$. Here, the sprouts can represent any bit length from 1 to 62 as*

$$\{2^1, \cdots, 2^{19}, r_1, 2r_1, 2^2 r_1, r_2, \cdots, 2^{19} r_2, r_2 r_1, \cdots, 2^{18} r_2 r_1\}.$$

C.6 Efficient Sprout Arithmetic

In this subsection, we discuss hardware-level arithmetic strategies, mostly focusing on standard 64-bit processors. In other words, we figure out efficient instantiations of the methods described in the previous section.

C.6.1 Basics for 64-bit processors. Given a (universal) sprout for standard 64-bit processors, a naive approach requires three 64-bit moduli to handle any polynomial multiplications via NTT. To check this, let $r < 2^{64}$ be a product of all moduli in the sprout. In order to deal with polynomial multiplication over modulo r, one may use *emulated NTT*: embed modulo r into a larger modulus $p > Nr^2$ so that NTT over modulo p emulates polynomial multiplication modulo r without modular reduction by p ever occurring.[11] In this case, as p should typically be larger than 2^{128} ($\because$ r is close to 2^{64}), we need three 64 bit NTT primes.

For 64-bit processors, we mainly suggest using two NTT moduli rather than three. This can be enabled through *composite number NTT* [CHK+21], which constructs an NTT modulus out of composite number instead of a prime number.

Example C.2. *We follow the settings in Example 3.2 so that $r_{top} = 2^{15} \cdot r_1 \cdot r_2$ where r_1 is a 16-bit NTT prime and r_2 is a 30-bit NTT prime. The key observation is that the modulus $r_1 r_2 \simeq 2^{46}$ can be handled with a composite NTT. As both r_1 and r_2 are NTT primes, one can easily find $2N$-th primitive roots of unity modulo $r_1 r_2$. Then, we may construct one NTT modulus to be $r_1 r_2$ and the other one to be a larger modulus $g > N \cdot 2^{30}$ so that we can emulate modulo 2^{15} arithmetic in modulo g arithmetic. This results in using two NTT moduli instead of three, as desired.*

Note that the strategy described in Example C.2 is compatible with RLWE switching keys: if one has only either r_1 or r_2 in the sprout modulus at some point, then this can be naturally embedded into the modulus $r_1 r_2$ without any costly transformations. Notably, we may properly define the embedding $\mathsf{Embed}_{r_1 \to r_1 r_2} : \mathcal{R}_{r_1} \to \mathcal{R}_{r_1 r_2}$ to be compatible with NTT. That is, we have

$$\mathsf{Embed}_{r_1 \to r_1 r_2} \circ \mathsf{NTT}_{r_1} = \mathsf{NTT}_{r_1 r_2} \circ \mathsf{Embed}_{r_1 \to r_1 r_2}$$

in $\mathcal{R}_{r_1}$, where NTT_r is an NTT over modulo r. For instance, one may define embedding as simply putting 0 modulo r_2 and performing the CRT. Then the equality is directly checked in both modulo r_1 (where both sides correspond to NTT_{r_1}) and modulo r_2 (where both sides are 0).

If the sprout modulus is sufficiently small, then we may even use one word-sized NTT for the sprout. In particular, the following types of sprouts from Example 3.2 support single-word arithmetic.

- Sprout with no r_2: As g is chosen to be sufficiently large to ensure that the coefficients modulo 2^{15} do not exceed g, the emulated part and the r_1 part can be computed using a single-word composite NTT, modulo $g \cdot r_1$.
- Sprout with sufficiently small power-of-two: Consider the sprout $2^t \cdot r_1 \cdot r_2$ for some $t \le 15$. Note that the modulus 2^t can be embedded into a modulus $g' > 2^{2t} \cdot N$, and the whole sprout can be embedded into $g' r_1 r_2$. Hence, if $g' r_1 r_2 < 2^{64}$, we may use a single-word modulus for composite NTT.

C.6.2 Other machine word sizes. We may consider using a different word size than the standard 64-bit. The main strategy is almost the same as the standard one. We pack as many NTT moduli as possible using composite NTT and embed the remaining moduli into a larger modulus.

