Grafting: Complementing RNS in CKKS

Jung Hee Cheon[1,2], Hyeongmin Choe[1], Minsik Kang[1], and Jaehyung Kim[2]

[1] Seoul National University, Seoul, Korea
{jhcheon, sixtail528, kaiser351}@snu.ac.kr
[2] CryptoLab Inc., Seoul, Korea
jaehyungkim@cryptolab.co.kr

Abstract. The RNS variant of the CKKS scheme (SAC 2018) is widely implemented due to its computational efficiency. However, the current optimized implementations of the RNS-CKKS scheme have a limitation when choosing the ciphertext modulus. It requires the scale factors to be approximately equal to a factor (or a product of factors) of the ciphertext modulus. This restriction causes inefficiency when the scale factor is not close to the power of the machine's word size, wasting the machine's computation budget.

In this paper, we solve this implementation-side issue algorithmically by introducing *Grafting*, a ciphertext modulus management system. In Grafting, we mitigate the link between the ciphertext modulus and the application-dependent scale factor. We efficiently enable rescaling by an arbitrary amount of bits by suggesting a method managing the ciphertext modulus with mostly word-sized factors. Thus, we can fully utilize the machine architecture with word-sized factors of the ciphertext modulus while keeping the application-dependent scale factors. This also leads to hardware-friendly RNS-CKKS implementation as a side effect. Furthermore, we apply our technique to Tuple-CKKS multiplication (CCS 2023), solving a restriction due to small scale factors.

Our proof-of-concept implementation shows that the overall complexity of RNS-CKKS is almost proportional to the number of coprime factors comprising the ciphertext modulus, of size smaller than the machine's word size. This results in a substantial speed-up from Grafting: 17-51% faster homomorphic multiplications and 43% faster CoeffsToSlots in bootstrapping, implemented based on the HEaaN library. We estimate that the computational gain could range up to 1.71× speed-up for the current parameters used in the RNS-CKKS libraries.

1 Introduction

The Cheon–Kim–Kim–Song (CKKS) scheme [10] is a Homomorphic Encryption (HE) scheme that allows approximate computation over real-valued, encrypted data. In CKKS, to ensure the precision of real messages, the messages are multiplied by a scale factor Δ during the message encoding procedure. Due to this scale factor, the CKKS homomorphic multiplication requires a *Rescale* procedure, which scales down the ciphertext during homomorphic multiplication from Δ^2 to Δ.

When CKKS was first introduced, the original implementation [24] used power-of-two moduli and scale factors. Thus, the Rescale operation was performed very efficiently by shifting the bits. However, for efficiency reasons, almost all existing CKKS libraries are implementing a variant based on the Residual Number System (RNS), namely the RNS-CKKS scheme [9]. In the current RNS-CKKS implementations, moduli and scale factors are not power-of-two. The modulus of a ciphertext is usually composed of different prime moduli, which we call *RNS Moduli*. Thanks to the Chinese Reminder Theorem (CRT), computations in a polynomial ring modulo the ciphertext modulus (i.e. $\mathbb{Z}_Q[x]/(x^N+1)$ for some integer $N > 0$ and ciphertext modulus Q) can be separated into parallel computations over the polynomial ring modulo each RNS modulus (i.e. $\mathbb{Z}_{q_i}[x]/(x^N+1)$ for each RNS modulus q_i, satisfying $Q = \prod q_i$). The Number Theoretic Transform (NTT) further enables efficient multiplication (over each polynomial ring modulo RNS modulus) if the RNS moduli are NTT primes smaller than a machine word size. Likewise, RNS accelerates the overall homomorphic computations; however, the composite modulus restricts the scale factor to be approximately equal to one of the RNS moduli, that is, $\Delta \approx q_i$ (or a product of some q_i's). It makes the Rescale process erroneous and inefficient.

As the ciphertexts are represented, computed, and stored in the RNS with respect to each RNS modulus, the efficiency of homomorphic operations is proportional to the number of RNS moduli constituting the ciphertext modulus unless the moduli are much smaller than the machine word size. Thus, reducing the number of RNS moduli when designing HE parameters highly impacts the latency of homomorphic computations. In this regard, it is desirable to have the RNS moduli as close to the machine word size as possible. However, this is not the case in most of the RNS-CKKS implementations due to other efficiency reasons.

RNS Moduli in RNS-CKKS Implementations. The choice of RNS moduli depends on the specific circuit to evaluate and message precision, and sometimes, the RNS moduli are much smaller than the machine word size. As the ciphertext modulus is rescaled after each multiplication, the modulus decreases as multiplication is repeated. The ciphertext modulus needs to be large enough for a given multiplicative circuit depth, but it also cannot exceed a certain bound for IND-CPA security, which limits the allowed multiplicative depth. Therefore, having the smallest possible scale factor is desirable, as it would reduce each RNS modulus and allow deeper-depth homomorphic multiplications. Indeed, CKKS bootstrapping unlocks the limit by increasing the ciphertext modulus, but the figure remains the same due to its heavy cost. Bootstrapping is one of the most expensive homomorphic operations, costing hundreds of times more than homomorphic multiplication and tens of thousands of times more than homomorphic addition. Hence, having the smallest possible scale factor is even more efficient for deep-depth circuits.

In this regard, the scale factors are set as small as possible but still larger than the desired precision with an additional margin for errors. The resulting

RNS moduli is thus smaller than the machine word size, depending on the desired precision. For instance, in the two state-of-the-art RNS-CKKS libraries Lattigo [1] and HEaaN [12], the default IND-CPA-secure parameter has a scale factor Δ ranges from $\approx 2^{34}$ to $\approx 2^{51}$ depending on the desired precision, which is way smaller than the default 64-bit machine word size. For higher message precisions, which is required for security against the IND-CPA^D attacks [21], $\Delta \approx 2^{45} \cdot 2^{45}$ is used in Lattigo. Other libraries such as OpenFHE [3] or SEAL [23] use scale factor ranges from $\approx 2^{48}$ to $\approx 2^{55}$ for IND-CPA security, or $\approx 2^{78}$ to $\approx 2^{90}$ for IND-CPA^D security. The RNS moduli used for homomorphic multiplications must have similar sizes to the scale factor (or square root of the scale factor) and are mostly smaller than the machine word size. One exception is IND-CPA^D-secure OpenFHE parameters, which use primes larger than the machine word size, which is, in general, less efficient than using a composition of smaller primes.

The discrepancy between the ideal word-size RNS moduli and the smaller moduli commonly used in most of the RNS-CKKS implementations is due to the linkage between the RNS moduli and the *Rescaling Factor*. The rescaling factor, i.e. the amount scaled during the CKKS Rescale procedure, is set to be one of the RNS moduli and should be similar to the scale factor Δ. However, if we can Rescale by a factor independent of the RNS moduli, then we can set the RNS moduli close to the machine word size and speed up the whole homomorphic operations as desired. For example, the Lattigo library [1] default parameter for Somewhat Homomorphic Encryption (SHE) uses 12 different RNS moduli of sizes $\approx 2^{34}$ or $\approx 2^{45}$, and a total modulus of $\approx 2^{438}$. This could be modified to use 7 ($\gtrsim 438/62$) RNS moduli of size $\approx 2^{62}$, resulting in a naïve speed-up of a factor of $12/7 = 1.71$ in a 64-bit machine.[3]

The linkage also makes it harder to optimize the RNS-CKKS implementation in several aspects. One needs to consider complex conditions for optimized performance for a parameter set targeting a specific message precision and a circuit. For instance, to set the RNS moduli, one should consider not only the desired precision and the circuits but also the rank of the gadget decomposition and how the decomposition will be set, separating the ciphertext and the auxiliary moduli. Another limitation is the lack of NTT primes with small sizes - the ring dimension N is usually set from 2^{12} to 2^{17} and NTT primes should be 1 modulo $2N$. The limited number of NTT primes makes it hard to optimize the parameters for low-precision computations or for advanced techniques using smaller rescaling factors such as Tuple-CKKS [8]. Such dependencies are not so desired for hardware implementations, of which the low-level design may be changed depending on the parameters. The non-word size RNS moduli also affect the memory usage, making it larger than the theoretical expectation, e.g., a ciphertext size of $2nN \log w$ rather than $2N \log Q_{\mathrm{ctxt}}$, where n is the number of RNS moduli and w is the machine word size noting that $w^n \geq Q_{\mathrm{ctxt}}$.

[3] Note that a margin of 1-3 bits is required for NTT, and the maximum possible size for RNS modulus is $\approx 2^{62}$.

1.1 Our Contribution

In this work, we focus on separating the computational modulus, a product of RNS moduli, from the rescaling factor to fully utilize the efficiency gains from the machine word size RNS moduli. In this context, we outline the ciphertext modulus requirements to achieve computation and space efficiency.

- *Mostly word size RNS moduli:* As per the benefits of using word-size RNS moduli, it is desirable to use the word-size moduli as much as possible and reduce the number of RNS moduli.
- *Flexible rescaling factor:* To completely decouple the linkage between the RNS moduli and the rescaling factor, a flexible and independent choice of rescaling factors is required.
- *Ciphertext and key modulus:* Before key switching, one of the most frequently used homomorphic operations, the ciphertext modulus Q_{ctxt} must divide the modulus of a public switching key Q_{key}. Otherwise, additional keys are required, often increasing communication costs by orders of magnitude.[4]

We solve this problem by introducing a novel modulus management system called *Grafting*. Grafting manages the RNS moduli composing the ciphertext modulus (which decreases as homomorphic multiplication proceeds) to make the modulus satisfy the above-mentioned requirements. Furthermore, Grafting can be utilized with various state-of-the-art techniques for RNS-CKKS implementation.

Technical Details. As a ciphertext modulus is not a multiple of rescaling factors anymore, we utilize the modulus switching technique from BGV/BFV between two integer moduli that are not a divisor and a multiple of each other to introduce the rescaling factor to the ciphertext modulus. However, the modulus switching requires many NTT conversions, which is the major source of the heavy computational cost of homomorphic multiplication, and it can dilute the efficiency gains from the word-size RNS moduli.

Rational Rescale. We extensively study the potential points during homomorphic multiplication where modulus switching can be applied. Due to the substantial cost of NTT conversion, viable options are the points where the NTT conversion is already needed, e.g. before or after ModUp, ModDown, and Rescale operations.[5] We conclude that the Rescale operation is the best among these options and introduce *Rational Rescale*, a Rescale procedure fused with the modulus switching.

Rational Rescale (or $\mathbb{Q}$-Rescale in short) can replace the original Rescale operation without any significant impact on the computational cost and the

[4] The switching keys should be generated by the secret key owner for each ciphertext modulus if the set of ciphertext moduli does not form a sequence where each term is a multiple of the preceding ones.

[5] Indeed, there could be other options, e.g. directly switching the public switching key modulus, but with a huge computation error or high memory/computation cost.

error when the ciphertext modulus does not have any factor approximately the size of the desired rescaling factor. That is, $\mathbb{Q}$-Rescale maps a ciphertext with a pair of ciphertext modulus and scale factor (Q, Δ^2) into a ciphertext with $(Q', \Delta') \approx (Q/\Delta, \Delta)$, where $\Delta \nmid Q$. To do so, we first multiply the ciphertext by $Q' \approx Q/\Delta$ and get an intermediate state $(Q \cdot Q', \Delta^2 \cdot Q')$. Then we Rescale by a rescaling factor Q, end up with $\Delta' = \Delta^2 \cdot Q'/Q \approx \Delta$.

Modulus Resurrection with Universal Rescalability. $\mathbb{Q}$-Rescale allows us to Rescale a ciphertext by an arbitrary rescaling factor, satisfying the second requirement for ciphertext modulus. However, the $\mathbb{Q}$-Rescale introduces new factors that are not divisors of the input ciphertext modulus if the rescaling factors are smaller than the machine word size. The new factors will make the ciphertext modulus Q_{ctxt} not to divide the public switching key modulus Q_{key} unless they are managed appropriately. Moreover, they should be managed not to pile up; otherwise, it will increase the number of non-word size RNS moduli.

We introduce the *Modulus Resurrection* technique to fulfill the first and third requirements while utilizing the $\mathbb{Q}$-Rescale operation. Modulus resurrection is a method of selecting and managing the RNS moduli that forms the modulus of public switching keys Q_{key}. It re-introduces the factors that were a part of the RNS moduli of a larger ciphertext modulus and the switching key modulus, but which have already been eliminated in the current ciphertext modulus. By reusing the same factors, no new factors are piled up. We repeatedly eliminate and resurrect the part through ($\mathbb{Q}$-)Rescales and modulus resurrections. We refer to this part as a *Sprout*, which emerges after each $\mathbb{Q}$-Rescale absorbs a word-sized RNS modulus as a nutrient. The sprout is a product of several small primes and is of, a multiple of machine word size.

In addition, to guarantee that the modulus resurrection is always possible, we define a corresponding property, namely the *Universal Rescalability*. We call a sprout *a universal sprout* if the size of its divisors covers all positive integers less than the machine word size. We prove the universal rescalability of modulus resurrection utilizing the universal sprout and provide possible choices of universal sprout. For instance, 2^{15}, a prime $2^{16} + 1$, and a 30-bit prime can constitute a universal sprout in 64-bit machines, replacing a 61-bit RNS modulus if needed; and a product of 2^{15} and $2^{16} + 1$ can be a universal sprout in 32-bit machines. A power-of-two integer 2^{61} serves as another universal sprout for 64-bit machines, potentially bridging the original CKKS [10] and RNS-CKKS [9] implementations.

Gadget Resurrection and Grafting. The modulus resurrection with universal rescalability already meets the above-mentioned requirements. However, it may be less efficient in certain but limited cases when employing the gadget decomposition technique for key switching and the ciphertext modulus is small. When there are two gadget blocks[6] that are partially consumed, the two blocks may participate

[6] For gadget decomposition, the RNS moduli are grouped into several sets, say gadget blocks. Once the RNS moduli within a gadget block are completely consumed, the block is no longer participating in the key-switching operation. The running time of

in the key-switching procedure instead of one. To prevent such inefficiency, we maintain the number of partially consumed gadget blocks to be less or equal to one, by resurrecting a gadget block that contains the sprout instead of consuming another gadget block by part. We call this procedure the *Gadget Resurrection*. As a result, the gadget block containing the sprout is the only partially consumed block, which complements the *Grafting: ciphertext modulus management system consisting of moduli and gadget resurrections with universal rescalability*.

