이학박사 학위논문

Accelerating Homomorphic Computation through Machine-Efficient Arithmetic

(기계 친화적 산술 구조를 통한 동형 연산 가속화 연구)

2024년 12월

서울대학교 대학원

수리과학부

최 형 민

TODO: This page will be replaced with a signed copy.

Accelerating Homomorphic Computation through Machine-Efficient Arithmetic

(기계 친화적 산술 구조를 통한 동형 연산 가속화 연구)

지도교수 천 정 희

이 논문을 이학박사 학위논문으로 제출함

2025년 2월

서울대학교 대학원

수리과학부

최 형 민

최형민의 이학박사 학위논문을 인준함

2024년 12월

위 원 장	(인)
부 위 원 장	(인)
위 원	(인)
위 원	(인)
위 원	(인)

Accelerating Homomorphic Computation through Machine-Efficient Arithmetic

by

Hyeongmin Choe

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

in

Mathematics

Supervised by Professor Jung Hee Cheon

College of Natural Sciences
Department of Mathematical Sciences
Seoul National University

December, 2024

Abstract

최 형 민 (Hyeongmin Choe)
수리과학부 (Department of Mathematical Sciences)
The Graduate School
Seoul National University

The Residue Number System (RNS) variant of the Cheon-Kim-Kim-Song (CKKS) scheme (SAC 2018) has been widely implemented due to its computational efficiency. However, state-of-the-art implementations fail to use the machine word size tightly, creating inefficiency. As rescaling moduli are chosen to be approximately equal to the scaling factors, the machine's computation budget can be wasted when the scaling factors are not close to the machine's word size.

To solve this problem, we present a novel moduli management system called *Grafting*, that fills the ciphertext moduli with word-sized factors while allowing arbitrary rescaling. Grafting speeds up computations, reduces memory footprints, and makes universal parameters that can be used regardless of target precision. The key ingredient of Grafting is the *sprout*, which can be repeatedly used as a part of the ciphertext modulus. In general, when using an arbitrary ciphertext modulus, several copies of evaluation keys in various moduli are required. However, sprout allows rescaling by various amounts with the same keys, which can be extended to *universal sprout* that accommodates rescaling by an arbitrary amount. By breaking the dependency of the rescaling amount, scale factors, and RNS modulus, Grafting resolves restrictions in previous RNS-CKKS implementations.

Based on the new arithmetic introduced by Grafting, we revisit the homomorphic computations on their efficiency in terms of execution times and modulus consumption. As a straightforward application of grafted arithmetic, techniques requiring especially small scaling factors, such as Tuple-CKKS multiplication (CCS 2023), can be addressed with better efficiency by more than a factor of two. Also, Grafting naturally supports arbitrary precisions, without changing the low-level parameters, which can be extended to adaptively changing the scaling factors during computation. Corresponding to each sub-procedure, we can reduce the modulus consumption by utilizing an optimized scaling factor. We generalize the Baby-Step-Giant-Step (BGSGS) polynomial evaluation to use multiple scaling factors, and study the polynomial approximation specific to the evaluation method, based on the Error Variance Minimization technique (Eurocrypt 2022).

We *grafted* a parameter preset in the HEaaN library and implemented a grafted and non-grafted RNS-CKKS library to measure the improvements with concrete experiments. Benchmark results demonstrate the significance of Grafting, showing a speed-up of 2.20× for homomorphic multiplications and 2.05× for bootstrapping compared to the non-grafted variant. The ciphertext and evaluation key sizes are also reduced at a similar rate, while the precision remains the same. Based on the grafted implementation, we demonstrate that the homomorphic polynomial approximation using multiple precisions indeed benefits modulus consumption with comparable execution time, concretely in the EvalMod operation.

Keywords : Cryptography, Ring Learning With Errors (RLWE), Homomorphic Encryption (HE), Residue Number System (RNS), Polynomial Approximation, Adaptive Precision Computation.
Student Number : 2019-24352

Contents

Chapter 1

Introduction

1.1 Background

The Cheon–Kim–Kim–Song (CKKS) scheme [CKKS17] is a Fully Homomorphic Encryption (FHE) scheme that allows approximate computation over real-valued encrypted data. In CKKS encoding, messages are multiplied by a *scaling factor* and rounded to ensure precision. The scaling factor is squared during multiplication, necessitating *rescaling* to revert its size back. This procedure not only scales down the scaling factor but also the ciphertext modulus, requiring *Bootstrapping* to recover the modulus after a series of multiplications.

The state-of-the-art CKKS implementations adopt the Residue Number System variant of CKKS (RNS-CKKS) [CHK$^+$19] for efficiency. In RNS-CKKS, the ciphertext modulus is a product of Number Theoretic Transform (NTT) primes, referred to as the *RNS moduli*. Thanks to the Chinese Reminder Theorem (CRT), arithmetic modulo $Q = \prod q_i$ is decomposed into computations modulo q_i, the NTT primes. However, since rescaling is allowed only for the NTT primes dividing the ciphertext modulus, then the primes must be as large as the scaling factors. This linkage leads to computational inefficiency, as the NTT primes can be much smaller than the word size. We elaborate on this difficulty to clarify the problem we try to solve in the paper.

Most implementations choose scaling factors to be as small as possible so as not to waste the modulus. Recall that modulus is a limited resource[1] that significantly affects the computational efficiency. In particular, it is important to efficiently distribute the limited modulus budget across different functionalities. As bootstrapping consumes a huge amount of modulus, we usually have a smaller amount of modulus reserved for homomorphic computations. Hence, saving modulus consumption is often a top priority, leading to choosing as small scaling factors as possible. On the other hand, the number of RNS moduli needs to be minimized for efficiency. As ciphertexts

[1]As CKKS relies on the Ring Learning With Errors (RLWE) assumption, the security reduces as we increase modulus. When we fix all the other RLWE parameters, there is a budget limit often estimated by the Lattice Estimator [APS15].

are represented, computed, and stored with respect to each RNS modulus, the computational and space complexity is proportional to the number of RNS moduli constituting the ciphertext modulus. In this regard, the efficiency is maximized when the RNS moduli are chosen to be as close as possible to the machine word size. Note that this conflicts with the scaling factor minimizing strategy described earlier. As a result, the existing efficient implementations of CKKS choose one of the two possible options: 1) to fill the given maximum modulus budget with the machine word-sized NTT primes, minimizing the number of RNS moduli, or 2) to fill with the scaling factor size NTT primes, maximizing the multiplicative depth.

In the first option, we can use the RNS moduli of roughly the machine's word size. For instance, OpenFHE [ABBB$^+$22] supports the first option as default. The 1,579-bit ciphertext modulus comprises one 60-bit base primes, five 60-bit primes for switching key modulus, and twenty-one 58-bit primes. This choice can be used to evaluate any circuit whose target precision is less than the bootstrapping precision. However, this choice may waste the modulus budget when the required precision is low. Note that only 4 among the 21 available multiplicative depths can be used for computations, as the rest are for bootstrapping. The second option uses application-dependent scaling factors with respect to the target precision. In particular, one often chooses larger scaling factors for bootstrapping. For instance, Lattigo [EL23] and HEaaN [Cry22] support parameter presets whose scaling factors range from 34 bits to 51 bits, which is way smaller than the 64-bit word size. In FGb parameter of HEaaN, 1,555-bit ciphertext modulus is composed of a 58-bit base modulus, five auxiliary moduli of 59-60 bits, fifteen moduli of 42-58 bits for bootstrapping, and nine 42-bit moduli for homomorphic computations. Likewise, optimizing the scaling factors improves efficiency, especially for deep circuits, as bootstrapping is significantly more expensive than the other operations. The comparison between the two options can be described as a trade-off between flexibility and efficiency. The first option allows a single parameter set to cover broader applications than the second one, but is less computationally efficient. [2] The second option requires re-generating the public parameters and the evaluation keys depending on the circuits if one needs larger scaling factors.