[11] We compute via inclusions $\mathcal{R}_r \hookrightarrow \mathcal{R} \cup [-p, p]^N \hookrightarrow \mathcal{R}_p$.

Example C.3. *The strategy for 2^ω-bit word size for $\omega \geq 7$ (i.e., 128-bit or larger size architectures) can easily be generalized from the 64-bit case. That is, we choose $r_{top} = 2^{15} \cdot r_1 \cdot r_2 \cdot \cdots \cdot r_{\log_2(\omega)-4}$ where r_i is a 2^{i+3}-bit prime for each $1 \leq i \leq \log_2(w) - 4$. Then, this sprout can be computed with two words, as in the standard 64-bit case, by embedding the power-of-two part to a larger prime and using composite NTT for the rest.*

Example C.4. *We describe a strategy for a 32-bit word size. Let $r_{top} = 2^{15} \cdot r$ be a sprout where $r = 2^{16}+1$ is an NTT prime. We embed 2^{15} to $g_1 g_2$ where g_1, g_2 are sufficiently large word-sized NTT primes such that $2^{30} \cdot N < g_1 g_2$. Then we may handle the sprout arithmetic with three words g_1, g_2, and r. If the power-of-two part 2^t become sufficiently small such that $2^{2t} \cdot N < g_1$, we can use two words.*

D TUPLE-CKKS, REVISITED

In [CCKS23], the authors introduced a novel multiplication algorithm for CKKS, reducing the amount of modulus consumption for each homomorphic multiplication. Asymptotically, their algorithm should have similar throughput and possibly better latency when switched to a smaller ring. However, in many cases, the reduced modulus consumption is not converted directly to efficiency gain because any computation modulo q has roughly the same performance as long as q fits in the machine word size. Grafting bridges the gap between expectation and reality: the tuple multiplication no longer has significant throughput degradation compared to the original (single) multiplication.

In this section, we check the compatibility of Grafting with the Tuple multiplication and their efficiency. We follow the notations from [CCKS23]: CT denotes a tuple of ciphertexts, $Q^{(\ell)}$ denotes the modulus for the ciphertext of level ℓ, $\otimes$ denotes the CKKS tensor operation, Relin denotes the CKKS relinearization, RS_q denotes the rescaling by q, DCP and RCB denote the decomposition of CKKS ciphertext into quotient and remainder and their recombination.

D.1 Compatibility

We check the compatibility of our method with the multiplication of [CCKS23]. For simplicity, we stick to the pair multiplication - the general tuple multiplication should be checked almost the same.

Let's recall the definitions of the components of the pair multiplication in [CCKS23, Definition 4.1, 4.3, 4.5]. In the definitions, $\otimes$ denotes the CKKS tensor operation, Relin denotes the CKKS relinearization, RS_q denotes the rescaling by q, DCP and RCB denote the decomposition of CKKS ciphertext into quotient and remainder and their recombination as defined in [CCKS23, Definition 3.3].

Definition D.1 (Pair Tensor). Let $\mathsf{CT}_1 = (\hat{\mathrm{ct}}_1, \check{\mathrm{ct}}_1), \mathsf{CT}_2 = (\hat{\mathrm{ct}}_2, \check{\mathrm{ct}}_2) \in R^2_{Q^{(\ell)}} \times R^2_{Q^{(\ell)}}$ be ciphertext pairs. The tensor of CT_1 and CT_2 is defined as

$$\mathsf{CT}_1 \otimes^2 \mathsf{CT}_2 := \left(\hat{\mathrm{ct}}_1 \otimes \hat{\mathrm{ct}}_2, \hat{\mathrm{ct}}_1 \otimes \check{\mathrm{ct}}_2 + \check{\mathrm{ct}}_1 \otimes \hat{\mathrm{ct}}_2\right) \in R^3_{Q^{(\ell)}} \times R^3_{Q^{(\ell)}}.$$