When Grafting is applied, the 'level' of a ciphertext is unnecessary. Instead, we only consider its ciphertext modulus, and its scale factor. Having this, the RNS moduli of a ciphertext is determined. Additions and multiplications between two ciphertexts with different moduli can be made accordingly via adapting the so-called *level adjustment technique* [18].

Experimental Results and Estimations. We experimentally verified the theoretical efficiency gains of Grafting with several proof-of-concept implementations based on HEaaN [12] library and compared the result with the prior implementation.

Faster Homomorphic Linear Transforms. We may have a simplified version of Grafting in linear transformations that does not require ciphertext-ciphertext multiplication. That is, we may switch the modulus of a ciphertext to a modulus that is a composition of mostly the word size RNS moduli. We perform the linear transformations without the Rescale operation, then switch back to the ciphertext modulus that should have been obtained through the original linear transformation, by utilizing the $\mathbb{Q}$-Rescale operation. This should give a factor that is close to the word size ratio, introducing negligible overheads.

A crucial application of this simplified Grafting is the CKKS bootstrapping, where the significant cost arises from the homomorphic linear transformations. We checked the impact with experiments detailed in Section 4.1, showing a factor $1.43\times$ improvement over the FTa parameter of HEaaN library [12], despite a larger gadget rank (See Table 1 for the RNS moduli we used for the comparison).

Concrete Parameters and Implementations. We revisit the parameters used in HE libraries supporting the CKKS scheme. Since the RNS moduli in their parameters vary from 20 to 60 bits, the amount that Grafting can accelerate the homomorphic computations also varies. For each parameter, we construct a corresponding concrete parameter that fully utilizes the 64-bit machine word size and compare the estimated costs of homomorphic operations. Assuming that NTT is the dominant cost of homomorphic operations, we could have at most a factor two in terms of computation cost. Depending on parameters, we expect 1.04-$1.72\times$ improvements compared to the usual RNS-CKKS homomorphic operations.

---

the key-switching operation is highly affected by the number of participating gadget blocks; thus, having two partially-consumed gadget blocks may be less efficient than having one.

	$\log q_i$					$\log p_i$
	Base	StC	Mult	EvalMod	CtS	
HEaaN.FTa	38	$32 + 28 \times 2$	28×5	38×8	41×3	42×2
Proposed	59×10				$61 + 62$	62

Table 1. Size of RNS moduli for comparison, borrowed from Table 3. Columns corresponding to $\log q_i$ and $\log p_i$ refer to the sizes (in logarithm two) times the number of RNS moduli in the ciphertext and auxiliary modulus, respectively. Columns with labels Base, StC, EvalMod, and CtS show the moduli reserved for corresponding steps of bootstrapping, where Mult for homomorphic multiplications.

In addition, we provide a proof-of-concept implementation in Section 4.2 for Grafting that contains the main ingredients such as rational rescaling, modulus resurrection, and gadget resurrection. Depending on levels, we observe 1.17-1.51× improvements compared to the usual CKKS multiplications.

Applications to Tuple-CKKS. In [8], they proposed a novel CKKS multiplication, namely Mult^t, that asymptotically maintains throughput while greatly improving latency and memory footprint. However, these asymptotics rely on the fact that the cost of k-bits arithmetic is proportional to k, which is not the case in reality, as illustrated in the previous sections, and due to the lack of small NTT primes. Grafting fully utilizes the machine word size, leading to the expected performance as mentioned in [8]. Depending on the precision and number of slots, the improvement can reach a factor 2 to 3.

1.2 Related Works

After the first RNS-CKKS scheme was introduced [9], most of the currently available homomorphic encryption libraries implementing CKKS scheme only focus on the RNS version of it due to its efficiency [1, 3, 12, 15, 23]. The ciphertexts and the switching keys are decomposed into a small integer modulo each RNS moduli, and the computation is done in the decomposed format, which is homomorphic thanks to the Chinese Remainder Theorem (CRT). From its RNS-friendly nature, Han and Ki [17] introduced a key-switching technique to RNS-CKKS, adopted from [13, 5], to trade-off between the usable ciphertext modulus and running time of the key switching algorithm (including the size of the switching keys), so-called RNS gadget decomposition of the switching key. By decomposing the key and a ciphertext into several pieces, one can perform KeySwitch by separately multiplying and combining them.

As the decomposed blocks for each RNS moduli are the units constituting the homomorphic computations in RNS-CKKS, in some libraries such as Lattigo [1], OpenFHE [3], and HEaaN [12] or in the recent works accelerating and improving the RNS-CKKS implementations [20, 18, 7] tried to elaborately design the ciphertext modulus and the switching key modulus, at least for some possible parameters set. However, as mentioned earlier, in the existence of the Rescale

algorithm in the CKKS scheme, the RNS moduli should be the rescaling moduli of which the size is determined depending on the circuits to be evaluated. Thus, it is hard to fully utilize the budgets from the machine word size.

Approaches Filling the Machine Word Sizes. The idea to use word-size primes mostly and a few small primes in the RNS moduli was first proposed by Gentry, Halevi, and Smart [14] without details. In BGV [16], 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-size primes and dropping non-word-size primes, which maintains the message the same. In doing so, the ciphertext modulus Q is set to be the product of word-size primes, inducing little loss in the ciphertext capacity, i.e., the quantity Q/ϵ, where ϵ is the error bound. However, this approach is not applicable to CKKS since the message is not well preserved as in BGV, inducing an absolutely larger error during modulus-switching, unless the input/output ciphertext moduli are extremely close. We note that for BFV, the ciphertext modulus during key-switching is fixed and independent of the given circuits. Hence, the RNS moduli are usually chosen as large as the size of the machine words.

When focusing on CKKS, recently Mono et al. [22] adopted the nature of using mostly word size primes in [16] from RNS-BGV to RNS-CKKS. They constitute the RNS moduli with mostly word-size primes, and a few small primes of the same size, whose product is close to the multiple of the word-size primes, e.g., two word-size 54-bit primes can be replaced by three smaller primes of 36-bit. Then, they continuously rescale by 36-bit, performing modulus switching from two word-size 54-bit primes into three smaller 36-bit primes if necessary. This can be seen as a type of modulus resurrection; however, in their method, scaling is only possible with either word size or fixed small-size primes. Thus, the composition of the RNS moduli needs to be adjusted according to the given circuit and is not compatible with the practically chosen parameters in the libraries implementing RNS-CKKS.

A recent study on the double-gadget-based key switching technique [19] partly solves this problem but focuses on the KeySwitch operation. The key-switching operations can fully utilize the machine word size budgets by embedding the $\mathcal{R}$-module computations into $\mathcal{R}_B$ computations where B is composed of the word-size RNS moduli. However, their technique is advantageous only for sufficiently large gadget ranks and is applicable only to the KeySwitch operation, while ours can accelerate the whole homomorphic computations with no restrictions on gadget ranks. A following work by Belorgey et al. [6] extended this double-gadget technique to digit-based gadget decompositions and proposed to implement CKKS using binary arithmetic computations and Discrete Fourier Transform (DFT). We remark that the approach toward using binary modulus CKKS is similarly allowing rescaling by arbitrary bits, which is one of our results, decoupling the computations moduli and the rescaling moduli; however, they are in the opposite direction, using even smaller power-of-two moduli.

Machine-dependent Approaches. An implementation depending on the machine word size is also studied by Agrawal et al. [2] to design a 32-bit hardware implementation with the RNS-CKKS parameters. The rescaling factors range from 48-58 bits, and two 24-29 bits of NTT primes are required for each Rescale operation. It is worth noting that the lower moduli are hard to use since there are fewer NTT primes having smaller sizes, such as 24 to 29 bits, which was also the case in Tuple-CKKS [8].

Paper Organization. In Section 2, we introduce the RNS-CKKS scheme with useful notations used throughout this paper. In Section 3, we introduce our main algorithm Grafting, which consists of rational rescale, modulus resurrection, and gadget resurrection. The universal rescalability theorem and the correctness theorems with the error bound of each suggested operation are given as well. In Section 4, several experimental results will be presented regarding RNS-CKKS implementation using Grafting. Furthermore, we will analyze the current parameters of existing CKKS libraries and present the expected acceleration with the Grafting technique. Section 5 will focus on an application of Grafting to tuple-CKKS [8], and we conclude in Section 6.

2 Preliminary

2.1 Notations

Polynomials are denoted in bold font and lower case 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 equal to or smaller than n as $[n]$, i.e., $[n] = \{0, 1, \cdots, n\}$.

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 $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R} = Z_Q[x]/(x^N + 1)$. We let $\mathsf{ct} = (\mathbf{b}, \mathbf{a}) \in \mathcal{R}_Q^2$ be a ciphertext with ciphertext modulus Q with respect to a secret key $\mathbf{s} \in \mathcal{R}$ if it satisfies $\langle \mathsf{ct}, (1, \mathbf{s}) \rangle \approx \mathbf{m} \mod Q$ for some message $\mathbf{m} \in \mathcal{R}$ with respect to a target message precision.

2.2 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 $NTT(\mathbf{a}) = (\mathbf{a}(\zeta^0), \mathbf{a}(\zeta^1), \cdots, \mathbf{a}(\zeta^{N-1})) \in \mathbb{Z}_q^N$ 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 $iNTT(\mathbf{b}) = \sum_{i=0}^{N-1} \tilde{b_i} x^i$ 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.,

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

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 $NTT(\mathbf{a}) \in \mathbb{Z}_q^N$ be an *NTT-evaluated* format of $\mathbf{a}$.

2.3 Computation in RNS-CKKS

RNS-CKKS scheme is an RNS variant of the CKKS scheme, which was first introduced in [9]. The ciphertext and the switching key modulus are composed of NTT primes, which also constitute the RNS moduli. 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 $\ell \in [L]$. 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 modulus $q_i | Q$. We note that the CRT decomposition is homomorphic.

We now recall some main features of the RNS-CKKS scheme.

Fast Basis Conversion in [9]. Let $\mathcal{B} = \{p_0, \ldots, p_{k-1}\}$ and $\mathcal{C} = \{q_0, \ldots, q_{l-1}\}$ 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

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

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

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

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 $|\tilde{\mathbf{a}}| \le (\ell/2) \cdot Q$.

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. $\mathsf{ModDown}$ 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 $\mathsf{ModDown}$ but is rescaled by fewer moduli factors than $\mathsf{ModDown}$. It reduces the size of the polynomial and the modulus by q_ℓ. Let us borrow the notations from the previous Section and let $\mathcal{D} = \mathcal{B} \cup \mathcal{C}$.

Precisely,

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

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

We note that the ModUp operation maps $\mathbf{a} \in \mathcal{R}_Q$ to $\mathbf{a} + Q\mathbf{e} \in \mathcal{R}_{PQ}$, where $|\mathbf{e}| \leq \ell/2$ from the fast basis conversion. The $\mathsf{ModDown}$ operation maps $\mathbf{a} \in \mathcal{R}_{PQ}$ 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 $|\tilde{\mathbf{a}}| \leq (k/2) \cdot P$, resulting $|\mathbf{a}' - P^{-1} \cdot \mathbf{a}| = P^{-1} \cdot |\tilde{\mathbf{a}}| \leq k/2$. RS introduces an error of size $\leq 1/2$. When applied to ciphertext, the error becomes multiplied by the secret key $\mathbf{s}$ and thus has an infinity norm of $\leq k/2 \cdot (\|\mathbf{s}\|_1 + 1)$ and $\leq 1/2 \cdot (\|\mathbf{s}\|_1 + 1)$, respectively, for $\mathsf{ModDown}$ and RS.

We additionally define the $\mathsf{Inv\text{-}RS}$ 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.

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

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

or in the RNS representation, one can write as:

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

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.

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 can be decided based on the estimated attack costs of the known attacks via Lattice estimator [4]. The switching key is generated in modulus PQ, possibly using gadget decomposition. The RNS gadgets Q_i are composed of the RNS moduli and $Q_{\max} = Q_0 \cdots Q_{\mathsf{dnum}-1} = (q_0 \cdots q_{\alpha-1}) \cdot (q_\alpha \cdots q_{2\alpha-1}) \cdots (q_{\alpha(\mathsf{dnum}-1)} \cdots q_{\alpha\mathsf{dnum}-1})$, where $L + 1 = \alpha \cdot \mathsf{dnum}$ and $\mathsf{dnum} \in \mathbb{N}$. Since $P \geq Q_i$ should hold for all i, the maximum possible modulus

PQ should be split into P and Q with roughly $P \approx Q^{1/\mathsf{dnum}}$. Let $P = p_0 \cdots p_{K-1}$. The switching keys are $\{\mathsf{swk}_i\}_{i\in[\mathsf{dnum}-1]} = \{(\boldsymbol{\beta}_i, \boldsymbol{\alpha}_i)\}_{i\in[\mathsf{dnum}-1]} \in \mathcal{R}_{PQ}^{2\times\mathsf{dnum}}$, where

$$\boldsymbol{\beta}_i = -\boldsymbol{\alpha}_i \cdot \mathbf{s} + P \cdot \hat{Q}_i \mathbf{s}' + \mathbf{e}_i \in \mathcal{R}_{PQ},$$

where $\hat{Q}_i = Q_{\max}/Q_i$, 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 due to a larger switching key size.

Key switching. Key switching consists of the following procedures for *gadget rank* d:

1. Inv-RS the ciphertext from modulus $Q = q_0 \cdots q_\ell$ to $Q_0 \cdots Q_{d-1}$, where $\alpha(d-1) \le \ell < \alpha d$ for some integer α, and perform RNS decompose.
2. ModUp the ciphertext in each modulus Q_i to $PQ_0 \cdots Q_{d-1}$ for $i \in [d-1]$, resulting in d ciphertexts in modulus $PQ_0 \cdots Q_{d-1}$.
3. External product a part of each ciphertext[7] with swk_i for $i \in [d-1]$, and add the remaining parts of the ciphertext.[8]
4. ModDown from modulus $PQ_0 \cdots Q_{d-1}$ to Q.