Considering the conflict described above, it is tempting to break the link between RNS moduli and scaling factor. If we can freely choose them independently, we may choose word-sized RNS moduli and application-dependent scaling factors, speeding up the overall computation. For example, the Lattigo library has some default parameters with 12 different RNS moduli of sizes approximately 34 to 45 bits with a total modulus of 438 bits. This could be modified to use $438/62 \approx 7$ standard 64-bit NTT primes, accelerating roughly by a factor of

[2]As an efficiency measure of FHE schemes, we can consider *the running time of bootstrapping* divided by *the available multiplicative depths after bootstrapping*. A smaller value for the quantity indicates better performance. We can approximate it as $\frac{\text{mod}_{\text{ct}}}{\text{mod}_{\text{ct}}-\text{mod}_{\text{bts}}}$, where mod_{ct} represents the number of NTT primes for ciphertexts and mod_{bts} for bootstrapping. For the same bootstrapping precision, mod_{bts} remains constant. Thus, a larger mod_{ct} makes the quantity smaller.

$12/7 \approx 1.71$, while maintaining the same scaling factors and available multiplicative depths. In addition, the ciphertext and the key sizes can be reduced by a similar factor. Recently, [SS24] suggested a solution to the problem, but it requires additional evaluation and switching keys in each modulus, which is not a divisor of others. That is, we require several copies of the public key sets, significantly increasing the switching key sizes and communication costs. When the key set and the circuit to evaluate are not compatible, re-generating the keys is required. In this paper, we propose an effective solution (called *Grafting*) that introduces negligible overheads, introducing new arithmetic to RNS-CKKS.

1.2 Contributions and Results

1.2.1 Grafting, the New Arithmetic (Chapter 3)

Our first main goal is to decouple ciphertext moduli from scaling factors. This separation is designed to fully utilize the machine's computation and memory budget by using the word-sized RNS moduli, while keeping the same number of switching keys.

Grafting (Section 3.1)

As per the benefits of using word-sized RNS moduli, we propose *Grafting*, the ciphertext modulus management technique that decomposes the ciphertext modulus with the word-sized moduli as much as possible, while keeping the rescalibility among the ciphertext modulus. We decoupled the RNS moduli and the scaling factors by generalizing the rescaling procedure, allowing rescale between the moduli that are not multiples of each other. As the procedure rescales by a rational factor instead of an integer factor, we refer to it as *rational rescale*. To further allow efficient key switching without introducing an additional copy of the switching keys in different modulus, we suggest *sprout*, a small and reusable part of the ciphertext modulus. It has a size of the remainder of the ciphertext modulus size divided by the machine's word size. Sprouts allow us to use the same number of switching keys as before while keeping the RNS moduli mostly word-sized. We provide a grafted variant of the building blocks of RNS-CKKS, such as key switching and homomorphic multiplication. In addition, by extending the level adjustment techniques [KPP22], we introduce an adjusting technique for the ciphertexts with different scaling factors at different moduli, in Grafting.

We illustrate how the modulus changes in Grafting in Figure 1.1. The figure on the left shows the modulus descending scenario. Each multiplication consumes the modulus roughly by a scaling factor, while the word-sized moduli are maintained as many as possible. The number of non-empty gadget blocks is optimized during the whole procedure. The figure in the middle explains the rational rescale procedure, especially when the output sprout is larger than the input

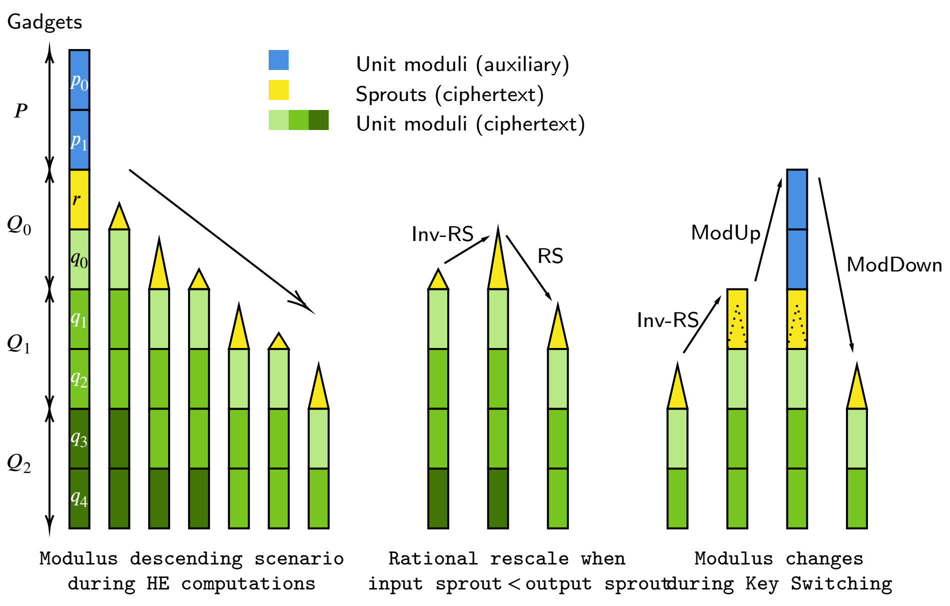

Figure 1.1: Modulus changes in Grafting.

sprout. In such cases, some parts of the sprouts are resurrected, making an intermediate modulus a multiple of the output modulus, allowing the conventional rescale. The right figure shows the key-switching procedure.

Universal Sprouts (Section 3.2)

Grafting enables the ciphertext modulus to be a product of as many word-sized moduli as possible; however, rational rescaling can only be done with selective amounts, depending on the choice of sprout moduli. Thus, we further extend the sprout and introduce the universal property to the sprouts. *Universal sprouts* are the sprouts that have divisors of roughly every arbitrary bit-length, which allows rational rescaling by any arbitrary bit-sizes. We suggest some candidates for universal sprouts, for instance, 2^{61} or $2^{15} \cdot r_1 \cdot r_2$, where $r_1 = 2^{16} + 1$ and $r_2 = 2^{30} - 2^{18} + 1$, which allows rescaling by any integer bit length with controllable errors. As the universal sprouts are required to have RNS modulus whose divisors have small bit-length differences, we demonstrate that the power-of-two moduli can be efficiently utilized in RNS-based implementations, proposing concrete modifications to the algorithms such as ModUp and ModDown.