Definition D.2 (Pair Relinearize). Let $\mathsf{CT} = (\hat{\mathrm{ct}}, \check{\mathrm{ct}}) \in R^3_{Q^{(\ell)}} \times R^3_{Q^{(\ell)}}$ be an output of $\otimes^2$. The relinearization of CT is defined as

$$\mathsf{Relin}^2(\mathsf{CT}) = \mathsf{DCP}_{q_{\mathrm{div}}}(\mathsf{Relin}(q_{\mathrm{div}} \cdot \hat{\mathrm{ct}})) + (0, \mathsf{Relin}(\check{\mathrm{ct}})).$$

Definition D.3 (Pair Rescale). Let $\mathsf{CT} = (\hat{\mathrm{ct}}, \check{\mathrm{ct}}) \in R^2_{Q^{(\ell)}} \times R^2_{Q^{(\ell)}}$ be a ciphertext pair. Let $q_\ell = Q_\ell / Q_{\ell-1}$. The rescale of CT is defined as

$$\mathsf{RS}^2_{Q^{(\ell)}}(\mathsf{CT}) = \left(\mathsf{RS}_{Q^{(\ell)}}(\hat{\mathrm{ct}}),\ \mathsf{RS}_{Q^{(\ell)}}(q_{\mathrm{div}} \cdot \hat{\mathrm{ct}} + \check{\mathrm{ct}}) - q_{\mathrm{div}} \cdot \mathsf{RS}_{Q^{(\ell)}}(\hat{\mathrm{ct}})\right).$$

It belongs to $R^2_{Q^{(\ell-1)}} \times R^2_{Q^{(\ell-1)}}$.

When applying the concept of grafting to the double multiplication framework, we may perform all the operations except $\mathsf{Relin}(q_{\mathrm{div}} \cdot \hat{\mathrm{ct}})$ and $(0, \mathsf{Relin}(\check{\mathrm{ct}}))$, which we may outsource the computation to bigger modulus. The outsourced relinearization can be used in a black box manner, regarding them as a relinearization with slightly different error distributions. Although the relinearization error upper bound E_{Relin} is different, the new pair relinearization should give exactly the same inequality as the one in [CCKS23, Lemma 4.4].

Hence, the only difficulty is to define pair rescale when $Q^{(\ell)}/Q^{(\ell-1)}$ is not an integer, which can happen in our new framework. We define a generalized (rational) version of pair rescale as follows:

Definition D.4 (Pair Rescale, Rational). Let $\mathsf{CT} = (\hat{\mathrm{ct}}, \check{\mathrm{ct}}) \in R^2_{Q^{(\ell)}} \times R^2_{Q^{(\ell)}}$ be a ciphertext pair. Let $\alpha_\ell / \beta_\ell = Q^{(\ell)}/Q^{(\ell-1)}$ where $\alpha_\ell, \beta_\ell \in \mathbb{Z}_{>0}$ are coprime. The rescale $\mathsf{RS}^2_{\alpha_\ell/\beta_\ell}$ of CT is defined in $R^2_{Q^{(\ell-1)}} \times R^2_{Q^{(\ell-1)}}$ as

$$\left(\mathsf{RS}_{\alpha_\ell}(\beta_\ell \cdot \hat{\mathrm{ct}}), \mathsf{RS}_{\alpha_\ell}(q_{\mathrm{div}}\beta_\ell \cdot \hat{\mathrm{ct}} + \beta_\ell \cdot \check{\mathrm{ct}}) - q_{\mathrm{div}} \cdot \mathsf{RS}_{\alpha_\ell}(\beta_\ell \cdot \hat{\mathrm{ct}})\right).$$

When applying the concept of Grafting to the double multiplication framework, we may perform all the operations except $\mathsf{Relin}(q_{\mathrm{div}} \cdot \hat{\mathrm{ct}})$ and $(0, \mathsf{Relin}(\check{\mathrm{ct}}))$, which we may outsource the computation to bigger modulus. The outsourced relinearization can be used in a black box manner, regarding them as a relinearization with slightly different error distributions. Although the relinearization error upper bound E_{Relin} is different, the new pair relinearization should give exactly the same inequality as the one in [CCKS23, Lemma 4.4].