Note, from [17], the key switching procedure requires $2(\ell+1) + (d+2)\cdot(K + \alpha\cdot d)$ (i)NTTs. We remark that the Inv-RS procedure is indeed unnecessary since the padded zeros are multiplied by the switching keys and later added, which makes no difference in the key switching, technically.

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, then RS by q_ℓ. Note, the tensor of ciphertexts ct_1 and ct_2 satisfies $\langle \mathsf{ct}_1 \otimes \mathsf{ct}_2, (1, s, s^2)\rangle \equiv \langle \mathsf{ct}_1, (1, s)\rangle \cdot \langle \mathsf{ct}_2, (1, s)\rangle \mod Q$, where $\mathsf{sk} = (1, s)$. The overall cost is similar to the key switching, requiring $3(\ell+1) + (d+1)\cdot(K + \alpha\cdot d)$ number of (i)NTTs.

Level Adjustment. In the RNS setting, we use a ciphertext modulus $Q = q_0 \cdots q_\ell$ to ease the RS operation, i.e., each rescaling is done by q_ℓ, and thus we let ℓ be the ciphertext level. However, the rescaling factor q_ℓ and the scaling factor Δ have a gap, which yields an additional error after a series of rescaling. In [9], it is suggested that choosing each modulus q_i as close as possible to Δ minimizes this error, and later in [18] the authors suggest using level-specific scale 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

[7] Specifically, '$\mathbf{a}$' part of each d ciphertexts $(\mathbf{b}, \mathbf{a})$ is multiplied.

[8] Note, only the part that will be multiplied by the switching keys can be ModUp and the remaining parts can be added after ModDown; however, the overall (i)NTT cost is the same and the total cost is similar.

may need to manipulate input ciphertexts to have the same level and the same scaling factor before the homomorphic operations.

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 $\mathsf{ct} = [\mathsf{ct}]_{q_0 \cdots q_{\ell'+1}} \in \mathcal{R}^2_{q_0 \cdots q_{\ell'+1}}$ by dropping the RNS moduli $\{q_{\ell'+1}, \ldots, q_\ell\}$,
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 scale factor $\Delta_{\ell'}$ with an additional error, which is approximately a rounding error.

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 moduli, and each RNS block is computed with the machine's word size, e.g., $\lesssim 32$ bits on GPU or $\lesssim 64$ bits on CPU. Therefore, if we can reduce the number of RNS blocks representing the ciphertext, the whole FHE computations will benefit from a straightforward speed-up of the reduced ratio, for e.g., the number of NTT/iNTT conversions, fast basis conversions, tensor products, and external products. An appealing approach is to use the word size primes for the RNS moduli. However, one cannot use the moduli to be the word size since the moduli are set approximately the same as the scaling factor Δ, which varies (from 30 to 120 in general) on the target message precision.

In this section, we introduce *Grafting*, a method of using word-size primes for RNS moduli. Grafting reduces the wasted spaces that existed in all of the previous approaches, as far as we are aware. We first introduce a tool for switching the ciphertext modulus, namely, *Rational Rescaling*, which is a counterpart of the modulus switching in BGV/FV to RNS-CKKS, in the sense that the modulus is changed into a modulus that is not a divisor or a multiple of the previous modulus while preserving the message. We then introduce how to maintain the modulus of ciphertexts, which is filled with word-size NTT primes, as much as possible, namely, *resurrections of modulus and gadget* in Sections 3.2 and 3.3, respectively. Finally, we introduce how to adjust the different moduli of the ciphertexts when a modulus is not divisible by the other modulus in Section 3.4 and how to use power-of-two moduli in RNS in Section 3.5, both in the Grafting scenario.

3.1 Rational Rescaling: Rescaling by a non-divisor rescale factor

Modulus switching allows us to change the modulus from a modulus composed of word-size NTT primes to a modulus divisible by (approximately) a rescaling

factor. However, this was not used in any of the RNS-CKKS constructions, implementations, and applications. It is possible that the reason behind selecting the ciphertext modulus in RNS-CKKS from the moduli chain and rescaling it from the top modulus to the bottom modulus is not only to ensure efficient rescaling but also to make ModUp operation cheaper.

More precisely, if a ciphertext modulus is not a divisor of a switching key modulus, it is costly to switch the modulus of the ciphertext to be a divisor. As modulus switching can only be applied to the NTT-coefficient, we first need to apply iNTT (inverse-NTT) transforms on the tensored ciphertext, then switch the ciphertext modulus, followed by ModUp operation. One may think that the ModUp operation also requires the iNTT, so the number of required iNTT remains the same. This is true, but the number of iNTT and NTT increases due to an interesting fact: ModUp operation requires the same number of (i)NTT for NTT-coefficient inputs and for NTT-evaluated inputs since some NTT-coefficient inputs could be reused[9]. Reversing the order, i.e., first ModUp, then modulus switching, also requires the same amount of additional iNTT costs.

To avoid costly modulus switching before or after the ModUp operation, one may use switching keys specific to (possibly) all the ciphertext moduli, which introduces a huge amount of switching key sizes.

As a result, it is indeed efficient to keep the ciphertext modulus to be a divisor of the switching key modulus, before key switching procedure. To this end, we moved the moment of modulus switching to the rescaling procedure. Before ModDown (or RS), the ciphertext exists in NTT-coefficient format; thus, no additional (i)NTT cost is introduced from the modulus switching. We, therefore, suggest modulus switching and then rescaling as a new rescaling tool, which can (even) be done from a word size modulus that has no rescale factor as a divisor. This can also be seen as rescaling not by an integer but by a rational number. That is, one wants to rescale the ciphertext from a modulus Q to a modulus $Q' \approx Q/\Delta$, and the message from Δ^2 to Δ, where Δ is the scale factor. The *Rational Rescaling* rescales the modulus Q by a rational number $Q/Q' \in \mathbb{Q}$ to $Q/(Q/Q') = Q' \approx Q/\Delta$, which can be seen as a type of modulus switching.

This can be done easily and directly when the ciphertext is embedded in $\mathcal{R} \subset \mathcal{R}_{\mathbb{Q}}$, with roundings. But in $\mathcal{R}_Q$, the multiplication by a rational number can not be directly defined, as $\mathcal{R}_Q$ is not an $\mathbb{Q}$-module; even worse when it is decomposed into RNS. In that case, we detour the rational rescaling by first multiplying Q', then dividing by Q. During the multiplication, the ciphertext modulus is temporarily moved to QQ' and returns back after the division. Indeed, this is a much closer way of computing $\lfloor Q'/Q \cdot \mathsf{ct} \rceil = \lfloor Q' \cdot \mathsf{ct}/Q \rceil$ more accurately in machines.

[9] From NTT-evaluated $\mathsf{ct} \in \mathcal{R}_Q$, we first iNTT for Q, apply fast basis conversion, then NTT for P to make $\mathsf{ModUp}(\mathsf{ct}) \in \mathcal{R}_{PQ}$ by reusing the Q parts, on the other hand, from NTT-coefficient $\mathsf{ct} \in \mathcal{R}_Q$, we first apply fast basis conversion, then NTT for PQ.

Definition 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* $S = \mathrm{lcm}(Q, Q')/Q' \in \mathbb{Z}$*.*

We abuse the RS notation to also denote the rational rescale by denoting the rational rescale factor in the subscript. If needed, we will call the original rescale procedure using integer rescale factors as integral rescale. The rational rescaling can be naturally extended to a vector of polynomials or ciphertexts.

We note that inverse rescale during rational rescale has a totally different role compared to the one during key switching.

In the previous RNS-CKKS schemes and implementations, ModUp and $\mathsf{ModDown}$ are used to switch the modulus between multiples and divisors. The fast basis conversion was used as a subroutine to switch the modulus between non-divisor moduli. However, the role of the fast basis conversion was to make a new ciphertext that has the same ciphertext in a different modulus. This was done by approximating each RNS block, but introducing an additional error, a small multiple of the original modulus. Since the error has a large size compared to the message included in the ciphertext, the fast basis conversion can only be used in restricted cases, such as in ModUp, $\mathsf{ModDown}$, and RS. Rational rescaling also uses the fast basis conversion as a subroutine when rescaling from $\mathrm{lcm}(Q, Q')$ to Q'.

The following theorem shows that the rational rescaling error is the same type as $\mathsf{ModDown}$ and RS errors, which is linear on the number of eliminated RNS blocks, say, $\mathrm{lcm}(Q, Q')/Q'$, P, and q_ℓ, respectively. However, it is worth minimizing the RNS moduli factors of $\mathrm{lcm}(Q, Q')/Q'$ to reduce the error. The proof of the theorem can be found in Appendix A.1.

Theorem 1 (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* $\|e_{res}\|_\infty \leq \ell/2 \cdot (\|s\|_1 + 1)$*, 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 scale factor Δ^2 (after tensor) to a ciphertext whose message has a scale factor $\Delta^2 \cdot Q'/Q \approx \Delta$, instead of $\Delta^2/q_\ell \approx \Delta$. We also note that the rational scale factor can be tracked as in [18], which will be treated in Section 3.4.

3.2 Modulus Resurrection: Rescaling in the word-size moduli chain

Rational rescaling allows us to rescale with a non-divisor of the current ciphertext modulus during homomorphic multiplications. However, to continue homomorphic computations on the resulting ciphertext, one should make the ciphertext modulus

a divisor of the switching key modulus again. To enable this, we reuse some factors of the switching key modulus and the ciphertext modulus. As a primary condition for the RNS moduli is being relatively prime, we resurrect some factors of the top ciphertext modulus, which was scaled out earlier. By doing so, the RNS moduli remain relatively prime, while the ciphertext modulus is a divisor of the switching key modulus. We call this procedure as *Modulus Resurrection* specified below. We focus only on differences between RNS-CKKS [17] and ours. The other parts that are not mentioned below are the same as [17] but using our modifications consequently.

$\underline{\mathsf{Setup}^*(N, h, \eta, \mathsf{dnum} = 1, 1^\lambda)}$: For given ring dimension N, the secret key hamming weight h, and the security parameter λ, the maximum possible modulus size is set. Each ciphertext modulus is a composition of distinct word-size NTT primes, which we call *unit moduli*, and a small and flexible modulus, which we call *sprout*[10]. Specifically, the maximum ciphertext modulus is $Q_{\max} = q_0 \cdots q_{L-1} \cdot r_{\text{top}}$ and each ciphertext modulus is $Q = q_0 \cdots q_{\ell-1} \cdot r$, where q_i's be unit moduli in $2^w \cdot [1 - 2^{-\eta}, 1 + 2^{-\eta}]$, r be a sprout modulus, and r_{top} be a common multiple of all the possible sprouts, which indeed can be the maximum possible sprout. The switching key modulus is set to $PQ_{\max}$, where P is chosen to satisfy $P = p_0 \cdots p_{K-1}$, where p_i are relatively prime unit moduli, P and $Q_{\max}$ are relatively prime, $P \geq Q_{\max}$, and $PQ_{\max}$ is smaller than the maximum possible modulus size.

$\underline{\mathsf{SwkGen}^*(\mathbf{s}_1, \mathbf{s}_2, \mathsf{dnum} = 1)}$: For given secret keys $\mathbf{s}_1$ and $\mathbf{s}_2$, and $\mathsf{dnum} = 1$, it generates and outputs a switching key

$$\mathsf{swk} = (b = a \cdot s_2 + P s_1 + e, a) \in \mathcal{R}_{PQ_{\max}}.$$

$\underline{\mathsf{RS}^*(\mathsf{ct}, \delta)}$: For given a ciphertext ct over $\mathcal{R}_Q$, where $Q = q_0 \cdots q_{\ell-1} \cdot r$, if there is a divisor q of Q which is $\approx 2^\delta$, it outputs $\mathsf{RS}_q(\mathsf{ct})$. Else, it outputs $\mathsf{RS}_{q_{\ell'} \cdots q_{\ell-1} r / r'}$, where $\ell' = \lfloor (w\ell + \lfloor \log_2 r \rceil - \delta)/w \rfloor$, and r' is a sprout $\delta - w(\ell - \ell')$ bits smaller than r.

Please find how close the divisor q of Q in RS^* should be to 2^δ in Theorem 5.

This can be seen as a resurrection of the topmost ciphertext modulus, which was supposed to have been eliminated. We note that only a few unit moduli are eliminated during RS; the error has approximately the same magnitude as the previous RS error.

Here are some toy examples of modulus resurrection. In the following examples, we abuse the definition of n bit integers (or primes) and identify them to be integers (or primes) close to 2^n (roughly, ratio within 1 ± 2^{-16}), which may be greater than 2^n.

Example 2 *Let q_i's be the 60-bit NTT primes and $r_{top} = r_0 \cdot r_1$, where r_0 and r_1 are 30-bit NTT primes. Each sprout r can be represented as $r \in \{1, r_0, r_1, r_0 r_1\}$.*

[10] The sprouts can be chosen universally, or specific to the circuits to be evaluated, which will be discussed later in this section.

From a modulus $Q = q_0 \cdots q_\ell \cdot r$, we can continually scale by 30 bits for e.g., rescale by r_0 or r_1 when $r_0 | r$ or $r_1 | r$, respectively, and rational rescale by $q_\ell / r_1 \approx$ 30 bits when $r = 1$.