Straightforward Applications (Section 3.3)

Grafting offers significant advantages by decoupling scaling factors from RNS moduli, addressing various limitations of previous implementations. It is effective across both low- and high-precision scenarios due to its flexible scaling factor sizes. Notably, Grafting eliminates the need for small NTT primes to accelerate parameters for special purposes, benefiting both composite rescaling and Tuple-CKKS multiplications [CCKS23]. This approach removes the requirement for special RNS moduli reserved for fixed precision bootstrapping, leading to hardware and compiler-friendly computations independent of homomorphic operation choices.

By simplifying FHE parameter selection, Grafting mitigates a traditional obstacle in FHE implementation and opens the door for further optimization techniques in HE applications. This flexibility allows the computation of arbitrary circuits with any precision, regardless of ciphertext modulus choice, removing the need for special primes or modulus reservations for specific tasks.

In [CCKS23], they proposed a novel CKKS multiplication that splits a ciphertext into tuples, namely the Tuple-CKKS multiplication. It 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 enables arbitrary k-bit arithmetic while fully utilizing the machine word size, resulting in the expected performance as mentioned in [CCKS23], close to the asymptotic analysis. Depending on the precision and number of slots, the improvement can reach a factor of 2 to 3.

Implementation (Section 3.4)

We implemented the grafted variant of RNS-CKKS in C++ using state-of-the-art techniques for efficiency. We also implemented the non-grafted variant of RNS-CKKS to compare performance; we will refer to it as a simple variant. For the simple variant, we use the level-optimized parameter set FTa of the HEaaN library [Cry22]. For the grafted variant, we use the same ciphertext modulus but with fewer RNS primes and a universal sprout including a power-of-two factor.

Grafting speeds up bootstrapping by a factor of 2.05× and reduces the ciphertext size by 40.0 % and key switching key sizes by 61.8 % as shown in Table 1.1; please see Section 3.4.2 for further details. Using Grafting, we also enhance bootstrapping, most notably allowing for an arbitrarily low output modulus and reducing the modulus consumption during EvalMod.

Table 1.1: Execution times and memory usage for the simple and grafted implementations of RNS-CKKS using the FTa parameter set from the HEaaN library [Cry22]; please see Section 3.4.2 for further details.

	Performance [ms]				Memory [KiB]	
	Mult	Tensor	Relin.	Bootstrap.	Ciphertext	Switching key
simple	298.04	9.86	260.03	13908.11	10240.1	112640.2
grafted	135.31	5.49	108.96	6794.56	6144.1	43008.2
Gain	2.20×	1.80×	2.39×	2.05×	−40.0 %	−61.8 %

1.2.2 Homomorphic Polynomial Approximation with Adaptive Precisions (Chapter 4)

TODO: add figures on multi-track/adaptive scaling factor evaluation.

Our second main goal is efficiently utilizing the new arithmetic, Grafting, in homomorphic computations. One of the key features of Grafting arithmetic is its support for homomorphic computations with arbitrary scaling factors. Grafting enables not only the use of scaling factors that exceed the size of the machine word or are too small to have enough NTT primes, but also the ability to change scaling factors during the computation. It allows to manage the computation precision and modulus consumption effectively. This involves using different scaling factors in sub-circuits and adapting precision based on computations, reducing ciphertext modulus consumption.

We focus on the homomorphic evaluation of polynomials approximating a given functionality, adaptively using the scaling factors, and how to approximate the functions with respect to this new methodology of evaluating polynomials.

Homomorphic Evaluation using Adaptive Precisions

We present a new approach for homomorphic computation with real/complex-valued numbers with reduced modulus consumptions using adaptive precision. That is, multiple scaling factors are used for evaluating a polynomial, possibly in parallel. We introduce a variant of the Baby-Step-Giant-Step (BSGS) algorithm with Chebyshev polynomials, enabling polynomial evaluations with adaptive scaling factors and optimizing the modulus use by strategically choosing scaling factors based on computation depth and coefficient size. Error analysis based on the error variance minimization technique [LLK$^+$22] shows how to approximate a given function to obtain a good approximation with respect to the adaptive scaling factors to lower the modulus consumption.

Application to EvalMod

This section discusses EvalMod, a component of CKKS bootstrapping that evaluates the modulo function homomorphically. EvalMod approximates the modulo operation using a polynomial, specifically obtained from Remez mini-max approximation on the cosine function, which is not friendly to the adaptive polynomial evaluation and modulus consumption reduction. We, instead, introduce how to approximate the EvalMod function beneficial to the adaptive evaluation, directly approximating it with the variant of error variance minimization. Concretely, for given scaling factors to evaluate the modulo function, the maximum degree, ring dimension, and the secret key Hamming weight, we can obtain an optimal polynomial approximating the modulo function with respect to the evaluation method using the scaling factors.

Implementation

We implemented the polynomial evaluation with adaptive scaling factors based on the grafted CKKS implementation. TODO: how slow, how much modulus saved, how much precision.

1.3 Related Works

RNS-CKKS

After the first RNS-CKKS scheme was introduced [CHK+19], most of the currently available homomorphic encryption libraries implementing the CKKS scheme only focus on the RNS version of it due to its efficiency [EL23, ABBB+22, Cry22, HHS+21, SEA23]. In RNS-CKKS, the ciphertexts and the switching keys are decomposed into a small integer modulo for each RNS modulus, and the computation is done in the decomposed format, thanks to the CRT. From its RNS-friendly nature, Han and Ki [HK20] introduced an advanced key-switching technique for RNS-CKKS, adopted from [BEHZ16, GHS12], which gave the flexibility of choosing the usable ciphertext modulus compensating the size of the switching keys and the key switching running times. Using the so-called RNS gadget decomposition, the ciphertexts are decomposed into several blocks and multiplied with the corresponding switching keys. By elaborately designing the choice of RNS moduli and the gadget blocks for each operation, some recent works [KPK+22, KPP22, BMTH21] and libraries [EL23, Cry22] achieve efficiency gains. However, the size of the RNS moduli was set far from the machine word sizes, making it hard to fully utilize the machine's architecture.

Approaches Filling the Machine Word Sizes

The idea to use word-sized primes mostly and a few small primes in the RNS moduli was first proposed by Gentry, Halevi, and Smart [GHS12], but missing some details. In the BGV scheme [HS20], they introduce a method of choosing word size primes and some smaller primes, enabling the ciphertext modulus to be fairly close to any desired target value. For key-switching, they switch the modulus by putting more word-sized primes and dropping non-word-sized primes, which maintains the message the same. 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.

When focusing on CKKS, Mono et al. [MG23] adopted the nature of using mostly word-sized primes from the Double-CRT format in BGV [HS20] to RNS-CKKS. They constitute the RNS moduli with mostly word-sized primes and a few small primes of the same size, whose product is close to the multiple of the word-sized primes. For example, three smaller primes of 36-bit can replace two 54-bit primes. This can be seen as a special type of Grafting, in a restricted manner. In Grafting, such restrictions were solved by setting the sprouts to a machine word size and introducing the universal sprouts.

Recently and concurrently, Samardzic and Sanchez [SS24] suggested a similar concept of modulus managing technique, focusing on designing a hardware accelerator. Given the largest possible modulus and a sequence of desired rescaling factors, they find a descending chain of ciphertext moduli, where each modulus is a composition of word-sized NTT primes with possibly several smaller NTT primes. They utilize the rational rescale to move between the consecutive moduli in the chain, restricting the rank of the gadgets and introducing a significant additional communication cost, which may be in an order of magnitude, depending on the circuits to evaluate. In Grafting, we avoid such inefficiencies. We keep the same number of keys, each having smaller sizes than before, while allowing universal use of the parameters not to re-generate the keys depending on the applications.