We also give a generalized version of [CCKS23, Lemma 4.6] in Lemma D.5, showing the correctness of the pair multiplication after changing the pair rescale definition.

Lemma D.5. *Let $\mathsf{CT} \in R^2_{Q^{(\ell)}} \times R^2_{Q^{(\ell)}}$ be a ciphertext pair. Let $\alpha_\ell/\beta_\ell = Q^{(\ell)}/Q^{(\ell-1)}$ where $\alpha_\ell, \beta_\ell \in \mathbb{Z}_{>0}$ are coprime. Let $\mathsf{sk} = (1, \mathbf{s}) \in R^2$ be a secret key with $\mathbf{s}$ of Hamming weight h. Then the following quantity has an infinity norm of $\leq (h+1)/2$.*

$$\left[\left(\mathsf{RCB}_{q_{\mathrm{div}}}(\mathsf{RS}^2_{\alpha_\ell/\beta_\ell}(\mathsf{CT}))\right) \cdot \mathsf{sk}\right]_{Q^{(\ell-1)}} - \frac{\beta_\ell}{\alpha_\ell}\left[\left(\mathsf{RCB}_{q_{\mathrm{div}}}(\mathsf{CT})\right) \cdot \mathsf{sk}\right]_{Q^{(\ell)}}.$$

Proof. Let the quantity be S where $\alpha_\ell S \in \mathbb{Z}$. We have, modulo $Q^{(\ell-1)}$,

$$\left(\mathsf{RCB}_{q_{\mathrm{div}}}(\mathsf{RS}^2_{\alpha_\ell/\beta_\ell}(\mathsf{CT}))\right) \cdot \mathsf{sk} = \left(\mathsf{RS}_{\alpha_\ell}(\mathsf{RCB}_{q_{\mathrm{div}}}(\mathsf{CT}) \cdot \beta_\ell)\right) \cdot \mathsf{sk}.$$

Now, to complete the proof, note that

$$\alpha_\ell\left[\left(\mathsf{RS}_{\alpha_\ell}(\mathsf{RCB}_{q_{\mathrm{div}}}(\mathsf{CT}) \cdot \beta_\ell)\right) \cdot \mathsf{sk}\right]_{Q^{(\ell-1)}} - \beta_\ell\left[\left(\mathsf{RCB}_{q_{\mathrm{div}}}(\mathsf{CT})\right) \cdot \mathsf{sk}\right]_{Q^{(\ell)}}$$

has infinity norm $\leq \alpha_\ell \cdot (h+1)/2$. □

The Theorem D.6 is an analog of [CCKS23, Theorem 4.8], guaranteeing the correctness of pair multiplication. The correctness proof follows from the Lemma D.5. We define Mult^2 as a composition of tensor, relinearize, and rescale.

Theorem D.6. *Let* $\mathsf{CT} = (\hat{\mathsf{ct}}_1, \check{\mathsf{ct}}_1), \mathsf{CT}_2 = (\hat{\mathsf{ct}}_2, \check{\mathsf{ct}}_2) \in R^2_{Q^{(\ell)}} \times R^2_{Q^{(\ell)}}$ *be ciphertext pairs. Let* $\alpha_\ell/\beta_\ell = Q^{(\ell)}/Q^{(\ell-1)}$ *where* $(\alpha_\ell, \beta_\ell) = 1$ *and* $\mathsf{sk} = (1, \mathbf{s}) \in R^2$ *be a secret key with* $\mathbf{s}$ *of Hamming weight* h. *Assume that* $\|\mathsf{Dec}(\hat{\mathsf{ct}}_i)\|_\infty \le \hat{M}$ *and* $\|\mathsf{Dec}(\check{\mathsf{ct}}_i)\|_\infty \le \check{M}$ *for all* $i \in \{1, 2\}$ *and for some* $\hat{M}, \check{M}$ *satisfying* $N(\hat{M}q_{\mathsf{div}} + \check{M})^2 + E_{\mathsf{Relin}} + h < Q^{(\ell)}/2$. *Then the following quantity has an infinity norm of* $\le (N\check{M}^2/q_{\mathsf{div}} + E_{\mathsf{Relin}} + h)(\beta_\ell/\alpha_\ell) + (h+1)/2$.