Example 3 *Let q_i's be the 61-bit NTT primes and $r_{top} = 2^{61}$.*[11] *Each sprouts r is a power of two integers dividing $r_{top} = 2^{61}$. From a modulus $Q = q_0 \cdots q_\ell \cdot 2^{35}$, we can easily rescale by 1 to 35 bits. If we want to rescale by 36 bits (or more), we can rational rescale by $\left(q_\ell / 2^{25}\right) \approx 2^{36}$, resulting in a ciphertext modulus $Q' = q_0 \cdots q_{\ell-1} \cdot 2^{60}$.*

Example 4 *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*[12]*. Each sprout r can be represented as $r = 2^\alpha \cdot r_1^{\beta_1} \cdot r_2^{\beta_2}$, where $0 \leq \alpha \leq 15$, $\beta_i \in \{0, 1\}$. From a modulus $Q = q_0 \cdots q_\ell \cdot r$, where $r = 2^{13} \cdot r_2$ is $13 + 30 = 43$ bits, we want to rescale by 29 bits. As r has no factor of 29 bits but is larger than that, we can rational rescale by $(r_2/2)$, resulting in a ciphertext modulus $Q' = q_0 \cdots q_\ell \cdot r'$, where $r' = \left(2^{13} \cdot r_2\right) / (r_2/2) = 2^{14}$. If we want to rescale by 34 bits in addition, we can rational rescale by $\left(2^3 \cdot q_\ell / r_2\right)$, resulting in a ciphertext modulus $Q'' = q_0 \cdots q_{\ell-1} \cdot r''$, where $r'' = \left(q_\ell \cdot 2^{14}\right) / \left(2^3 \cdot q_\ell / r_2\right) = 2^{11} \cdot r_2$ is a $14 + 61 - 34 = 41$ bit long sprout.*

Note that a larger amount of rescaling, e.g., 90 bits or more, can be done similarly.

The typical choice of sprout could vary depending on the rescaling factors we will use. For instance, let the machine word size w bits. As in Example 2, the sprout could be $r = r_1 \cdot r_2$ for $(w/2)$-bit (NTT) primes r_1 and r_2, which will allow us to rescale every $(w/2)$ bits. Another option is to have three $(w/3)$-bit (NTT) primes or to have three different sizes so that their additions well represent the possible scale factors, e.g., $10, 20, 30$-bit (NTT) primes can represent all the 10's multiples and thus can be used for such purpose.

In the extreme case, as in Example 2, we can use w number of 1-bit moduli as $r = 2^w$, which can be seen as a hybrid of (locally) binary and (globally) RNS CKKS. Any bit length can be rescaled in this setting and thus is able to be used universally, independent of the parameters.

The following theorem states the rescalability conditions on the sprout. Informally, if the sprouts (i.e., the divisors of r_{top}) can approximately represent all the bit lengths from 1 to w, then the rational rescaling can be done by any arbitrary bit. For instance, the sprout $r_{\text{top}} = 2^{15} \cdot r_1 \cdot r_2$ in Example 4 can represent any bit length from 1 to 60 as

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

[11] To handle modulo 2^{61}, one needs to embed it into larger NTT or just use DFT. See Section 3.5 for more details.

[12] Typically, we can choose $r_1 = 2^{16} + 1$ and $r_2 = 2^{30} + 2^{17} + 1$, the NTT primes for ring dimension $N \leq 2^{15}$.

If the current sprout is, for e.g., $r_1 \cdot 2^{13} \approx 2^{29}$ and we want to rescale by $\approx 2^{41}$, we can simply choose the next sprout to be $r_2 \cdot r_1 \cdot 2^3$, which is $\approx 2^{29+61-41} = 2^{49}$. We then rational rescale by $r_2/2^{10}$. The proof of the theorem can be found in Appendix A.2.

Theorem 5 (Universal Rescalability). *Let the ciphertext modulus $q_i \in [2^w(1-\eta), 2^w(1+\eta)]$ 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 $\gamma \leq w$, 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 $2^\delta \cdot \left(1 \pm (n\eta + 2\epsilon) + \mathcal{O}\left(\eta^2 + \epsilon^2\right)\right)$ for any positive integer $\delta < \log_2(Q)$, where $n = \lceil \delta/w \rceil$.*

Note that the choice of sprout in Examples 3 and 4 satisfies the assumption of Theorem 5 with $\epsilon < 2^{-13}$, and there are plenty of 61 bit primes with $\eta < 2^{-20}$. Thus, the sprouts can be used universally, i.e., rescalable with $2^\delta \cdot (1 \pm 2^{-12})$ for any $\delta \in \mathbb{N}$ smaller than the current ciphertext modulus. We call these sprouts, *universal sprouts*. More generally, universal sprouts can be used independent of the operations or the choice of parameters such as message precision or ring dimension.

We note that the key switching with $\mathsf{dnum} = 1$ can be done the same as the previous RNS-CKKS scheme since each ciphertext modulus is a divisor of the switching key modulus. The only non-trivial part is how to deal with the non-square-free modulus, e.g., the power of two moduli in sprouts, which will be discussed in Section 3.5.

We also note that the key switching with full dnum, i.e., $k = 1$ and $P \approx q_j$, is also possible by filling up the sprout r to r_{top}, where the gadget blocks are $q_0, \ldots, q_{L-1}, r_{\mathrm{top}}$. This can be done by inverse rescaling, $\mathsf{Inv\text{-}RS}_{(r_{\mathrm{top}}/r)}$, from the modulus $Q = q_0 \cdots q_\ell \cdot r$ to $Q' = q_0 \cdots q_\ell \cdot r_{\mathrm{top}}$, and modifying the $\mathsf{ModDown}$ procedure to rescale from the modulus $PQ' = P \cdot (r_{\mathrm{top}}/r) \cdot Q$ to Q.

3.3 Gadget Resurrection: Enabling efficient key switching with gadget decomposition

Modulus resurrection allows us to use word size moduli chain and thus gain efficiency from the reduced number of RNS blocks. However, it is unclear whether one can use key switching with gadget decomposition, i.e., using dnum, during modulus resurrection.

For the $\mathsf{dnum} = 1$ or the full dnum cases, the switching key using gadget decomposition is clearly possible since the current ciphertext modulus can be mapped to the gadget blocks. Otherwise, the resurrected sprout makes the key-switching more costly. For a case when $\mathsf{dnum} > 1$, we have gadget blocks

$$\begin{aligned} Q_i &= q_{\alpha i} q_{\alpha i+1} \cdots q_{\alpha(i+1)-1} \text{ for } i \in [\mathsf{dnum} - 2], \\ Q_{\mathrm{top}} &= q_{\alpha(\mathsf{dnum}-1)} \cdots q_{L-1} r_{\mathrm{top}}, \end{aligned}$$

so that the maximum ciphertext modulus is

$$Q_{\max} = q_0 q_1 \cdots q_{L-1} r_{\mathrm{top}} = Q_0 \cdots Q_{\mathsf{dnum}-2} Q_{\mathrm{top}},$$

where $L + 1 = \mathsf{dnum} \cdot \alpha$.

If the ciphertext modulus is $Q = q_0 \cdots q_\ell$, we can use $d = \lceil \ell/\alpha \rceil$ switching keys, corresponding to the modulus blocks $Q_0, Q_1, \cdots, Q_{\lceil \ell/\alpha \rceil - 1}$. This is just the amount used in the previous approaches, ignoring the fact that the moduli are of word size. Note that the modulus $Q_0 \cdots Q_{\lceil \ell/\alpha \rceil - 1}$ is the smallest multiple of Q among $\Pi_{i \in I} Q_i$, where $I \subseteq [\mathsf{dnum} - 1]$. Else if the ciphertext modulus is a multiple of word size moduli and a sprout, e.g., $Q = q_0 \cdots q_{\ell-1} r$, where $\ell < L$ and $r > 1$, then we need $\lceil (\ell - 1)/\alpha \rceil + 1$ switching keys, corresponding to the gadget blocks $Q_0, Q_1, \cdots, Q_{\lceil (\ell-1)/\alpha \rceil}$, and Q_{top}. This implies that depending on the current ciphertext modulus, one more switching key should be input to each key-switching procedure, assuming that dnum is fixed. This will also slow down the key switching procedure by roughly a factor of $(\lceil (\ell - 1)/\alpha \rceil + 1)/\lceil (\ell - 1)/\alpha \rceil = 2/1, 3/2, 4/3, \ldots$ times, depending on ℓ and α[13].

To avoid such inefficiency, the ciphertext modulus should be well decomposed into moduli, that are mapped to corresponding gadget moduli Q_is and Q_{top}. We introduce our solution managing the ciphertext modulus for efficient key switchings, namely, *Gadget Resurrection*. We focus only on differences between RNS-CKKS [17] and ours. The other parts that are not mentioned below are the same as [17] but using our modifications, consequently.

$\mathsf{Setup}^{**}(N, h, \eta, \mathsf{dnum}, 1^\lambda)$: For given ring dimension N, the secret key hamming weight h, and the security parameter λ, the maximum possible modulus size is set. Each ciphertext modulus is a composition of distinct word-size NTT primes, which we call *unit moduli*, and a small and flexible modulus, which we call *sprout*[14]. Specifically, the maximum ciphertext modulus is $Q_{\max} = q_0 \cdots q_{L-1} \cdot r_{\mathrm{top}}$, where q_is be unit moduli in $2^w \cdot [1 - 2^{-\eta}, 1 + 2^{-\eta}]$, and r_{top} be a maximum possible sprout, which is a common multiple of all the possible sprouts. The switching key modulus is set to $PQ_{\max}$, where P is chosen to satisfy $P = p_0 \cdots p_{K-1}$, where p_i are relatively prime unit moduli, P and $Q_{\max}$ are relatively prime, $P \geq Q_{\max}^{1/\mathsf{dnum}}$, and $PQ_{\max}$ is smaller than the maximum possible modulus size. Depending on the gadget rank dnum, we decompose $Q_{\max}$ into gadget blocks $Q_0, \cdots, Q_{d-2}, Q_{\mathrm{top}}$ having approximately the same size, and $P \geq \max_{i \in I} Q_i$, where $I = [\mathsf{dnum} - 2] \cup \{\mathrm{top}\}$. We assume each gadget block contains α RNS moduli, thus $L + 1 = \mathsf{dnum} \cdot \alpha$. Each ciphertext modulus consists of $Q_0, \cdots, Q_{\beta-2}$ and a divisor of Q_{top} as,

$$Q = Q_0 \cdots Q_{\beta-2} \cdot (q_{\alpha(\mathsf{dnum}-1)} \cdots q_{\alpha(\mathsf{dnum}-1)+l} \cdot r), \text{ or } Q = Q_0 \cdots Q_{\beta-2} \cdot (r),$$

[13] Note, this factor should be multiplied by the speed-up factor comes from the reduction in the number of RNS moduli, thus the actual slow down may less than the given factors, or even, there may be no slow down. However, this factor destroys the naïve speed-up that comes from the reduced number of RNS moduli.

[14] The sprouts can be chosen universally, or specific to the circuits to be evaluated, which will be discussed later in this section.

where $0 \leq l < \alpha - 1$.

$\underline{\mathsf{SwkGen}^{**}(\mathbf{s}_1, \mathbf{s}_2, \mathsf{dnum})}$: For given secret keys $\mathbf{s}_1$ and $\mathbf{s}_2$, it generates and outputs dnum switching keys

$$\mathsf{swk}_i = (\mathbf{a}_i, \mathbf{b}_i = \mathbf{a}_i\mathbf{s}_2 + P\hat{Q}_i\mathbf{s}_1 + \mathbf{e}_i) \in \mathcal{R}_{PQ_{\max}}$$

for $i \in [\mathsf{dnum} - 2] \cup \{\mathrm{top}\}$, where $\hat{Q}_i = Q_{\max}/Q_i$.

$\underline{\mathsf{RS}^{**}(\mathsf{ct}, \delta)}$: For given a ciphertext ct over $\mathcal{R}_Q$, if there is a divisor q of Q which is $\approx 2^{\delta}$, it outputs $\mathsf{RS}_q(\mathsf{ct})$. Else, it computes $\ell' = \lfloor(\lfloor \log_2 Q \rceil - \delta)/w \rfloor$, and if $\ell' \leq \alpha(\beta - 2)$ or $\alpha | \ell'$, it outputs $\mathsf{RS}^*(\mathsf{ct}, \delta)$ (modulus resurrection). Else, let $\ell' = \alpha \cdot (\beta' - 2) + l'$ for some $1 \leq l' < \alpha - 1$, $\tilde{Q}_{\mathrm{top}} = Q/(Q_0 \cdots Q_{\beta-2} r)$, and let $\tilde{Q}_{\beta'-2} = q_{\ell'} \cdots q_{\alpha(\beta-1)-1}$. It outputs $\mathsf{RS}_{(\tilde{Q}_{\mathrm{top}} \cdot r')/(\tilde{Q}_{\beta'-2} Q_{\beta'-1} \cdots Q_{\beta-2} \cdot r)}$, where r' be a sprout $(\delta - \lfloor \log_2 Q \rceil + w\ell')$ bits smaller than r (gadget resurrection).

$\underline{\mathsf{KeySwitch}^{**}(\mathsf{ct}, \{\mathsf{swk}_i\})}$: Identical to the original $\mathsf{KeySwitch}$, except that the switching keys are generated from SwkGen^{**}, and the gadget blocks that are used for key switching are changed to $Q_0, \cdots, Q_{\beta-1}, Q_{\mathrm{top}}$, instead of $Q_0, \cdots, Q_{\beta-1}$.

$\underline{\mathsf{Mult}^{**}(\mathsf{ct}_1, \mathsf{ct}_2, \{\mathsf{evk}_i\})}$: Identical to the original Mult, except that the relinearization is performed using $\mathsf{KeySwitch}^{**}$, and the RS_{q_ℓ} is replaced by RS^{**} with respect to δ bits approximately the scale factor of ct_1 and ct_2, where the ciphertext moduli are the same[15].

We note that the modified RS^{**} (rational) rescale as in RS^* if the rescaling does not change the gadget block $Q_{\beta-2}$ in the cases when $\delta < w$. Otherwise, i.e., if the rescaling will change the gadget block $Q_{\beta-2}$, it is replaced by Q_{top} during the rational rescaling. That is, if $Q = Q_0 \cdots Q_{\beta-2} \cdot r$ and the ciphertext will be rational rescaled by $r'/(q_{\alpha(\beta-1)-1} r)$ to a modulus

$$Q' = Q_0 \cdots Q_{\beta-3} \cdot (q_{\alpha(\beta-2)} \cdots q_{\alpha(\beta-1)-2}) \cdot (r'),$$

we instead, rational rescale by $(r' \cdot q_{\alpha(d-1)} \cdots q_{L-1})/(Q_{\beta-2} \cdot r)$, resulting in a ciphertext modulus of

$$Q'' = Q_0 \cdots Q_{\beta-3} \cdot (q_{\alpha(d-1)} \cdots q_{L-1} r').$$

As in the modulus resurrection, the topmost gadget is resurrected when it was supposed to have been eliminated.