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

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

Machine-dependent Approaches

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

Polynomial Approximations for CKKS

As CKKS supports homomorphic polynomial evaluations, non-polynomial functions are usually approximate to a polynomial in a certain approximation range and errors. Cheon et al. [CKKS17, CHK+18] introduced using polynomial approximations based on the Taylor series expansion and applied it to CKKC bootstrapping. However, the Taylor approximations target a single narrow interval for the approximating range, and periodic functions like modulo operations or cosine or sine functions are unstable. To overcome this, Chen et al. [CCS19] utilized the Chebyshev approximation using the Chebyshev trigonometric polynomials, making the sine function approximation more accurate and stable. Extending this, Han et al. [HK20] introduced choosing Chebyshev points for Remez approximation depending on the intervals. This methodology was optimized by Lee et al. [LLL$^+$21] by using the multi-interval optimal minimax approximation. Bossuat et al. [BMTH21] suggest an error-reduced variant of the Baby-Step Giant-Step (BSGS) polynomial evaluation technique, and Lee et al. [LLK$^+$22] introduced a variant reducing the number of relinearizations, so-called lazy-BSGS, and suggested the error variance minimization technique to approximate the $\mathsf{EvalMod}$ function directly.

1.4 List of Papers

The results of the following papers are included.

1. Jung Hee Cheon, Hyeongmin Choe, Minsik Kang, Jaehyung Kim, Seonghak Kim, Johannes Mono, and Taeyeong Noh, Grafting: Complementing RNS in CKKS, Cryptology ePrint Archive, Paper 2024/1014 (2024), https://eprint.iacr.org/2023/771. In submission.

2. Hyeongmin Choe, (Working Title) Reduced Modulus Consumption for Homomorphic Polynomial Approximation Using Adaptive Precisions and its Application to Bootstrapping. 2024. Manuscript in preparation.

Chapter 2

Background

2.1 Notation

Polynomials and vectors are denoted in bold font and lowercase letters. Every vector is a column vector. Matrices are denoted in bold font and uppercase letters.

We let $\lfloor y \rceil$ be a rounding of $y \in \mathbb{R}$ to the nearest integer. We naturally extend the rounding notation to vectors and polynomials by applying it component-wise. For an integer n, we denote a set of non-negative integers smaller than n as $[n]$, i.e., $[n] = \{0, 1, \cdots, n-1\}$. For positive integer r and Q, we denote $\mathrm{ord}_r\, Q$ as the largest integer k such that r^k divides Q.

We let $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ be a polynomial ring where N is a power-of-two integer. For any positive integer Q, let the quotient ring $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R} = \mathbb{Z}_Q[X]/(X^N + 1)$. For a positive integer q and a polynomial $\mathbf{a} \in \mathcal{R}$ (or $\mathcal{R}_Q$ for some $q \mid Q$), we let $[\mathbf{a}]_q$ be a representative of $\mathbf{a}$ mod q in $\mathcal{R}_q$.

We define an isomorphism $\Phi : \mathbb{R}[X]/(X^N + 1) \to \mathbb{C}^{N/2}$ by $\mathbf{m}(X) \mapsto \left(\mathbf{m}(\zeta^{5^j})\right)_{0 \le j < N/2}$, where $\mathbf{m} \in \mathbb{R}[X]$ and $\zeta = e^{\frac{\pi i}{2N}}$ is a primitive $2N$-th root of unity.

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 Mathematical Backgrounds

2.2.1 Chinese Remainder Theorem

As it is computationally costly to handle integers modulo large modulus $Q \in \mathbb{Z}$, or $\mathbb{Z}_Q$, we can use the Chinese Remainder Theorem (CRT) to represent the elements in a parallel way. Let $Q = q_1 \cdot q_2 \cdots q_k$ for some pairwise coprime integers $q_1, \ldots, q_k$. Then there is a ring isomorphism, via the so-called CRT and inverse-CRT (iCRT) maps,

$$\begin{array}{ccc} \mathbb{Z}_Q & \cong & \mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_k} \\ a \bmod Q & \leftrightarrows & (a \bmod q_1, \cdots, a \bmod q_k). \end{array}$$

By computing the ring operations in a split and parallel state, the multiplication complexity significantly decreases from $\mathcal{O}(\log^2 Q)$ to $\mathcal{O}(\sum \log^2 q_i)$ which is roughly $\mathcal{O}(\log^2 Q/k)$.

The CRT and iCRT can be naturally extended to polynomial rings, between $\mathcal{R}_Q$ and $\mathcal{R}_{q_1} \times \cdots \times \mathcal{R}_{q_k}$.

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

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

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

NTT is also for computations with reduced complexity, which reduces the polynomial multiplication complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$.

2.2.3 Chebyshev Polynomials

The Chebyshev polynomials (of first kind) are a sequence of polynomials that form a complete orthogonal system. It can be defined in several ways, using trigonometric functions,

$$T_n(\cos\theta) = \cos(n\theta),$$

for all $\theta \in \mathbb{R}$ or using the recurrence relation:

$$\begin{aligned} T_0(x) &= 1, \\ T_1(x) &= x, \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \text{ for } n \geq 1. \end{aligned}$$

As mentioned above, the polynomials are orthogonal with respect to the integration:

$$\langle f, g \rangle := \int_{-1}^{1} f(x)g(x)\frac{dx}{\sqrt{1-x^2}}. \tag{2.2.1}$$

The Chebyshev polynomials also satisfy the following relation, which is frequently used for evaluating them:

$$T_m(x)T_n(x) = \frac{T_{m+n}(x) + T_{|m-n|}(x)}{2},$$

for all $m, n \geq 0$.

As the Chebyshev polynomials form an orthogonal basis, an infinite sum $\sum_n^\infty c_n T_n(x)$ is called the Chebyshev series or expansion, and the partial sums can be used for polynomial approximation in $[-1, 1]$.

2.2.4 Remez Mini-max ApproximationAlgorithm

The Remez algorithm is an iterative method for finding the minimax approximate polynomial of a continuous function f over an interval $[a, b]$. It ensures that the resulting polynomial is guaranteed to converge to the minimax approximate polynomial, which is numerically optimal in the sense of maximum approximation error.

Starting with an initial set of reference points $\{x_1, \ldots, x_{n+1}\}$, it computes a polynomial $p(x)$ such that the error $p(x_i) - f(x_i) = (-1)^i E$ alternates in sign, forming a system of $n + 1$ linear equations. The algorithm then identifies the zeros of the error and $n + 1$ extreme points where the error alternates between maximum and minimum values. If the so-called equioscillation condition is met at these extreme points, $p(x)$ is the minimax polynomial. Otherwise, the reference points are updated, and the process repeats. The algorithm is guaranteed to converge to the minimax polynomial.

2.3 Homomorphic Encryption

We first recall the formalisms of Homomorphic Encryption.