$$\left[\left(\mathsf{RCB}_{q_{\mathsf{div}}}(\mathsf{Mult}^2(\mathsf{CT}_1, \mathsf{CT}_2))\right) \cdot \mathsf{sk}\right]_{Q^{(\ell-1)}} - \frac{\beta_\ell}{\alpha_\ell} \cdot \left[\left(\mathsf{RCB}_{q_{\mathsf{div}}}(\mathsf{CT}_1) \cdot \mathsf{sk}\right) \cdot \left(\mathsf{RCB}_{q_{\mathsf{div}}}(\mathsf{CT}_2) \cdot \mathsf{sk}\right)\right]_{Q^{(\ell)}}.$$

Proof. Let the quantity be S where $\alpha_\ell S \in \mathbb{Z}$. We have, modulo $Q^{(\ell-1)}$,

$$\left(\mathsf{RCB}_{q_{\mathsf{div}}}(\mathsf{RS}^2_{\alpha_\ell/\beta_\ell}(\mathsf{CT}))\right) \cdot \mathsf{sk} = \left(\mathsf{RS}_{\alpha_\ell}(\mathsf{RCB}_{q_{\mathsf{div}}}(\mathsf{CT}) \cdot \beta_\ell)\right) \cdot \mathsf{sk}.$$

Now, to complete the proof, note that

$$\alpha_\ell \left[\left(\mathsf{RS}_{\alpha_\ell}(\mathsf{RCB}_{q_{\mathsf{div}}}(\mathsf{CT}) \cdot \beta_\ell)\right) \cdot \mathsf{sk}\right]_{Q^{(\ell-1)}} - \beta_\ell \left[\left(\mathsf{RCB}_{q_{\mathsf{div}}}(\mathsf{CT})\right) \cdot \mathsf{sk}\right]_{Q^{(\ell)}}$$

has infinity norm $\le \alpha_\ell \cdot (h+1)/2$. □

D.2 Efficiency

Next, we focus on efficiency gain when applying Grafting to double-CKKS. Let $a = \lfloor \log_2(q_{\mathsf{div}}) \rceil, b = \lfloor \log_2\left(Q^{(\ell)}/Q^{(\ell-1)}\right) \rceil$ be sizes of the division prime and the rescaling factors, respectively. Here b can be chosen so that b is slightly larger than a. When comparing Mult and Mult^2, one $a+b$ bit RLWE multiplication in Mult is compared with two b bit multiplications in Mult^2. Assuming that $a+b$ bit computation is $(a+b)/b$ times more expensive than b bit computation, the throughput of Mult^2 should be asymptotically the same as that of Mult. However, actual implementations are affected by the machine word size, and in the worst case, it was estimated in [CCKS23] that the Mult^2 could be two times slower than Mult in terms of throughput.

In [CCKS23, Table 2], they provided a parameter that increases the homomorphic capacity compared to the conventional CKKS parameter using 57-bit primes. The modulus they used is consisting of two 61-bit primes, eighteen 38-bit primes, and three 23-bit primes. Using Grafting, we can use 15 unit moduli of ≈ 61 bits. This allows us to use dnum of 14 instead of 22 and the number of RNS factors of 15 instead of 23. As a result, the double-CKKS with Grafting should win by roughly a factor of $\frac{22}{14} \times \frac{23}{15} \approx 2.41$.

We note that the efficiency gain could be even more significant for lower precision or with fewer slots that require smaller division primes and rescaling factors. For instance, we may use $\log_2(q_{\mathsf{div}}) \le 10$ and $\log_2\left(Q^{(\ell)}/Q^{(\ell-1)}\right) \le 20$ for $\lesssim$ 15-bit precision, leading to a factor of ≥ 3 improvements compared to the naïve double-CKKS implementation.