We remark that if the rescaling amount is determined beforehand, and the modulus resurrection needs to be done after the key switching operation, one can modify the $\mathsf{ModDown}$ operation to eliminate only the factor P, and we integral rescale by $Q_{\beta-2} \cdot (q_{\alpha(d-1)+l+1} \cdots q_L \cdot r')$, where r_{top}/r' is the target sprout.

We revisit the toy examples from Examples 2, 3, and 4 in the below.

[15] The scale factor needs to be tracked as in [18], and the scale factors need to be adjusted when the input ciphertexts have different moduli (See Section 3.4 for detail).

Ex.	Ciphertext modulus	Rescaling factor	Gadget blocks
6	$(q_0q_1)(q_2r_0r_1)$		Q_0, Q_{top}
	$\downarrow$	r_1	
	$(q_0q_1)(q_2r_0)$		Q_0, Q_{top}
	$\downarrow$	r_0	
	$(q_0q_1)(q_2)$		Q_0, Q_{top}
	$\downarrow$	q_2/r_0 (moduli rez.)	
	$(q_0q_1)(r_0)$		Q_0, Q_{top}
	$\downarrow$	r_0	
	(q_0q_1)		Q_0
	$\downarrow$	q_0q_1/q_2r_0 (gadget rez.)	
	(q_2r_0)		Q_{top}
	$\vdots$	$\vdots$	$\vdots$
7	$(q_0q_1)(q_2q_3)(q_4 \cdot 2^{61})$		Q_0, Q_1, Q_{top}
	$\downarrow$	2^{26}	
	$(q_0q_1)(q_2q_3)(q_4 \cdot 2^{35})$		Q_0, Q_1, Q_{top}
	$\downarrow$	$2^{25}/q_4 \approx 2^{36}$ (moduli rez.)	
	$(q_0q_1)(q_2q_3)(2^{60})$		Q_0, Q_1, Q_{top}
	$\downarrow$	$q_2q_3 \cdot 2^{11}/q_4 \approx 2^{72}$	
	$(q_0q_1)(q_4 \cdot 2^{49})$	(gadget rez.)	Q_0, Q_{top}
	$\downarrow$	$q_4/2^{10} \approx 2^{51}$ (moduli rez.)	
	$(q_0q_1)(2^{59})$		Q_0, Q_{top}
	$\vdots$	$\vdots$	$\vdots$
8	$(q_0q_1)(q_2q_3)(q_4 \cdot 2^{15} \cdot r_1 \cdot r_2)$		Q_0, Q_1, Q_{top}
	$\downarrow$	$2^2 \cdot r_1 \approx 2^{18}$	
	$(q_0q_1)(q_2q_3)(q_4 \cdot 2^{13} \cdot r_2)$		Q_0, Q_1, Q_{top}
	$\downarrow$	$r_2/2^1 \approx 2^{29}$ (moduli rez.)	
	$(q_0q_1)(q_2q_3)(q_4 \cdot 2^{14})$		Q_0, Q_1, Q_{top}
	$\downarrow$	$q_4 \cdot 2^3/r_2 \approx 2^{34}$ (moduli rez.)	
	$(q_0q_1)(q_2q_3)(2^{11} \cdot r_2)$		Q_0, Q_1, Q_{top}
	$\downarrow$	$q_2q_3 \cdot 2^6/(q_4 \cdot r_1) \approx 2^{51}$	
	$(q_0q_1)(q_4 \cdot 2^5 \cdot r_1)$	(gadget rez.)	Q_0, Q_{top}
	$\vdots$	$\vdots$	$\vdots$

Table 2. Modulus descending scenario in Examples 6, 7, and 8. '$\downarrow$' implies (integral) RS if the rescaling factor is an integer, else rational RS. Resurrections are abbreviated as rez.

Example 6 *We borrow the notations from Example 2, and let the 60-bit NTT primes as q_0, q_1, q_2, and 30-bit NTT primes as r_0, r_1. Then, we can design the ciphertext modulus and the switching key modulus as*

- $Q_{max} = (q_0q_1) \cdot (q_2r_0r_1)$*, where the parenthesis represents the gadget blocks* $Q_0 = q_0q_1$ *and* $Q_{top} = q_2r_0r_1$*,*
- $P = p_0p_1$*, where* p_0 *and* p_1 *are 60-bit NTT primes, so* $\mathsf{dnum} = 2$*.*

From a modulus Q_{max}, we can continually rescale by ≈ 30 bits as in Table 2.

Example 7 *We borrow the notations from Example 3. Then, we can design the ciphertext modulus and the switching key modulus as*

- $Q_{max} = (q_0q_1) \cdot (q_2q_3) \cdot (q_4 \cdot 2^{61})$*, where the parenthesis represents the gadget blocks* $Q_0 = q_0q_1$*,* $Q_1 = q_2q_3$*, and* $Q_{top} = q_4 \cdot 2^{61}$*,*
- $P = p_0p_1$*, where* p_0 *and* p_1 *are 61-bit NTT primes, so* $\mathsf{dnum} = 3$*.*

From a modulus Q_{max}*, we can rescale by any bit lengths as in Table 2.*

Example 8 *We borrow the notations from Example 4. Then, we can design the ciphertext modulus and the switching key modulus as*

- $Q_{max} = (q_0q_1) \cdot (q_2q_3) \cdot (q_4 \cdot 2^{15} \cdot r_1 \cdot r_2)$*, where the parenthesis represents the gadget blocks* $Q_0 = q_0q_1$*,* $Q_1 = q_2q_3$*, and* $Q_{top} = q_4 \cdot 2^{15} \cdot r_1 \cdot r_2$*,*
- $P = p_0p_1$*, where* p_0 *and* p_1 *are 61-bit NTT primes, so* $\mathsf{dnum} = 3$*.*

From a modulus Q_{max}*, we can rescale by any bit lengths as in Table 2.*

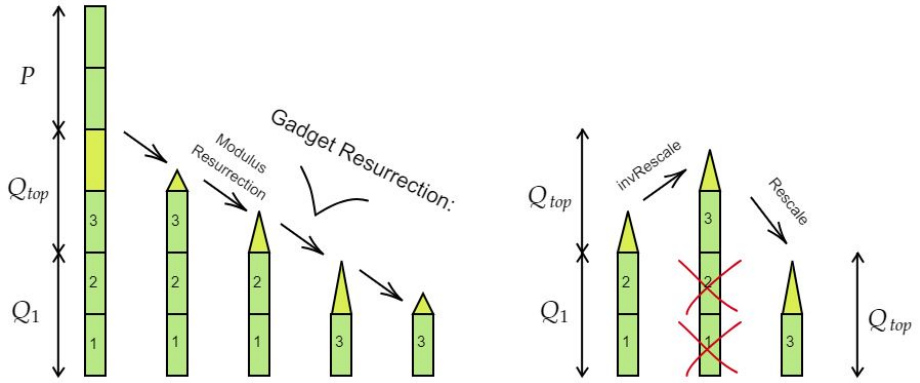

Fig. 1. Gadget Resurrection. Each gadget resurrection is done when a gadget block is changed during modulus resurrection.

In Theorem 9, we prove the correctness of the key switching procedure in the gadget resurrection scenario, which invokes the proof in [17]. The proof of the theorem can be found in Appendix A.3.

Theorem 9 (KeySwitch **Correctness).** *Let* $\mathsf{ct}_{ks} = \mathsf{KeySwitch}(\mathsf{ct}, \{\mathsf{swk}_i\}_{i \in I})$ *for a ciphertext* $\mathsf{ct} = (\mathbf{b}, \mathbf{a}) \in \mathcal{R}_Q^2$ *and* $\mathsf{swk}_i \in \mathcal{R}_{PQ_{max}}^2$ *for* $i \in I$*, where* β *is the smallest integer satisfying* $Q | Q_0 \cdots Q_{\beta-2} Q_{top}$ *and* $I = [\beta - 2] \cup \{top\}$*. Then it holds that* $[\langle \mathsf{ct}_{ks}, (1, \mathbf{s}_2) \rangle]_Q = [\langle \mathsf{ct}, (1, \mathbf{s}_1) \rangle]_Q + \mathbf{e}_{ks}$*, where* $\mathbf{e}_{ks}$ *be a key switching error bounded from above by* $\|\mathbf{s}\|_1 + N \cdot \beta \cdot \max_{i \in I}(\|\mathbf{e}_i\|_\infty)$*, where* $\mathbf{e}_i$ *is an error included in* swk_i*.*

Note that the error introduced from $\mathsf{KeySwitch}^{**}$ is of the same size compared to $\mathsf{KeySwitch}$. The correctness of homomorphic multiplication follows from Theorem 1 and 9. If the modulus resurrection is performed, an additional error amount of RS error is added, and if the gadget resurrection is performed, an additional error amount of $\mathsf{ModDown}$ error is added. Note the gadget resurrection is rarely performed in general scenarios using not too huge dnum, for e.g., less than 5, and is small compared to the $\mathsf{ModDown}$ error, which is added once a level decreases.

3.4 Level Adjustment

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

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

with scaling factor Δ, where the modulus Q can be written as $Q = q_0 \dots q_t r$ for machine word-size primes q_i's and a sprout r'. We aim to adjust two ciphertexts in modulus Q and $Q' = q_0 \dots q_{t'} r' (< Q)$, where the scaling factors are Δ and Δ', respectively. We denote g and g' the positive integers satisfying $\mathrm{lcm}(r, r') = r \cdot g = r' \cdot g'$.

To address the level adjustment in our scenario, we classify the possible cases into three: 1) $t = t'$, 2) $t - t' \geq 2$, and 3) $t - t' = 1$.

Case 1) The first case implies that the sprout r is strictly larger than r' with roughly a rescale factor Δ, i.e., $r \gtrsim r'\Delta$. In this case, we adopt and modify the level adjusting method in [18] to our scenario, by incorporating a suitable amount of integer constant multiplication after $\mathsf{Inv\text{-}RS}_g$, then $\mathsf{RS}_{g'}$. Note that the constant should be large enough so that the factor multiplied in the integer constant can precisely be scaled out when RS. More precisely, we multiply a constant $\left\lceil \frac{g'\Delta'}{g\Delta} \right\rfloor = \frac{g'\Delta'}{g\Delta} + \delta \approx \Delta$, where $\delta \in \left(-\frac{1}{2}, \frac{1}{2}\right]$.

After the adjustment, the output ciphertext $\mathsf{ct}_{\mathrm{adj}} = (a_{\mathrm{adj}}, b_{\mathrm{adj}}) \in \mathcal{R}_{Q'}^2$ satisfies

$$\begin{aligned} b_{\mathrm{adj}} + a_{\mathrm{adj}} \cdot s &= \frac{1}{g'} \left(\frac{g'\Delta'}{g\Delta} + \delta \right) \cdot g(\Delta \cdot m + e) + e_{\mathrm{rs}} \\ &= \Delta' \cdot m + e_{\mathrm{adj}} \pmod{Q'}, \end{aligned}$$

where the e_{rs} is an error added in RS, and the new error e_{adj} is defined as

$$e_{\mathrm{adj}} = \frac{\Delta'}{\Delta} \cdot e + \frac{\delta r' \Delta \cdot m}{r} + \frac{\delta r' e}{r} + e_{\mathrm{rs}}.$$

We note that the condition $r \gtrsim r'\Delta$ allows us to manage the error e_{adj} to be sufficiently small.

We omit the level adjustment techniques for cases 2 and 3 since they are similar in a high-level view, which can be found in Appendix B.

3.5 Exploiting Power-of-Two Modulus in RNS-CKKS

As demonstrated in Example 3, the most intuitive way to set the sprout modulus r_{top} is to choose it as a power-of-two of the machine word-size. In this case, we can seamlessly leverage the advantages of the previous binary CKKS scheme [10] in the RNS manner by efficiently performing rescaling operations for scaling factors of arbitrary bit sizes.

However, we note that in our case, we need to additionally consider the Fast Basis Conversion between the bases containing power-of-two elements, whereas the Fast Basis Conversion in the original RNS-CKKS scheme was performed between the bases composed of distinct NTT primes. In this section, we extend the existing Fast Basis Conversion to ensure that operations such as rational rescale, and modulus resurrection can be implemented in the RNS manner, even for modulus with power-of-two as a sprout r_{top}.

Definition 2 (Rescale by Power-of-Two in RNS). *Let $\mathcal{C}$ and $\mathcal{C}'$ be the bases $\{q_0, \ldots, q_{\ell-1}, q_\ell\}$ and $\{q_0, \ldots, q_{\ell-1}, q'_\ell\}$, respectively, where $q_\ell = 2^\delta$ and $q'_\ell = 2^{\delta+e}$ for some integer $e > 0$ with $Q = q_0 \ldots q_{\ell-1} 2^\delta$ and $Q' = q_0 \ldots q_{\ell-1} 2^{\delta+e} = 2^e Q$. For a given RNS representation $[a]_{\mathcal{C}} = \left(a^{(0)}, \ldots, a^{(\ell)}\right)$ of an integer $a \in \mathbb{Z}_Q$, its* Inv-RS *by a factor 2^e is defined by*

$$\mathsf{Inv\text{-}RS}_{2^e}([a]_{\mathcal{C}}) = [a' = 2^e \cdot a]_{\mathcal{C}'},$$

where its RNS representation is given as

$$a' \equiv \begin{cases} [2^e]_{q_i} \cdot a^{(i)} & (\mathrm{mod}\ q_i) \\ 2^e \cdot a^{(\ell)} & (\mathrm{mod}\ 2^{\delta+e}). \end{cases}$$

For a given RNS representation $[b']_{\mathcal{C}'} = \left(b'^{(0)}, \ldots, b'^{(\ell)}\right)$ of an integer $b \in \mathbb{Z}_{Q'}$, on the other hand, its RS *by a factor 2^e is defined by*

$$\mathsf{RS}_{2^e}([b']_{\mathcal{C}'}) = \left[b = \frac{b' - [b']_{2^e}}{2^e}\right]_{\mathcal{C}},$$

where its RNS representation is given as

$$b \equiv \begin{cases} [(2^e)^{-1}]_{q_i} \cdot \left(b'^{(i)} - [b'^{(\ell)}]_{2^e}\right) & (\mathrm{mod}\ q_i) \\ \dfrac{b'^{(\ell)} - [b'^{(\ell)}]_{2^e}}{2^e} & (\mathrm{mod}\ 2^\delta). \end{cases}$$

We note that these two definitions are well-defined, accompanying the modulus change at power-of-two. Exploiting RS and Inv-RS by a power-of-two factor, one can effectively implement modulus resurrection with power-of-two r_{top} in the RNS manner, and so is gadget resurrection. We give some toy examples for modulus resurrection as follows:

Example 10 *We borrow the notations and parameters setting from Example 7.*

Suppose we aim to rescale by $\Delta = 2^{41}$ at modulus $Q = (q_0 q_1) \cdot (q_2 q_3) \cdot 2^{20}$. Then, we perform $\mathsf{RS}_{(q_3/2^{20})}$ with modulus resurrection as follows:

1. $\mathsf{Inv\text{-}RS}_{2^{20}}$ *from moduli* $\{q_0, q_1, q_2, q_3, 2^{20}\}$ *to* $\{q_0, q_1, q_2, q_3, 2^{40}\}$
2. RS_{q_3} *from moduli* $\{q_0, q_1, q_2, q_3, 2^{40}\}$ *to* $\{q_0, q_1, q_2, 2^{40}\}$.