Definition 2.3.1 (Homomorphic Encryption)**.** *A homomorphic encryption scheme* HE *is a tuple of PPT algorithms* $\mathsf{HE} = (\mathsf{HE.KeyGen}, \mathsf{HE.Enc}, \mathsf{HE.Eval}, \mathsf{HE.Dec})$ *defined as follows:*

- $\mathsf{HE.KeyGen}(1^\lambda) \rightarrow (\mathsf{pk}, \mathsf{sk})$*: given input the security parameter* λ*, the* KeyGen *algorithm outputs a key pair (*pk, sk*),*

- $\mathsf{HE.Enc}(\mathsf{pk}, \mu) \rightarrow \mathsf{ct}$*: given input a public key* pk *and a plaintext message* $\mu \in \mathcal{P}$*, the encryption algorithm outputs a ciphertext* ct*,*

- $\mathsf{HE.Eval}(\mathsf{pk}, C, \mathsf{ct}_1, \dots, \mathsf{ct}_k) \to \mathsf{ct}_{res}$: *given input a public key* pk*, a circuit* $C : \{0,1\}^k \to \{0,1\}$*, and a tuple of ciphertexts* $\mathsf{ct}_1, \dots, \mathsf{ct}_k$*, the evaluation algorithm outputs an evaluated ciphertext* ct_{res}*,*

- $\mathsf{HE.Dec}(\mathsf{pk}, \mathsf{sk}, \mathsf{ct}_{res}) \to \mu_{res}$: *given input a public key* pk*, a secret key* sk *and a ciphertext* ct_{res}*, the decryption algorithm outputs a message* $\mu_{res} \in \mathcal{P}$*.*

If there is no limit on the depth of the circuit, we say HE *the fully homomorphic encryption.*

Definition 2.3.2 (Correctness). *We say that an* HE *scheme is correct if for all* λ*, depth bound* d*, circuit* $C : \mathcal{P}^k \to \mathcal{P}$ *of depth at most* d*, and* $\mu_i \in \mathcal{P}$ *for* $i \in [k]$*, the following holds: for* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{HE.KeyGen}(1^\lambda, 1^d)$, $\mathsf{ct}_i \leftarrow \mathsf{HE.Enc}(\mathsf{pk}, \mu_i)$ *for* $i \in [k]$, $\mathsf{ct}_{res} \leftarrow \mathsf{HE.Eval}(\mathsf{pk}, C, \mathsf{ct}_1, \dots, \mathsf{ct}_k)$*, we have*

$$\Pr[\mathsf{HE.Dec}(\mathsf{pk}, \mathsf{sk}, \mathsf{ct}_{res}) = C(\mu_1, \dots, \mu_k)] = 1 - \lambda^{-\omega(1)}.$$

Definition 2.3.3 (Security). *We say that an* HE *scheme is* IND-CPA*-secure if for all* λ *and depth bound* d*, the following holds: for any adversary* $\mathcal{A}$ *with run-time* $2^{o(\lambda)}$*, the following experiment outputs* 1 *with probability* $2^{-\Omega(\lambda)}$*.*

1. *On input the security parameter* λ *and a depth bound* d*, the challenger runs* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{HE.KeyGen}(1^\lambda, 1^d)$ *and* $\mathsf{ct}_i \leftarrow \mathsf{HE.Enc}(\mathsf{pk}, \mu_i)$ *for* $i \leftarrow \{0,1\}$ *where* $\mu_i \in \mathcal{P}$ *is chosen by the adversary; it provides* $(\mathsf{pk}, \mathsf{ct}_b)$ *to* $\mathcal{A}$ *for some* $b \leftarrow \{0,1\}$*.*
2. *Adversary* $\mathcal{A}$ *outputs a guess* b'*; the challenger outputs* 1 *if* $b = b'$*.*

2.3.1 CKKS Scheme

The Cheon-Kim-Kim-Song (CKKS) scheme is an approximate HE scheme that can handle real-valued data [CKKS17]. CKKS utilizes a unique encoding and decoding structure designed specifically for handling real-valued data.

The complex-valued message $\mathbf{z} \in \mathbb{C}^{N/2}$ (or $\mathbb{R}^N$) with $\|\mathbf{z}\|_\infty \le 1$ can be scaled by a scaling factor Δ, and embedded to a polynomial in $\mathbb{R}[X]/(X^N + 1)$ by the isomorphism Φ. Then, this can be rounded to a plaintext $m \in \mathbb{Z}[X]/(X^N + 1)$. We call this mapping and its approximate inverse as encoding (Encode) and decoding (Decode), formally defined as:

$$\mathsf{Encode}(\mathbf{z} \in \mathbb{C}^{N/2}; \Delta \in \mathbb{R}) = \lfloor \Phi^{-1}(\Delta \cdot \mathbf{z}) \rceil_{\mathcal{R}} \in \mathcal{R},$$
$$\mathsf{Decode}(\mathsf{pt} \in \mathcal{R}; \Delta \in \mathbb{R}) = \Delta^{-1} \cdot \Phi(\mathsf{pt}) \in \mathbb{C}^{N/2},$$

where the plaintext space becomes $\mathcal{P} = \mathcal{R}$ with bounded canonical norm. Basic algorithms in the CKKS scheme can be described as follows, which is based on RLWE encryption:

- $\mathsf{KeyGen}(1^\lambda)$: Sample s uniformly from $\{0, \pm 1\}^N$ with Hamming weight h (i.e., $\|s\|_1 = h$), and set secret key $\mathsf{sk} \leftarrow (1, s) \in \mathcal{R}^2$. Sample u, u' and u_i's uniformly from $\mathcal{R}_q$ and e, e' and e_i's from an error distribution χ and set public key $\mathsf{pk} \leftarrow (-u \cdot s + e, u) \in \mathcal{R}_q^2$. For an integer P, set evaluation key $\mathsf{evk} \leftarrow (-u' \cdot s + e' + Ps^2, u') \in \mathcal{R}_{Pq}^2$ and rotation keys $\mathsf{rotk}_i \leftarrow (-u_i \cdot s + e_i + Ps^{5^i}, u_i) \in \mathcal{R}_{Pq}^2$.

- $\mathsf{Enc}(\mathsf{pt}; \mathsf{pk})$: First sample $v \leftarrow \mathcal{R}_q$ uniformly and compute $\mathsf{ct} = v \cdot \mathsf{pk} + (\mathsf{pt} + e_0,\ e_1) \in \mathbb{R}_q^2$, where the error e_0 and e_1 are sampled from χ.

- $\mathsf{Dec}(\mathsf{ct}; \mathsf{sk})$: For $\mathsf{ct} = (c_0, c_1)$ and $\mathsf{sk} = (1, s)$, compute $\mathsf{pt} = c_0 + c_1 \cdot s$.

- $\mathsf{Add}(\mathsf{ct}_1, \mathsf{ct}_2)$: For two ciphertexts ct_1, ct_2, compute $\mathsf{ct}_{\mathsf{add}} = \mathsf{ct}_1 + \mathsf{ct}_2 \mod q$.

- $\mathsf{Mult}(\mathsf{ct}_1, \mathsf{ct}_2; \mathsf{evk})$: For two ciphertexts $\mathsf{ct}_1 = (c_0, c_1)$ and $\mathsf{ct}_2 = (c_0', c_1')$, compute $\mathsf{ct}_{\mathsf{mult}} = (c_0 \cdot c_0',\ c_0' \cdot c_1 + c_0 \cdot c_1') + \lfloor \frac{c_1 \cdot c_1' \cdot \mathsf{evk}}{P} \rceil \in \mathcal{R}_q^2$.