E ARBITRARY PRECISION COMPUTATION

As the sizes of the RNS factors no longer limit the precision, we can easily obtain lower or higher precision computation results by changing the scale factor (and its data type). We can use smaller scale factors of 15-30 bits to achieve low-precision homomorphic multiplications and reduce the modulus consumption. For higher precisions, the only parts we should handle is the data type for the scale factors. After the messages are encoded, the homomorphic computation error is proportional to the scale factor, requiring the large scale factors. To accurately handle the scale factors for highly precise encoding and decoding, we also need the scale factors to be high-precision–and it is the only requirement for high-precision computation in Grafting! By replacing the double data type for scale factors with quadmath data type–supporting the quadruple-precision–, we could compute up to 113-bit precise homomorphic multiplications and up to 107-bit precise homomorphic linear transformations. Our benchmark reported, the speed-up of 2.20× for the standard double 53-bit precision homomorphic multiplication, using 66-bit Δ. The non-grafted counterpart is implemented using the composite-rescaling [AAB+23] in simple RNS-CKKS implementation. In Table 9, we also report the precisions obtained for SlotToCoeff and CoeffToSlot, the linear transformations for homomorphic en/decoding, for the changing scale factor sizes.

Table 9: Precision of homomorphic DFT for various scaling factor sizes with the error statistics.

Hom. DFT	$\log_2 \Delta$	Prec.	Error Average	Error Std. Dev.	Error Max
StC	20	1.93	2.37e-01	1.23e-02	2.62e-01
	40	21.95	2.14e-07	5.01e-08	2.47e-07
	60	41.97	2.00e-13	4.67e-14	2.33e-13
	80	61.89	1.97e-19	4.63e-20	2.34e-19
	100	81.82	1.97e-25	1.37e-26	2.34e-25
	120	98.94	1.42e-30	8.81e-32	1.64e-30
CtS	20	7.43	4.62e-03	5.14e-04	5.79e-03
	40	27.49	4.31e-09	1.08e-09	5.31e-09
	60	47.95	2.78e-15	7.34e-16	3.67e-15
	80	66.16	1.08e-20	5.00e-22	1.21e-20
	100	87.11	4.68e-27	1.20e-27	5.98e-27
	120	107.36	3.61e-33	9.10e-34	4.81e-33

F BOOTSTRAPPING, REVISITED

F.1 Bootstrapping with Adaptive Precision.

During EvalMod, the reduction modulo q_0 is approximated to a polynomial, composing the polynomial approximating the cosine, double angle formula, and arcsine. We focus on the cosine function, which is commonly approximated via Chebyshev approximation, e.g., a 63-degree polynomial with coefficients for the Chebyshev basis. In general, the Chebyshev coefficients rapidly decay when the degree increases, and for the cosine function, it is known that it decreases in quadratic order. We can consider sparing modulus using smaller scale factors for the higher-degree coefficients

(multiplied by a large constant to compensate) without damaging the final precision.

As an instance, we can evaluate the polynomial with a circuit that is divided into two subordinate circuits using different moduli chains: 1) compute the first half of the polynomial, i.e., for degrees 0 to 31, consuming 5 multiplicative depths with a desired precision, and 2) compute the second half, e.g., for degrees 32 to 63 with smaller coefficients consuming 6 multiplicative depths with lesser precision. As the second half requires more multiplicative depths, we can reduce the total modulus consumed during the polynomial approximation, e.g., 5 depths with $\Delta \approx 42$ bits, and 6 depths with $\Delta \approx 37$ bits in parallel, consuming the modulus of total $\approx \max(5\times 42,\ 6\times 37) = 222$ bits, instead of $6\times 42 = 252$ bits, saving 30 bits of modulus. Note that the cost of evaluating the cosine function increases slightly since the Chebyshev bases are computed in both subordinate circuits, but are by little, only about 5-10% of the EvalModcost. Note that this can be extended to general Chebyshev approximation evaluations, including other approximation methods that guarantee the coefficient decay, resulting in reduced modulus consumption.