Here, we utilize $\mathsf{Inv\text{-}RS}_{2^{20}}$ *from the definition 2 so the above procedure can be launched efficiently in the RNS.*

We note that NTT operations can not be utilized in a polynomial ring of power-of-two modulus. Instead, one can address this issue by using DFT instead [6] or embedding the ring into a sufficiently large ring $\mathcal{R}_B$, where B is a product of NTT primes so that NTT operation is applicable in $\mathcal{R}_B$ as partly exploited in [19]. We detail this in Section 4.2. We also remark that power-of-three or other extension may used instead of power-of-two moduli; however, the efficiency gain will not be much.

4 Experiments and Improvements

In this section, we provide some proof-of-concept implementations for our method as well as some concrete parameters showing improvements over major open-source libraries. Our implementations are developed upon the C++ HEaaN library [12]. The experiments are conducted on an Intel Xeon Gold 6242 at 2.8GHz with 503GiB of RAM running Linux, single-threaded.

When describing CKKS parameters, N denotes the ring dimension, $\log(QP)$ denotes the bit size of the maximum switching key modulus, h denotes the hamming weight of secret keys, dnum denotes the gadget decomposition number, $\log(q)$ denotes the bit sizes of the moduli in the ciphertext modulus, and $\log(p)$ denotes the bit sizes of the moduli in the temporary modulus for key switching. When modulus is written as $X \times Y$ then it means that it consists of Y many moduli of size X bits each.

4.1 Homomoprphic Linear Transformation

In homomorphic linear transformations where we don't have ciphertext-ciphertext multiplications, we may use Grafting without sprout. Since linear transform is expensive and we can postpone rescaling until the end of linear transformation, we can afford a bit more expensive rational rescaling, which allows us to choose a more efficient moduli chain.

Let Q_{in} and Q_{out} be input and output moduli of linear transformation where we have evaluation keys at Q_{in}. Given a ciphertext $\mathsf{ct} \in R^2_{Q_{\text{in}}}$, the algorithm can be describes as follows:

1. We evaluate the homomorphic linear transform f and get $\mathsf{Eval}_f(\mathsf{ct}') \in R^2_{Q_{\text{in}}}$.
2. We perform rational rescaling from Q_{in} to Q_{out} and get $\mathsf{ct}_{\text{out}} \in R^2_{Q_{\text{out}}}$.

We may use Q_{in} to be a product of word-size primes without sprouts for better performance. [16]

The described algorithm does not involve special moduli chain adaptations like moduli/gadget resurrections described in Section 3. Instead, it increases the rational rescaling cost a little, which should still be as cheap as the cost of base conversion. In general, most homomorphic linear transformations contain a lot of key switching operations, and the rational rescaling cost can be considered almost negligible.

Recall that homomorphic linear transformations are often a bottleneck of CKKS. In particular, Slots-to-Coeffs and Coeffs-to-Slots steps, which evaluate homomorphic (inverse) discrete Fourier transformations, often take more than half of CKKS bootstrapping. In this regard, the use of Grafting for homomorphic linear transformations should be especially effective in terms of efficiency. We can literally fill-up machine words in RNS, with negligible overhead.

We provide implementation on this technique, particularly on the Coeffs-to-Slots (abbreviated as CtS) step of CKKS bootstrapping, which can be described as a sequence of homomorphic linear transformations. We applied grafting to FTa parameter of the HEaaN library and used as large primes as possible to accelerate the CtS step. Table 3 illustrates the parameters for the initial and suggested parameters, especially focusing on the moduli chains. Instead of using primes with different sizes, we used primes of size $\approx 2^{60}$ so that we can fully use the 64-bit word size.

	N	h	$\log(QP)$	dnum	CtS (sec)
HEaaN.FTa	2^{15}	192	777	10	2.44
Proposed			775	12	1.71

	$\log(q)$					$\log(p)$
	Base	StC	Mult	EvalMod	CtS	
HEaaN.FTa	38	$32 + 28 \times 2$	28×5	38×8	41×3	42×2
Proposed	59×10				$61 + 62$	62

Table 3. Parameter for accelerating CtS.

We measured the CtS times of both parameters for comparison. The FTa and the proposed parameters took 2.44 and 1.71 seconds, respectively. The latter accelerated the former by a factor ≈ 1.43. In terms of the number of unit moduli, the initial and proposed parameters have 22 and 13 moduli, respectively. The difference in the size of dnum compensated this gain: $\frac{22}{13} \times \frac{10}{12} \approx 1.41$ is close to the actual factor 1.43.

For simplicity, we implemented only the CtS part of CKKS bootstrapping. As mentioned above, we may rational rescale to a proper modulus and continue

[16] In case where linear transformation is put in the middle of the homomorphic computations, we may modulus switch to the linear transform modulus, using an analogue of rational rescaling.

Grafting in the other parts of bootstrapping, and ration rescaling should be negligible compared to the CtS cost.

4.2 Proof-of-Concept Implementation for Grafting

We check the techniques proposed throughout the paper in a proof-of-concept manner. We use two different moduli chains, one for usual CKKS and the other for Grafting. The details of the parameters are illustrated in Table 4. Set I contains eight 30-bit primes for rescaling 30 bits per multiplication, while Set II contains three 60-bit primes and two 30-bit primes for Grafting. Two 30-bit primes in the moduli can be stored in one 64-bit word as long as we use both primes. All the modulus operations, including NTT, can be performed on this so-called *composite number NTT* only requires a primitive $2N$-th root of unity [11].

	N	$\log(QP)$	h	dnum	$\log(q)$	$\log(p)$
Set I	14	300	256	4	30×8	60
Set II					$60 \times 3 + (30 + 30)$	60

Table 4. Two moduli chains for comparison.

How to climb down the moduli chain. Let the primes in Set II be q_0, q_1, q_2, r_1, r_2 where $q_0, q_1, q_2 \sim 2^{60}$ and $r_1, r_2 \sim 2^{30}$. We elaborate the descending strategy as follows:

$$(q_0, q_1 \mid q_2, r_1 r_2) \xrightarrow{\mathsf{RS}_{r_2}} (q_0, q_1 \mid q_2, r_1) \xrightarrow{\mathsf{RS}_{r_1}} (q_0, q_1 \mid q_2) \xrightarrow{\mathsf{RS}_{q_2/r_1}} (q_0, q_1 \mid r_1)$$

$$\xrightarrow{\mathsf{RS}_{r_1}} (q_0, q_1) \xrightarrow{\mathsf{RS}_{q_0 q_1 / q_2 r_1 r_2}} (q_2, r_1 r_2) \xrightarrow{\mathsf{RS}_{r_2}} (q_2, r_1) \xrightarrow{\mathsf{RS}_{r_1}} (q_2) \xrightarrow{\mathsf{RS}_{q_2/r_1}} (r_1).$$

Here the divider '|' is used to illustrate different gadget blocks. 30 bit rescaling without 30 bit prime is performed using inverse rescaling by 30 bits and rescaling by 60 bits. The connection between (q_0, q_1) and $(q_2, r_1 r_2)$ is the gadget resurrection, where we revive the top gadget block. The remaining rational rescalings correspond to modulus resurrection. Despite of its simplicity, the 30 and 60 bits example reflects the general power-of-two modulus case, in the sense that it covers rational rescaling, modulus resurrection, and gadget resurrection. Here we could have used 30+30 moduli for all the 60 bit blocks, but we used only a single set in order to capture the problems in the power-of-two case.

Additional Operations in Grafting. Now we analyze the additional operations that the Grafting introduces over the conventional CKKS multiplication. First, rational rescaling is more expensive than integer rescaling because it is implemented by inverse rescaling and rescaling. Second, rescaling the composite unit modulus $r_1 r_2$ requires a different rescaling strategy. We implemented this by performing inverse

CRT plus usual rescaling. Third, embedding r_1 into $r_1 r_2$ could be implemented with identity. Although modulo r_2 part is completely ruined throughout the computation, it vanishes as soon as we take moudlo r_1 after key switching. These characteristics explain why the timings of Set II don't linearly increase by levels. For instance, the difference in rescaling time between levels 4 and 5 stems from the fact that rescaling at level 5 is RS_{q_2/r_1}, which is a rational rescaling that is more expensive than the others.

Performance Comparison. We implemented each component of homomorphic multiplication for each level of Set II. The timings for homomorphic multiplications, as well as the gadget resurrection, are given in Table 5. The speedup for overall multiplication varies from factor 1.17 to 1.51. Although Grafting introduces a new operation gadget resurrection, this can be performed only once during the whole computation and is negligible compared to the rest.

Level		1	2	3	Resurrect	4	5	6	7
Set I	Tensor	221	335	416	-	496	572	677	767
	Relin	2469	4165	4326		6062	6918	9542	10979
	Rescale	477	731	955		1149	1411	1655	1907
Set II	Tensor	122	318	269	1934	389	433	297	515
	Relin	1466	3234	3030		5192	5556	7188	7110
	Rescale	881	449	983		692	1611	1030	1436
Speedup		1.28x	1.31x	1.33x	-	1.23x	1.17x	1.39x	1.51x

Table 5. Performance comparison between Set I and Set II in terms of homomorphic multiplication at different levels. The numbers in the columns refer to the starting level of the operation and the times are microseconds. The Resurrect columns denote the gadget resurrection placed between level 3 and level 4 multiplications.

Level Adjustment. We explain level adjustment with an example that decreases the level from 6 to 3. Recall that level 6 corresponds to a moduli chain $(q_0, q_1 \mid q_2, r_1)$ and level 3 corresponds to a moduli chain $(q_2, r_1 r_2)$. To do this, we first raise the modulus to $q_0 q_1 q_2 r_1$ with inverse-rescale, multiply an appropriate constant to adjust scaling factors, and rescale by $q_0 q_1$. We implemented this level-down operation and checked that it has 18 bits of precision [17] which is almost the same as that of the level-down operation in Set I.

Implementing General Grafting. In this section, we only implemented a simple case of Grafting consisting of 30 and 60-bit primes. Although this simple example captures most of the ingredients of Grafting, there are a few differences when implementing more general Grafting. The most important difference is that it may include a modulus that does not support NTT, as in the power-of-two case

[17] Here, the precision is defined as the infinite norm of the error.

mentioned in Section 3.5. To handle such modulus, one approach is to regard polynomial multiplication in R_q as multiplication in R and use a larger modulus ($\gg q^2$) NTT to enable such multiplication. Such an approach can be regarded as a small modulus case for hybrid multi-precision implementation in [10]. In case where we use only small power-of-two modulus, we may also use DFT instead of NTT as in [6]. That is, we can choose the DFT precision to be sufficiently greater than the power-of-two modulus size and embed polynomial multiplication. The computation cost for moduli not supporting NTT will be greater than the 64-bit unit modulus computation cost that supports NTT, but it should be insignificant compared to the overall cost because we only have a few moduli requiring special treatment.

4.3 Estimation of Performance Improvements in the Existing Parameters

In this subsection, we investigate the CKKS scheme parameters available in HE libraries and estimate the acceleration our Grafting technique could achieve when applied to them. We examine the default CKKS scheme parameters currently launched in HEaaN [12], SEAL [23], Lattigo [1], and OpenFHE [3], categorizing them into two groups: SHE parameters and FHE parameters. For SHE parameters, we search for parameters with base rings of dimensions $N = 2^{14}$ to 15, while for FHE parameters, we search for ring dimensions of $N = 2^{15}$ to 16. The details of the original parameters are provided in Table 6, 7.

	$\log N$	$\log PQ$	$\log \Delta$	#mod.	$\log q_i$	$\log p_i$
HEaaN [12]	14	436	42	10	$50 + 42 \times 7$	46×2
	15	652	51	12	$61 + 51 \times 8$	61×3
Lattigo [1]	14	438	34	12	$45 + 34 \times 9$	43×2
	15	880	40	21	$50 + 40 \times 17$	50×3
OpenFHE [3]	14	371	50	7	$60 + 50 \times 5$	60
	15	675	90	7	$105 + 90 \times 5$	119
	16	794	90	8	$105 + 90 \times 5$	119×2
SEAL [23]	14	438	48	9	$48 \times 3 + 49 \times 6$	
	15	881	55	16	$55 \times 15 + 56$	

Table 6. SHE parameters for CKKS scheme in the literature. All the sizes are given in logarithms base two and #mod. denotes the number of RNS moduli (or NTT primes) comprising the switching key modulus PQ. The moduli $\log q_i$ and $\log p_i$ are given with each moduli size $\times$ the number of each size ($\times 1$ is omitted). Note, the first two parameter sets are SS7 and ST8 from the HEaaN library.