- $\mathsf{Rot}_i(\mathsf{ct}; \mathsf{rotk}_i)$: For $\mathsf{ct} = (c_0, c_1) \in \mathcal{R}_q^2$, compute $\mathsf{ct}_{\mathsf{rot},i} = (c_0, 0) + \lfloor \frac{c_1 \cdot \mathsf{rotk_i}}{P} \rceil \in \mathcal{R}_q^2$.

2.3.2 RNS-CKKS Scheme

RNS-CKKS scheme is an RNS variant of the CKKS scheme, first introduced in [CHK+19]. The ciphertext and the switching key modulus comprise NTT primes, constituting the RNS 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 $\ell \in [L+1]$. The switching key modulus is $PQ_{\max}$, where $P = p_0 \cdots p_{K-1}$, where p_j's are relatively prime NTT primes. In addition, q_i's and p_j's are relatively prime. The polynomials in $\mathcal{R}_Q$ are stored and computed in RNS, i.e., for $a \in \mathcal{R}_Q$, we indeed have $[\mathbf{a}]_{q_i} \in \mathcal{R}_{q_i}$ for every RNS 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 [CHK+19]

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 $\mathbb{Z}_P$ as

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

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

Modulus Switchings

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

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

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

We note that the ModUp operation maps $\mathbf{a}\in\mathcal{R}_Q$ to $\mathbf{a}+Q\mathbf{e}\in\mathcal{R}_{PQ}$, where $|\mathbf{e}|\le\ell/2$ from the fast basis conversion. The ModDown operation maps $\mathbf{a}=([\mathbf{a}]_P,[\mathbf{a}]_Q)\in\mathcal{R}_{PQ}$ to $\mathbf{a}'=P^{-1}\cdot(\mathbf{a}-\tilde{\mathbf{a}})\in\mathcal{R}_Q$, where $\tilde{\mathbf{a}}\equiv\mathbf{a}\pmod P$ and $\|\tilde{\mathbf{a}}\|_\infty\le(k/2)\cdot P$, resulting $\|\mathbf{a}'-P^{-1}\cdot\mathbf{a}\|_\infty=P^{-1}\cdot\|\tilde{\mathbf{a}}\|_\infty\le k/2$. RS introduces an error of size $\le 1/2$. When applied to ciphertext, the error becomes multiplied by the secret key $\mathbf{s}$ and thus has an infinity norm of $\le k/2\cdot(\|\mathbf{s}\|_1+1)$ and $\le 1/2\cdot(\|\mathbf{s}\|_1+1)$, respectively, for ModDown and RS. Let us use the notation $Q\to PQ$ (Resp. $PQ\to Q$) instead of $C\to\mathcal{D}$ (Resp. $\mathcal{D}\to C$) if there is no confusion. We note that in [AAB+23], it is demonstrated that rather than processing the ModDown at once, splitting the procedure into two or more steps – such as from PQ to p_0Q, and then Q – can reduce the output error by a factor of k with only small additional costs for Hadamard multiplications.

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

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:

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

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

Gadget Decomposition, Key Switching, and Multiplication

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

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

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

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

Let $P = p_0 \cdots p_{K-1}$. Then the switching keys are $\{\mathsf{swk}_i\}_{i \in [\mathsf{dnum}]} = \{(\boldsymbol{\beta}_i, \boldsymbol{\alpha}_i)\}_{i \in [\mathsf{dnum}]} \in \mathcal{R}_{PQ}^{2\times\mathsf{dnum}}$, where

$$\boldsymbol{\beta}_i = -\boldsymbol{\alpha}_i \cdot \mathbf{s} + P \cdot \hat{Q}_i \cdot [\hat{Q}_i^{-1}]_{Q_i} \cdot \mathbf{s}' + \mathbf{e}_i \in \mathcal{R}_{PQ},$$

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 also with a larger switching key size.

Key switching. Key switching of a ciphertext $\mathsf{ct} \in \mathcal{R}_Q^2$, denoted as $\mathsf{KeySwitch}_{\mathsf{swk}}(\mathsf{ct})$, involves the following procedures[1], requiring $(d+2)(\ell+K+1) \approx (d+3+2/d)(\ell+1)$ (i)NTTs [MG23].

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.

2. ModUp each components of the ciphertext in modulus Q_i, to $PQ_0 \cdots Q_{d-1}$ for $i \in [d]$, resulting in d ciphertexts in modulus $PQ_0 \cdots Q_{d-1}$.

3. External product part of each ciphertext with swk_i for $i \in [d]$, and add the remaining parts of the ciphertext.

4. ModDown from modulus $PQ_0 \cdots Q_{d-1}$ to Q.

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 $\mathbf{s}^2$ to $\mathbf{s}$ via the rlk, then RS by q_ℓ. Note, the tensor of ciphertexts ct_1 and ct_2 satisfies $\langle \mathsf{ct}_1 \otimes \mathsf{ct}_2, (1, s, s^2)\rangle = \langle \mathsf{ct}_1, (1, s)\rangle \cdot \langle \mathsf{ct}_2, (1, s)\rangle$, where $\mathsf{sk} = (1, s)$.

Level Adjustments

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

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

1. Let $\mathsf{ct} = [\mathsf{ct}]_{q_0 \cdots q_{\ell'+1}} \in \mathcal{R}^2_{q_0 \cdots q_{\ell'+1}}$ 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 $q_{\ell'+1}$.

[1]Sometimes, Inv-RS is explained as a part of the ModUp procedure; however, we split them for a detailed explanation.

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

2.4 Bootstrapping in CKKS

CKKS bootstrapping

In general, the CKKS bootstrapping is used to lengthen the multiplicative budget for further computations as much as possible, since the modulus is consumed after each homomorphic multiplication. It proceeds as follows[2] During bootstrapping, we homomorphically decode the ciphertext, raise the modulus, encode the ciphertext again, and apply a reduction modulo the base modulus, referred to as SlotsToCoeffs (StC), ModRaise, CoeffsToSlots (CtS), and EvalMod:

- SlotsToCoeffs (StC): moves the messages from slots to coefficients, resulting in a ciphertext $(\mathbf{a}, \mathbf{b}) \in \mathcal{R}^2_q$ satisfying $\mathbf{b} = \mathbf{a}\mathbf{s} + \mathbf{m} + \mathbf{e} + q\mathbf{I}$ for some $\mathbf{I} \in \mathcal{R}$ and error $\mathbf{e}$, where $\mathbf{m}$ is the message,
- ModRaise: raises the ciphertext modulus from q to Q, resulting in $(\mathbf{a}', \mathbf{b}') \in \mathcal{R}^2_Q$, where $(\mathbf{a}', \mathbf{b}') = (\mathbf{a}, \mathbf{b}) \bmod q$, which satisfies $\mathbf{b} = \mathbf{a}\mathbf{s} + \mathbf{m} + \mathbf{e} + q\mathbf{I} \bmod Q$,
- CoeffsToSlots (CtS): moves the messages back to slots, having $\mathbf{m} + \mathbf{e}' + q\mathbf{I}$ in slots, where $\mathbf{e}'$ is a new error,
- EvalMod: homomorphically evaluates a polynomial, approximating the modulo q operation over an interval $[(-K+1)q, (K-1)q]$.