We report the error statistics and the modulus consumption in Table 10, where 15 bits of modulus consumption can be reduced without precision loss. Note that this value can be significant when the polynomial approximation is designed friendly to the technique.

Table 10: Error statistics and modulus consumption (in bits) for grafted cosine function evaluation and its variants with reduced modulus consumption.

Chebyshev	Error			Modulus
	Average	Std.dev	Max	consum.
Original	2.44e-7	3.10e-8	3.65e-7	252
Reduced modulus consumption	2.45e-7	2.69e-8	3.13e-7	247
	2.42e-7	2.60e-8	3.33e-7	242
	2.45e-7	2.48e-8	3.15e-7	237
	1.44e-6	3.08e-8	1.53e-6	232
	1.10e-4	3.37e-8	1.10e-4	227

F.2 Bootstrapping with Flexible Output Modulus.

In scenarios where bootstrapping is required but the total modulus consumption is relatively modest[12], the bootstrapping time can be improved by lowering the ModRaise modulus.

We target the specific scenario where we want to evaluate a plaintext-ciphertext matrix multiplication, abbreviated as PCMM. It was demonstrated in [BCH+24] that evaluating the PCMM with the matrices in coefficient encoding can be done very efficiently, with one depth of available multiplications. We also note that matrix multiplication requires higher precision than usual because the messages and errors, as many as the dimensions of the matrices, are summed up. Hence, we do not need to make the full modulus available after bootstrapping but raise it to the smallest modulus, roughly a sum of the base modulus (for latter bootstrapping), and a larger-than-usual scaling factor of approximately 74 bits. In such a setting, we report bench-marked speedups of 5% and 51% in $\log_2 N = 15$ and 16, respectively. We note that the speed-up is somewhat restricted in $\log_2 N = 15$, since the maximum possible modulus is too small.

[12] Such as large matrix multiplications for machine learning, at the low levels. It is sometimes much faster to evaluate in the levels as low as possible [BCK+23, BCH+24, Par25, RKP+25, LLK+23] than to evaluate in the higher levels.

G EXPECTED SPEED-UP OF EXISTING PARAMETERS

We investigate the parameters used in the RNS-CKKS libraries and suggest their Grafted version. We list up the default or the pre-set parameters in the libraries HEaaN [Cry22], Lattigo [lat24], and OpenFHE [ABBB+22] in Table 11. All sizes are given in logarithms base-two and #mod. denotes the number of NTT primes comprising the switching key modulus $PQ_{\max}$, denoted as PQ due to space limit. The size of the RNS factors $\log q_i$ and $\log p_i$ are given with the number of moduli of that size. Moduli with fractional sizes are only partially used by the step they are allocated to, as referenced in [BMTH21]. For the FHE parameters, we additionally provide the size of the RNS factors reserved for each bootstrapping sub-procedure or general homomorphic multiplications.

In particular, the parameters for OpenFHE [ABBB+22] are automatically generated using the default setting, where parameter customization is also allowed. The default scaling factor is 59 bits, and the base modulus prime is 60 bits, which implies our Grafting technique may not be effective since all prime moduli are already set to be roughly the word size. It also supports higher precision, with a default scaling factor size of 78-bits and a base modulus prime from 89 to 105-bits. In this case, our technique reduces the inefficiency of using two RNS factors to perform 78-bit arithmetic operations.

Following the re-designing methodology introduced in Section 4.1, we provide the re-designed parameters in Table 12. Some of the parameters are expected to be accelerated well, but some are not, especially when Δ is close to the machine's word size or when dnumis set not so compatible with the mostly word-sized moduli.

Table 11: HE parameters for CKKS scheme in the literature. Here, S and F denote the SHE and FHE parameters, respectively. The RNS factors for SHE parameters are given for the base prime, the auxiliary modulus, and the rest. The RNS factors for FHE parameters are given for the base prime, the ones reserved for bootstrapping, the auxiliary modulus, and the rest.