We note that the size of RNS primes comprising modulus PQ is closely related to the size of scaling factor Δ: primes q_i's for multiplication has a bit-size of $\log \Delta$, and the base modulus prime q_0 and the special modulus primes p_j's are often set to be roughly 10-20 bits larger than them. The larger the dimension of the base

	$\log N$	$\log PQ$	$\log \Delta$	#mod.	$\log q_i$					$\log p_i$
					Base	StC	Mult	Sine	CtS	
HEaaN [12]	15	777	28	22	32	28×2	28×5	38×8	41×3	42×2
	16	1555	42	30	58	42×3	42×9	58×9	58×3	$59 \times 3, 60 \times 2$
Lattigo [1]	15	768	25	16	50	60	$50 + 25$	50×8	49×2	50×2
	16	1546	40	30	60	39×3	40×9	60×8	56×4	61×5
	16	1547	45	28	60	42×3	45×5	60×11	58×4	61×4
	16	1553	30	27	55	60×1.5	60×7.5	55×8	53×4	61×5
OpenFHE [3]	16	1579	58	27	60	58×2	58×4	58×13	58×2	60×5
	17	2910	78	34	89	78×3	78×8	78×13	78×3	119×6

Table 7. FHE parameters for CKKS scheme in the literature. All the numbers are given as in Table 6, except that $\log q_i$ shows each set of RNS moduli reserved for corresponding steps of CKKS bootstrapping, or homomorphic multiplications. Note, the first two parameter sets are FTa and FGb from the HEaaN library.

ring, the more modulus budget is available, so a larger scaling factor Δ is utilized to ensure high precision. In particular, OpenFHE offers parameter customization, where the default scaling factor size 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 64-bit word size. OpenFHE also supports 128-bit CKKS for high precision, where the default scaling factor size is 78-bits, and the base modulus prime is from 89 to 105-bits. In this case, our technique reduces the inefficiency of using either two 64-bit word-size moduli or a single large 128-bit modulus to perform 78-bit numeric arithmetic operations.

Reduction in the Number of NTT Blocks. We redesign the suggested parameters for the CKKS scheme with word-size NTT primes to utilize our Grafting technique while preserving the overall modulus $\log(PQ)$ and examine the reduction in the number of NTT primes comprising the ciphertext modulus. For example, the overall 777-bit modulus of 22 NTT primes in ring dimension $N = 2^{15}$ can be expressed as the product of 13 NTT primes or a 1555-bit modulus of 30 NTT primes as the product of 26 NTT primes, each ranging from 59 to 62 bits in size, as described in Table 8. We note that the major cost in the KeySwitch operation is dnum times NTT operations in modulus PQ, and our Grafting technique does not require additional NTT operations while performing KeySwitch. Therefore, we can expect $\frac{22}{13} \fallingdotseq 1.69\times$ and $\frac{30}{26} \fallingdotseq 1.15\times$ acceleration can be achieved during KeySwitch. We note that some parameters from [3] set the scaling factor size to be 59-bit size so that our grafting technique does not significantly reduce the number of NTT blocks. However, these parameters significantly limit the number of available Mult levels due to the large scaling factors, whereas our approach is not subject to this constraint.

For scaling factor $\Delta = 2^{78}$ in parameter sets of OpenFHE, 8 and 34 NTT blocks of 128-bit word size are transformed into 12 and 48 NTT blocks of 64-

		$\log N$	$\log PQ$	$\log \Delta$	#mod.	$\log q_i$	$\log p_i$	Speed-up
HEaaN [12]	SHE	14	436	42	$10 \to 7$	$4 \times 62 + 63$	$62 + 63$	$\times 1.43$
		15	652	51	$12 \to 11$	$60 + 7 \times 59$	$59 + 2 \times 60$	$\times 1.09$
	FHE	15	777	28	$22 \to 13$	$4 \times 59 + 7 \times 60$	$60 + 61$	$\times 1.69$
		16	1555	42	$30 \to 26$	$10 \times 59 + 11 \times 60$	5×61	$\times 1.15$
Lattigo [1]	SHE	14	438	34	$12 \to 7$	$3 \times 62 + 3 \times 63$	63	$\times 1.71$
		15	880	40	$21 \to 15$	$5 \times 58 + 7 \times 59$	3×59	$\times 1.40$
	FHE	15	768	25	$16 \to 13$	11×59	$59 + 60$	$\times 1.23$
		16	1546	40	$30 \to 26$	$19 \times 59 + 2 \times 60$	5×61	$\times 1.15$
		16	1547	45	$28 \to 26$	$17 \times 59 + 5 \times 60$	4×61	$\times 1.08$
		16	1553	30	$27 \to 26$	$12 \times 59 + 9 \times 60$	5×61	$\times 1.04$
OpenFHE [3]	SHE	14	371	50	$7 \to 6$	$4 \times 62 + 61$	62	$\times 1.17$
		15	675	90	$7 \to 6$	$111 \times 4 + 112$	119	$\times 1.16^*$
		16	794	90	$8 \to 7$	$111 \times 4 + 112$	2×119	$\times 1.14^*$
	FHE	16	1579	58	$27 \to 26$	$15 \times 61 + 6 \times 62$	5×62	$\times 1.04$
		17	2910	78	$34 \to 25$	$8 \times 115 + 11 \times 116$	6×119	$\times 1.35^*$
SEAL [23]	SHE	14	438	48	$9 \to 6$	$61 + 5 \times 62$		$\times 1.50$
		15	881	55	$16 \to 15$	$4 \times 58 + 11 \times 59$		$\times 1.07$

Table 8. Suggested RNS moduli with Grafting for existing SHE and FHE parameters in the literature. The security strength is exactly the same with the corresponding parameters, and the 64-bit machine is assumed. The numbers are given as in Table 6. The number of RNS moduli shows the changes, the numbers from the original parameters to that of the suggested parameters. In addition, the expected speed-up ratio is given in the last column. An asterisk ($*$) indicates that it assumes the machine of 128-bit word size for a fair comparison with the original choices of RNS moduli.

bit word size, respectively. Since one can estimate 128-bit numeric arithmetic operations cost at least $2\times$ slower than those of 64-bit, we expect at least $2 \times \frac{7}{11} \coloneqq 1.27\times$ and $2 \times \frac{34}{48} \coloneqq 1.42\times$ acceleration during $\mathsf{KeySwitch}$. We note that the magnitude of the acceleration is estimated based on the $\mathsf{KeySwitch}$ at the top level. In addition, our Grafting technique may modify the dnum during moduli transformation in practice, potentially altering the magnitude of acceleration based on the parameter conditions, as shown in the previous section.

5 Revisiting Tuple-CKKS

In [8], they 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. Our method 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 denote $Q^{(\ell)}$ the modulus for the ciphertext of level ℓ.

5.1 Compatibility

We check the compatibility of our method with the multiplication of [8]. For simplicity, we stick to the pair multiplication - the general tuple multiplication should be checked almost the same. In addition, we stick to the key-switching ladder that contains sprout modulus at the very top, and restores it every time we enter the new gadget block. Let's recall the definitions of the components of the pair multiplication in [8, 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, and DCP denotes the decomposition of CKKS ciphertext into quotient and remainder as defined in [8, Definition 3.3].

Definition 3 (Pair Tensor). *Let* $\mathsf{CT}_1 = (\hat{\mathsf{ct}}_1, \check{\mathsf{ct}}_1), \mathsf{CT}_2 = (\hat{\mathsf{ct}}_2, \check{\mathsf{ct}}_2) \in R^2_{Q^{(\ell)}} \times R^2_{Q^{(\ell)}}$ *be ciphertext pairs. The tensor of* CT_1 *and* CT_2 *is defined as*

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

Definition 4 (Pair Relinearize). *Let* $\mathsf{CT} = (\hat{\mathsf{ct}}, \check{\mathsf{ct}}) \in R^3_{Q^{(\ell)}} \times R^3_{Q^{(\ell)}}$ *be an output of* $\otimes^2$*. The relinearization of* CT *is defined as*

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

Definition 5 (Pair Rescale). *Let* $\mathsf{CT} = (\hat{\mathsf{ct}}, \check{\mathsf{ct}}) \in R^2_{Q^{(\ell)}} \times R^2_{Q^{(\ell)}}$ *be a ciphertext pair. Let* $q_\ell = Q_\ell / Q_{\ell-1}$*. The rescale of* CT *is defined as*

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

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 $\mathsf{Relin}(q_{\mathsf{div}} \cdot \hat{\mathsf{ct}})$ and $(0, \mathsf{Relin}(\check{\mathsf{ct}}))$, 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 [8, Lemma 4.4].

Hence, the only difficulty is to define pair rescale when $Q^{(\ell)}/Q_{\ell-1} \notin \mathbb{Z}$, which can happen in our new framework. We define a generalized (rational) version of pair rescale as follows:

Definition 6 (Pair Rescale, Rational). *Let* $\mathsf{CT} = (\hat{\mathsf{ct}}, \check{\mathsf{ct}}) \in R^2_{Q^{(\ell)}} \times R^2_{Q^{(\ell)}}$ *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. The rescale of* CT *is defined as*

$$\mathsf{RS}^2_{\alpha_\ell/\beta_\ell} := \left(\mathsf{RS}_{\alpha_\ell}(\beta_\ell \cdot \hat{\mathsf{ct}}), \mathsf{RS}_{\alpha_\ell}(q_{\mathsf{div}}\beta_\ell \cdot \hat{\mathsf{ct}} + \beta_\ell \cdot \check{\mathsf{ct}}) - q_{\mathsf{div}} \cdot \mathsf{RS}_{\alpha_\ell}(\beta_\ell \cdot \hat{\mathsf{ct}})\right).$$

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

We also need a generalized version of [8, Lemma 4.6], to check the correctness of the pair multiplication after changing the rescale definition.

Lemma 1. *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* $\mathsf{sk} = (1, s) \in R^2$ *be a secret key with* s *of Hamming weight* h*. Then the quantity*

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

has infinity norm $\leq (h+1)/2$.

The proof can be found in Appendix A.4. The Lemma gives an analogue of [8, Theorem 4.8], guaranteeing the correctness of pair multiplication. We define Mult^2 as a composition of tensor, relinearize, and rescale.

Theorem 11. *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* $\mathsf{sk} = (1, s) \in R^2$ *be a secret key with* s *of Hamming weight* h*. Assume that* $\|\mathsf{Dec}(\hat{\mathsf{ct}}_i)\|_\infty \leq \hat{M}$ *and* $\|\mathsf{Dec}(\check{\mathsf{ct}}_i)\|_\infty \leq \check{M}$ *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*

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

has infinity norm $\leq (N\check{M}^2/q_{\mathsf{div}} + E_{\mathsf{Relin}} + h)\beta_\ell/\alpha_\ell + (h+1)/2$.

5.2 Efficiency

Next, we focus on efficiency gain when applying grafting to double-CKKS. Let $a = \lfloor \log_2(q_{\mathsf{div}}) \rceil, b = \lfloor \log_2(Q^{(\ell)}) \rceil$ be sizes of the division prime and the rescaling primes, 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, which discretizes the computation performance. This could greatly degrade the performance of Mult, especially when we deal with moduli that are smaller than the word size. In the worst case where $a+b$ bits computation and b bits computation are handled almost the same in the machine, Mult^2 could be at most two times slower than Mult in terms of throughput.

In [8, Table 2], they provided a parameter that increases the homomorphic capacity compared to the conventional CKKS parameter using 57-bit primes. The chain they used is

$$61 + 38 \times 18 + 23 \times 3 + 61$$

which has an average moduli size of ≈ 38. Using grafting, we may increase the average moduli size to as high as ≈ 61, which would include 15 unit moduli in the moduli chain. This allows us to use dnum of 14 instead of 22. In terms of the number of unit NTTs, the new parameter should win by $\frac{22}{14} \times \frac{23}{15} \approx 2.41$. Note that many moderate precision CKKS parameters use scaling factors varying from 40 to 60 bits. The use of grafting should improve the performance of double-CKKS, achieving similar precision.

In addition, the efficiency gain could be even more significant for low precision or fewer slots scenarios. Recall that the CKKS errors, such as rescaling errors, are often proportional to $\sqrt{N}$ where N is the ring dimension. Hence, the size of rescaling moduli should be chosen considering precision and number of slots. We suggest some parameters where grafting could be especially powerful in terms of performance. For moderate precision ≈ 20 bits, we may consider using double-CKKS with $\log_2(q_{\mathsf{div}}) = 15$ and $\log_2(Q^{(\ell)}) = 30$. Here, we already gain at most a factor of two when replacing them to ≈ 60-bit primes via grafting. When we use only a few slots, we may even use $\log_2(q_{\mathsf{div}}) \leq 10$ and $\log_2(q_{\mathsf{div}}) \leq 20$, leading to factor ≥ 3 improvements compared to naive double-CKKS implementation. Note that when it comes to extremely small scaling factors, one may consider the hybrid multiprecision-RNS implementation as in the original CKKS paper [10], although it is much smaller than purely RNS implementation as mentioned in [9].

6 Conclusion

We propose Grafting, a ciphertext modulus management system, which allows us to fill the RNS-CKKS ciphertext modulus with machine word-size RNS moduli.

This results in substantial improvements in the efficiency of RNS-CKKS across various HE parameters. Our technique is effective in both low- and high-precision scenarios, as the size of the rescaling factor is often far from the multiples of the word size. In addition, Grafting decouples the RNS moduli comprising the modulus from the rescaling amount, allowing the choice of ciphertext modulus to be independent of specific applications. Such characteristics are hardware and compiler-friendly, as the computations are not tied to the choice of homomorphic operations for different circuits. Selecting HE parameters has traditionally been an obstacle in implementing HE due to its application-dependent nature. We expect that our research will also lower the hurdle of HE parameter selection, leading to more fruitful research in HE applications.