The last EvalMod step is to remove the unnecessary $q\mathbf{I}$ from the message, by (approximately) applying the modulo q operation. As it is a periodic function, we set a high-probability interval for the message $\mathbf{m} + \mathbf{e}' + q\mathbf{I}$, then approximate the function to a polynomial. The state-of-the-art implementation uses Chebyshev approximations on the sine or cosine functions with a scaled interval, then applies double angle formula [HK20, BMTH21, BTH22], or use direct approximation of modulo q function using error variance minimization [LLK+22].

[2]There is the other option for the orders of the sub-procedures, starting from ModRaise and ending with StC, but this makes only a little difference for our attack.

Chapter 3

Grafting, the New Arithmetic

3.1 Grafting

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 component is computed with the machine's word size. Therefore, if we can reduce the number of RNS components representing the ciphertext, the whole FHE computations will benefit from a straightforward speed-up by roughly the reduced ratio, depending on the ciphertext modulus. In particular, the number of operations, such as (i)NTT and the fast basis conversions, the tensor, and the external products, roughly follows the ratio with some additional costs. Thus, FBC, tensor products, and external products will be decreased by the ratio. An appealing approach is to use the word size primes for the RNS moduli. However, it compromises the available multiplicative depths, compared to the cases when the moduli are set approximately the same as the scaling factors, say Δ, which varies (from 20 to 120 in general) on the target message precision.

In this section, we introduce *Grafting*, a method of using word-sized primes for RNS moduli while allowing rescale by a smaller factor. We first introduce a tool for switching the ciphertext modulus to a non-divisor modulus, namely, *Rational Rescaling* in Section 3.1.1. We then introduce how to maintain the modulus of ciphertexts, which is filled with word-sized NTT primes, as much as possible, namely, *modulus resurrection* with sprouts in Section 3.1.2. Finally, we introduce how to decompose the modulus into gadgets for key switching in Section 3.1.3 and how to adjust the ciphertexts with different moduli in Section 3.1.4 in the Grafting scenario.

3.1.1 Rational Rescale: Rescale with non-divisor

In this section, we decouple the RNS moduli from the scaling factors and propose a method to rescale a ciphertext from a modulus Q to another modulus Q', where Q' does not necessarily divide Q, which we call the *Rational Rescaling*. The rational rescale procedure rescales an input ciphertext in modulus Q by a rational number $Q/Q' \in \mathbb{Q}$ to an output ciphertext modulo $Q/(Q/Q') = Q'$, rescaling the scaling factor as well. Concretely, we define the rational rescale operation as follows.

Definition 3.1.1 (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 denote the *rational* rescale as well as the *integer* rescale, i.e., the conventional rescaling by an integer. 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.

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 moduli. Thus, it is worth minimizing the RNS moduli factors of $\mathrm{lcm}(Q, Q')/Q'$ to reduce the error.

Theorem 3.1.1 (Rational Rescale Correctness)**.** *For given a ciphertext* $\mathsf{ct} \in \mathcal{R}_Q^2$ *and* $Q' \nmid Q$*, it holds that* $\left[\langle \mathsf{RS}_{Q/Q'}(\mathsf{ct}), \mathsf{sk}\rangle\right]_{Q'} = Q'/Q \cdot \left[\langle \mathsf{ct}, \mathsf{sk}\rangle\right]_Q + e_{res}$ *for some rescale error* $\mathbf{e}_{res}$ *satisfying* $\|\mathbf{e}_{res}\|_\infty \le \ell/2 \cdot (\|\mathbf{s}\|_1 + 1)$*, where* ℓ *is the number of RNS blocks in* $\mathrm{lcm}(Q, Q')/Q'$*, and* $\mathbf{s}$ *be a secret key.*

Proof. Let us follow the notations in Definition 3.1.1. Let $[\langle \mathsf{ct}, \mathsf{sk}\rangle]_Q = \Delta \cdot \mathbf{m} + \mathbf{e}$, or $\langle \mathsf{ct}, \mathsf{sk}\rangle = \Delta \cdot \mathbf{m} + \mathbf{e} + Q\mathbf{I}$ for some $\mathbf{I} \in \mathcal{R}$. Then $\langle \mathsf{Inv\text{-}RS}_R(\mathsf{ct}), \mathsf{sk}\rangle = R\Delta \cdot \mathbf{m} + R\mathbf{e} + QR\mathbf{I}$. The final rescaling RS_S introduces an additional error e_{res}, where $\|e_{\mathrm{res}}\|_\infty \le \ell/2 \cdot (\|\mathbf{s}\|_1 + 1)$ for ℓ the number of RNS blocks in $S = \mathrm{lcm}(Q, Q')/Q'$. Precisely, we have

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

which concludes the proof. □

Rational rescale maps a ciphertext including a message with a scaling factor Δ^2 (after tensor) to a ciphertext whose message has a scaling factor $\Delta^2 \cdot Q'/Q \approx \Delta$, instead of $\Delta^2/q_\ell \approx \Delta$. We also note that the rational scaling factor can be tracked as in [KPP22]. Once we utilize rational rescaling to manage the scaling factor, the resulting ciphertext modulus Q' may not be a divisor of the switching key modulus $PQ_{\max}$. In this case, the original key-switching procedure does not apply to the ciphertext due to the changed modulus, which may not be compatible with the gadget decomposition.

One way to solve this problem is to prepare switching keys in various moduli, which are multiples of possible ciphertext moduli. However, this will increase the number of switching keys, degrading the communication cost for key transmissions. When no specific circuits are predetermined, a huge number of keys need to be transmitted for general circuit evaluations. Having multiple copies is too much because the switching keys are already hundreds to thousands of megabytes for FHE parameters.

Key Switching in Temporary Modulus

Another possible solution is temporarily switching the modulus to an intermediate modulus, a divisor of the switching key modulus, and performing key switching in the intermediate modulus. That is, for a ciphertext encrypting an encoded message $\Delta\mathbf{m}+\mathbf{e}$ in modulus Q, one can choose a modulus $Q_{\text{int}} \gtrsim Q$ satisfying $Q_{\text{int}} \mid PQ_{\max}$. Similar to the rational rescaling, one can switch the modulus of the ciphertext to Q_{int}, then apply key switching. The resulting ciphertext will encrypt an encoded message $(Q_{\text{int}}/Q) \cdot (\Delta\mathbf{m} + \mathbf{e}) + \mathbf{e}'$ with a switched secret key, where $\mathbf{e}'$ includes the rescale and the key switching errors. We can convert the modulus back to Q using the rational rescale procedure while maintaining the error within $\mathbf{e}$ plus the rescale error.