		$\log N$	$\log PQ$	$\log \Delta$	#mod.	$\log q_i$					$\log p_i$
						Base	StC	Mult	EM	CtS	
HEaaN [Cry22]	S	13	217	41	5	47	41×3				47
		14	436	42	10	50	42×7				46×2
		15	866	42	20	48	42×14				46×5
		15	860	40	21	50	40×19				50
	F	15	771	36	17	49	33×3	36×1	49×8	47×3	54×1
		16	1,555	42	30	58	42×3	42×9	58×9	58×3	$59 \times 3 + 60 \times 2$
		17	2,070	51	40	61	51×3	51×13	51×10	51×3	53×10
Lattigo [lat24]	S	14	438	34	12	45	34×9				43×2
		15	880	40	21	50	40×17				50×3
	F	15	768	25	16	50	60	$50 + 25$	50×8	49×2	50×2
		16	1,546	40	30	60	39×3	40×9	60×8	56×4	61×5
		16	1,553	30	27	55	60×1.5	60×7.5	55×8	53×4	61×5
OpenFHE [ABBB+22]	S	14	371	50	7	60	50×5				60
		15	675	90	7	105	90×5				119
	F	16	1,579	58	27	60	58×2	58×4	58×13	58×2	60×5
		17	2,910	78	34	89	78×3	78×8	78×13	78×3	119×6

Table 12: Grafted parameters corresponding to the HE parameters in the literature. The number of RNS factors shows the changes from the original to the Grafted parameters. The scaling factor Δ is unnecessary for the Grafted parameters; however, we give them as a reference. The maximum ciphertext modulus $Q_{\max}$ is set approximately the same as the original parameters for fair comparisons. The expected speed-up ratios for addition and tensor products (Add, Tensor) and key-switching (KS) are given in the last columns. The ratio is especially small when Δ is close to the machine's word size or when $\log_2 PQ$ is already split well into word-sized moduli. An asterisk ($*$) indicates that the machine word size (or the unit arithmetic) is set to be 128 bits, expecting much larger speed-ups when using 64-bit with Grafting.

		$\log N$	$\log PQ$	$\log \Delta$	#mod.	$\log q_i$	$\log p_i$	Speed-ups	
								Add Tensor	KS
HEaaN [Cry22]	S	13	217	41	$5 \to 4$	54×3	55	1.00	1.20
		14	437	42	$10 \to 8$	$54 \times 3 + 55 \times 3$	55×2	1.14	1.33
		15	860	42	$20 \to 16$	$53 \times 4 + 54 \times 8$	54×4	1.15	1.18
		15	855	40	$21 \to 14$	61×13	62	1.43	2.05
	F	15	773	36	$17 \to 13$	$59 \times 8 + 60 \times 4$	61	1.23	1.56
		16	1,545	42	$30 \to 25$	$61 \times 5 + 62 \times 15$	62×5	1.19	1.35
		17	2,070	51	$40 \to 36$	$57 \times 15 + 58 \times 12$	$57 \times 3 + 58 \times 6$	1.07	1.08
Lattigo [lat24]	S	14	428	34	$12 \to 7$	61×6	62	1.43	1.31
		15	886	40	$21 \to 15$	59×12	$59 \times 2 + 60$	1.38	1.75
	F	15	768	25	$16 \to 13$	59×11	$59 + 60$	1.17	1.29
		16	1,546	40	$30 \to 25$	$61 \times 4 + 62 \times 16$	62×5	1.19	1.35
		16	1,553	30	$27 \to 25$	$61 \times 4 + 62 \times 16$	62×5	1.00	1.21
OpenFHE [ABBB+22]	S	14	371	50	$7 \to 6$	$61 + 62 \times 4$	62	1.00	1.14
		15	675	90	$7 \to 6$	$111 \times 4 + 112$	119	1.00^*	1.14^*
	F	16	1,558	58	$27 \to 26$	62×20	53×6	1.05	1.17
		17	2,910	78	$34 \to 24$	121×18	122×6	1.47^*	1.90^*