References

1. Lattigo v5. Online: https://github.com/tuneinsight/lattigo (Nov 2023), ePFL-LDS, Tune Insight SA
2. Agrawal, R., Ahn, J.H., Bergamaschi, F., Cammarota, R., Cheon, J.H., D. M. de Souza, F., Gong, H., Kang, M., Kim, D., Kim, J., de Lassus, H., Park, J.H., Steiner, M., Wang, W.: 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. p. 23–34. WAHC '23, Association for Computing Machinery, New York, NY, USA (2023)
3. Al Badawi, A., Bates, J., Bergamaschi, F., Cousins, D.B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., Kim, A., Lee, Y., Liu, Z., Micciancio, D., Quah, I., Polyakov, Y., R.V., S., Rohloff, K., Saylor, J., Suponitsky, D., Triplett, M., Vaikuntanathan, V., Zucca, V.: OpenFHE: Open-source fully homomorphic encryption library. In: Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography. pp. 53–63. WAHC'22, Association for Computing Machinery, New York, NY, USA (2022)
4. Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. Journal of Mathematical Cryptology 9(3), 169–203 (2015)
5. Bajard, J.C., Eynard, J., Hasan, M.A., Zucca, V.: A full RNS variant of FV like somewhat homomorphic encryption schemes. In: Avanzi, R., Heys, H.M. (eds.) SAC 2016. LNCS, vol. 10532, pp. 423–442. Springer, Heidelberg (Aug 2016)
6. Belorgey, M.G., Carpov, S., Gama, N., Guasch, S., Jetchev, D.: Revisiting key decomposition techniques for FHE: Simpler, faster and more generic. Cryptology ePrint Archive, Paper 2023/771 (2023), https://eprint.iacr.org/2023/771
7. Bossuat, J.P., Mouchet, C., Troncoso-Pastoriza, J.R., Hubaux, J.P.: Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In: Canteaut, A., Standaert, F.X. (eds.) EUROCRYPT 2021, Part I. LNCS, vol. 12696, pp. 587–617. Springer, Heidelberg (Oct 2021)
8. Cheon, J.H., Cho, W., Kim, J., Stehlé, D.: Homomorphic multiple precision multiplication for CKKS and reduced modulus consumption. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. p. 696–710. CCS '23, Association for Computing Machinery, New York, NY, USA (2023)
9. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A full RNS variant of approximate homomorphic encryption. In: Cid, C., Jacobson Jr:, M.J. (eds.) SAC 2018. LNCS, vol. 11349, pp. 347–368. Springer, Heidelberg (Aug 2019)

10. Cheon, J.H., Kim, A., Kim, M., Song, Y.S.: Homomorphic encryption for arithmetic of approximate numbers. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017, Part I. LNCS, vol. 10624, pp. 409–437. Springer, Heidelberg (Dec 2017)
11. Chung, C.M.M., Hwang, V., Kannwischer, M.J., Seiler, G., Shih, C.J., Yang, B.Y.: 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), https://tches.iacr.org/index.php/TCHES/article/view/8791
12. CryptoLab: HEaaN library (2022), Is available at https://heaan.it/
13. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 850–867. Springer, Heidelberg (Aug 2012)
14. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the aes circuit. In: Annual Cryptology Conference. pp. 850–867. Springer (2012)
15. Halevi, S., Hunt, H., Shoup, V., Masters, O., Bergamaschi, F., Crawford, J., Boemer, F., et al.: HElib (version 2.2.1) (October 2021), available at https://github.com/homenc/HElib
16. Halevi, S., Shoup, V.: Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive, Paper 2020/1481 (2020), https://eprint.iacr.org/2020/1481, https://eprint.iacr.org/2020/1481
17. Han, K., Ki, D.: Better bootstrapping for approximate homomorphic encryption. In: Jarecki, S. (ed.) CT-RSA 2020. LNCS, vol. 12006, pp. 364–390. Springer, Heidelberg (Feb 2020)
18. Kim, A., Papadimitriou, A., Polyakov, Y.: Approximate homomorphic encryption with reduced approximation error. In: Galbraith, S.D. (ed.) CT-RSA 2022. LNCS, vol. 13161, pp. 120–144. Springer, Heidelberg (Mar 2022)
19. Kim, M., Lee, D., Seo, J., Song, Y.: Accelerating HE operations from key decomposition technique. CRYPTO 2023. (Aug 2023)
20. Kim, S., Park, M., Kim, J., Kim, T., Min, C.: EvalRound algorithm in CKKS bootstrapping. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 161–187. Springer (2022)
21. Li, B., Micciancio, D.: On the security of homomorphic encryption on approximate numbers. In: Canteaut, A., Standaert, F.X. (eds.) EUROCRYPT 2021, Part I. LNCS, vol. 12696, pp. 648–677. Springer, Heidelberg (Oct 2021)
22. Mono, J., Güneysu, T.: A new perspective on key switching for bgv-like schemes. Cryptology ePrint Archive (2023)
23. Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL (Jan 2023), microsoft Research, Redmond, WA.
24. SNUCrypto: HEAAN library v1.1 (2018), Is available at https://github.com/snucrypto/HEAAN

A Proofs of Theorems and Lemmas

A.1 Proof of Theorem 1

Proof. Let us follow the notations in Definition 1 Let $[\langle \mathsf{ct}, \mathsf{sk} \rangle]_Q = \Delta \cdot m + e$, or $\langle \mathsf{ct}, \mathsf{sk} \rangle = \Delta \cdot m + e + QI$ for some $I \in \mathcal{R}$. Then $\langle \mathsf{Inv\text{-}RS}_R(\mathsf{ct}), \mathsf{sk} \rangle = R\Delta \cdot m + Re + QRI$. The final rescaling RS_S introduces an additional error e_{res}, where

$\|e_{\mathrm{res}}\| \leq \ell/2 \cdot (\|s\|_1 + 1)$ for ℓ the number of RNS blocks in $S = \mathrm{lcm}(Q, Q')/Q'$. Precisely, we have

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

which concludes the proof.

A.2 Proof of Theorem 5

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 $0 \leq \gamma < w$. Let $w\ell + \gamma - \delta = w\ell' + \gamma'$, where $0 \leq \gamma' < w$. Since $\delta = w(\ell - \ell') + \gamma - \gamma'$, it holds that $n = \lceil \delta/w \rceil = \lceil \ell - \ell' + (\gamma - \gamma')/w \rceil \geq \ell - \ell'$. 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

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

which concludes the proof.

A.3 Proof of Theorem 9

Proof. Step 1 outputs β number of $\mathbf{a}^{(i)}$ in $\mathcal{R}_{Q_0 \cdots Q_{\beta-2} Q_{\mathrm{top}}}$, each corresponds to gadget blocks Q_i, $i \in [d-2] \cup \{\mathrm{top}\}$, satisfying $\mathbf{a}^{(i)} \equiv \mathbf{a} \cdot \hat{Q}_i^{-1} \bmod Q_i$. Step 2 outputs $\tilde{\mathbf{a}}^{(i)} \in \mathcal{R}_{PQ_0 \cdots Q_{\beta-2} Q_{\mathrm{top}}}$ satisfying $\tilde{\mathbf{a}}^{(i)} \equiv \mathbf{a} \cdot \hat{Q}_i^{-1} \bmod Q_i$ and $\|\tilde{\mathbf{a}}^{(i)}\|_\infty \leq (\alpha+1) Q_i$. Step 3 outputs $(\mathbf{b}', \mathbf{a}') \in \mathcal{R}_{PQ_0 \cdots Q_{\beta-2} Q_{\mathrm{top}}}$ such that

$$\mathbf{b}' + \mathbf{a}'\mathbf{s} = P \sum_i \tilde{\mathbf{a}}^{(i)} \cdot \hat{Q}_i \cdot \mathbf{s}' + \sum_i \tilde{\mathbf{a}}^{(i)} \cdot \mathbf{e}_i = P \cdot \mathbf{a} \cdot \mathbf{s}' + \mathbf{e}',$$

where $\|\mathbf{e}'\|_\infty \leq N\beta \cdot \max(\|\mathbf{e}_i\|_\infty) \cdot \max(Q_i)$. This error term is scaled by P in step 4, and the rounding error is newly introduced, which is bounded from above by $\|\mathbf{s}\|_1$. This concludes the proof.

A.4 Proof of Lemma 1

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

$$\left(\mathsf{RCB}_{q_{\mathsf{div}}}(\mathsf{RS}^2_{\alpha_\ell/\beta_\ell}(\mathsf{CT}))\right) \cdot (1, s) = (\mathsf{RS}_{\alpha_\ell}(\mathsf{RCB}_{q_{\mathsf{div}}}(\mathsf{CT}) \cdot \beta_\ell)) \cdot (1, s).$$

Now, to complete the proof, note that

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

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

B Level Adjustment

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

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

with scaling factor Δ, where the modulus Q can be written as $Q = q_0 \dots q_t r$ for machine word-size primes q_i's and a sprout r'. We aim to adjust two ciphertexts in modulus Q and $Q' = q_0 \dots q_{t'} r' (< Q)$, where the scaling factors are Δ and Δ', respectively. We denote g and g' the positive integers satisfying $\mathrm{lcm}(r, r') = r \cdot g = r' \cdot g'$.

To address the level adjustment in our scenario, we classify the possible cases into three: 1) $t = t'$, 2) $t - t' \geq 2$, and 3) $t - t' = 1$.

Case 1) The first case implies that the sprout r is strictly larger than r' with roughly a rescale factor Δ, i.e., $r \gtrsim r'\Delta$. In this case, we adopt and modify the level adjusting method in [18] to our scenario, by incorporating a suitable amount of integer constant multiplication after $\mathsf{Inv\text{-}RS}_g$, then $\mathsf{RS}_{g'}$. Note that the constant should be large enough so that the factor multiplied in the integer constant can precisely be scaled out when RS. More precisely, we multiply a constant $\left\lceil \frac{g'\Delta'}{g\Delta} \right\rfloor = \frac{g'\Delta'}{g\Delta} + \delta \approx \Delta$, where $\delta \in \left(-\frac{1}{2}, \frac{1}{2}\right]$.

After the adjustment, the output ciphertext $\mathsf{ct}_{\mathrm{adj}} = (a_{\mathrm{adj}}, b_{\mathrm{adj}}) \in \mathcal{R}_{Q'}^2$ satisfies

$$\begin{aligned} b_{\mathrm{adj}} + a_{\mathrm{adj}} \cdot s &= \frac{1}{g'} \left(\frac{g'\Delta'}{g\Delta} + \delta \right) \cdot g(\Delta \cdot m + e) + e_{\mathrm{rs}} \\ &= \Delta' \cdot m + e_{\mathrm{adj}} \pmod{Q'}, \end{aligned}$$

where the e_{rs} is an error added in RS, and the new error e_{adj} is defined as

$$e_{\mathrm{adj}} = \frac{\Delta'}{\Delta} \cdot e + \frac{\delta r' \Delta \cdot m}{r} + \frac{\delta r' e}{r} + e_{\mathrm{rs}}.$$

We note that the condition $r \gtrsim r'\Delta$ allows us to manage the error e_{adj} to be sufficiently small.

Case 2) In the case when $t - t' \geq 2$, we extend the method in case 1. Note the relations between Δ and Δ' is now unclear, we reduce the ciphertext ct modulo $q_0 \cdots q_{t'+2} r$. We then $\mathsf{Inv\text{-}RS}$ by $\mathrm{lcm}(r, r')/r$, then multiply a constant that is sufficiently large to change the scale factor. We than RS by a factor of $q_{t'+1} q_{t'+2} \cdot g'$, which result in a modulus $q_0 \cdots q_{t'} r$ from $q_0 \cdots q_{t'+2} r' g'$. Precisely, the constant is $\left\lceil \frac{g'\Delta' \cdot q_{t'+1} q_{t'+2}}{g\Delta} \right\rfloor = \frac{g'\Delta' \cdot q_{t'+1} q_{t'+2}}{g\Delta} + \delta$, where $\delta \in \left(-\frac{1}{2}, \frac{1}{2}\right]$. This factor is in the worst case $\gtrsim q_{t'+1}$. The induced error e_{adj} is given as follows:

$$e_{\mathrm{adj}} = \frac{\Delta'}{\Delta} \cdot e + \frac{\delta r' \Delta \cdot m}{r \cdot q_{t'+1} q_{t'+2}} + \frac{\delta r' e}{r \cdot q_{t'+1} q_{t'+2}} + e_{\mathrm{rs}}.$$

We note that the ratio $\frac{r'}{r \cdot q_{t'+1} q_{t'+2}}$ is at most inverse of word-size prime, ensuring that the error e_{adj} remains sufficiently small.

Case 3) Now, let's delve into the case when $t - t' = 1$. This case usually arises when the ciphertexts are in the consecutive integer levels, i.e., modulus resurrection is performed via a rescaling factor of $r'/r \cdot q_t$. We recall that the scaling factor Δ' satisfies $\Delta' = (r' \cdot \Delta^2)/(r \cdot q_t)$. Utilizing this relation, one can adjust ciphertext similar to the above cases. Precisely, we $\mathsf{Inv\text{-}RS}$ by g, then multiply $\left\lceil \frac{g'\Delta' \cdot q_{t'+1}}{g\Delta} \right\rfloor = \frac{g'\Delta' \cdot q_{t'+1}}{g\Delta} + \delta \approx \Delta$, where $\delta \in \left(-\frac{1}{2}, \frac{1}{2}\right]$. We then RS by $q_{t'+1}g'$, chaging the ciphertext modulus from $q_0 \cdots q_{t'+1}g'r'$ to $q_0 \cdots q_{t'}r'$ as desired. In this case, the error e_{adj} can be written as

$$e_{\mathrm{adj}} = \frac{\Delta'}{\Delta} \cdot e + \frac{\delta r' \Delta \cdot m}{r \cdot q_{t'+1}} + \frac{\delta r' e}{r \cdot q_{t'+1}} + e_{\mathrm{rs}} = \frac{\Delta'}{\Delta} \cdot e + \frac{\delta \Delta' \cdot m}{\Delta} + \frac{\delta r' e}{r \cdot q_{t'+1}} + e_{\mathrm{rs}},$$

whose size can be properly bounded if we manage the scaling factor size to be $\Delta \approx \Delta'$. When handling the general case where the ciphertexts may not be in the consecutive levels, one can adjust as follows: (1) we first adjust ct to the lower level where the Case 3 is applicable, (2) apply the adjusting algorithm, and possibly apply the Case 1.