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

$$\begin{tikzcd}
 & Q_{\mathsf{swk}} \arrow[d, leftrightarrow, "\mathsf{KeySwitch}"] & Q_{\mathsf{swk}} \arrow[d, leftrightarrow, "\mathsf{KeySwitch}"] & Q_{\mathsf{swk}} \arrow[d, leftrightarrow, "\mathsf{KeySwitch}"] & \\
 & Q_{\text{int}} \arrow[d, leftrightarrow, "\mathsf{ModSwitch}"] & Q'_{\text{int}} \arrow[d, leftrightarrow, "\mathsf{ModSwitch}"] & Q''_{\text{int}} \arrow[d, leftrightarrow, "\mathsf{ModSwitch}"] & \\
\cdots \arrow[r] & Q \arrow[r] & Q' \approx Q/\Delta \arrow[r] & Q'' \approx Q'/\Delta \arrow[r] & \cdots
\end{tikzcd}$$

Figure 3.1: Visualization of the descending moduli chain ($\rightarrow \cdot \rightarrow \cdot \rightarrow$) and the changes in ciphertext modulus during key-switching ($\updownarrow$), where $Q^*_{\text{int}} \gtrsim Q^*$, $Q^* \nmid Q_{\mathsf{swk}}$, and $Q^*_{\text{int}} \mid Q_{\mathsf{swk}}$.

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

In addition, the temporary modulus switching, as well as the rational rescaling, induce additional Hadamard multiplication costs. They are not small, especially when the factors of the two moduli do not overlap much. Precisely, when assuming $\ell = \ell_1 + \ell_2$ factors in Q and

$\ell' = \ell_2 + \ell_3$ factors in Q' (or Q_{int}), where exactly the ℓ_2 factors are overlapped, the cost for changing the modulus from Q to Q' is $\mathcal{O}\left(N(\ell + (\ell - \ell_2) \cdot \ell')\right)$, where N is the ring dimension.

Outsourced Linear Transform

In the case when rescaling is unnecessary or can be delayed, the above scenario gets especially efficient. In homomorphic linear transformations where we do not have ciphertext-ciphertext multiplications, we can postpone rescaling until the end of the linear transformation and afford a slightly more expensive rational rescaling to reduce the expensive cost of the linear transform.

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 described as follows: 1) evaluate the homomorphic linear transform f and get $\mathsf{Eval}_f(\mathsf{ct}') \in R^2_{Q_{\text{in}}}$, then 2) perform rational rescale from Q_{in} to Q_{out} and get $\mathsf{ct}_{\text{out}} \in R^2_{Q_{\text{out}}}$.

To accelerate, we may use Q_{in} to be a product of word-size primes without sprouts and keep Q_{out} to be a normal RNS-CKKS modulus, which can be seen as outsourcing the computation from the original chain to a performant chain.

Recall that homomorphic linear transformations are often a bottleneck of CKKS. In particular, Slots-to-Coeffs (StC) and Coeffs-to-Slots (CtS) steps, which evaluate homomorphic (inverse) discrete Fourier transformations, often take more than half of CKKS bootstrapping. In this regard, this simple technique can largely improve the bootstrapping performance when applied to most of the RNS-CKKS libraries. We can literally fill up machine words in RNS with negligible overhead.

In particular, the CtS step of CKKS bootstrapping in FTa parameter of the HEaaN library [Cry22] uses 22 RNS moduli (primes) of 28-41 bits. When replacing the moduli chain into 12 moduli of 59-62 bits, we naïvely expect a speedup of $22/14 \approx 1.57$, assuming the same dnum.

3.1.2 Modulus Resurrection: Rescale in the word-sized moduli chain

We suggest a method to avoid generating more switching keys or introducing additional costs for key switching, the *Modulus Resurrection* technique, which reuses some factors of 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 previously. By doing so, the RNS moduli remain relatively prime, while the ciphertext modulus is kept to be a divisor of the switching key modulus. This can be viewed as fusing the ciphertext modulus chain and the temporary ciphertext modulus chain introduced in the previous subsection.

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

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

Specifically, we choose the maximal sprout to be similar to the machine word size $w = 2^{\omega}$. Here is an example of the sprout.

Example 1. *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 $r \in \{1, r_0, r_0 r_1\}$, where $r_0 r_1$ only appears in Q_{max}. From a modulus $Q = q_0 \cdots q_{\ell} \cdot r$, we can continually scale by ≈ 30 bits for e.g., rescale by r_i when $r_i | r$, and rational rescale by $q_{\ell}/r_1 \approx 2^{30}$ when $r = 1$.*

The resulting multiplication algorithm is specified in Algorithm 1. Compared to the conventional RNS-CKKS multiplications, there are two main differences: 1) the ciphertext moduli are changed into grafted moduli with the sprouts and 2) rescale (line 4) is replaced into the rational rescale to a corresponding modulus Q'. We note that only a few unit moduli are eliminated during rational rescale; the error has roughly the same magnitude as the original rescale error. We refer to Algorithm 2 for the detailed key switching procedures and Algorithm 4 for how to choose the output modulus Q'.

Algorithm 1 Homomorphic multiplication with modulus resurrection.

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

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

```
1: $(\mathbf{b}, \mathbf{a}, \mathbf{d}) \leftarrow \mathsf{ct}_1 \otimes \mathsf{ct}_2$        ▷ Tensor product ($\mathbf{b} + \mathbf{as} + \mathbf{ds}^2 \approx \Delta^2 \mathbf{m}_1 \mathbf{m}_2$)
2: $(\mathbf{b}', \mathbf{a}') \leftarrow \mathsf{KeySwitch}_{\mathsf{rlk}}(0, \mathbf{d})$        ▷ Relinearization ($\mathbf{b}' + \mathbf{a}'\mathbf{s} \approx \mathbf{ds}^2$)
3: $\mathsf{ct} \leftarrow (\mathbf{b}, \mathbf{a}) + (\mathbf{b}', \mathbf{a}')$
4: $\mathsf{ct} \leftarrow \mathsf{RS}_{Q/Q'}(\mathsf{ct})$        ▷ Rational rescale by $Q/Q' \approx \Delta$
   return $\mathsf{ct}$
```

The output ciphertext modulus Q' should roughly be Q/Δ to keep roughly the same scaling factor Δ. Thus, the typical choice of sprout could vary depending on the rescaling factors we will use. For instance, the sprouts in the above example allow us to rescale by only multiples of $\omega/2 = 30$ bits. This restricts the possible scaling factors for evaluating a circuit. Therefore, choosing an appropriate sprout for a circuit is important. We will introduce a sprout that can be used generally and universally, in Section 3.2.

3.1.3 Key Switching in Grafting

The key switching procedures, as well as the relinearization procedures, can be done as in the non-Grafted RNS-CKKS since all the ciphertext moduli Q are divisors of the switching key modulus $PQ_{\max}$. We decompose the largest ciphertext modulus $Q_{\max}$ into dnum number of similar-sized blocks $Q_0, \cdots, Q_{\mathsf{dnum}-1}$, known as the gadget blocks, where $P \gtrsim \max_i Q_i$. The sprout can be located in one of the gadget blocks. Depending on the gadget decomposition, the efficiency can vary.

For instance, one can place the sprout in the topmost gadget, $Q_{\mathsf{dnum}-1}$. Then, the number of gadgets for a ciphertext modulus $Q = r \cdot q_0 q_1 \cdots q_{\ell-1}$ can be larger than the optimal value $d = \lceil (\ell+1)/\alpha \rceil$ if $\alpha i \le \ell < \alpha(i+1) - 1$ and $i < \mathsf{dnum} - 1$ hold since

$$\begin{aligned} Q = q_0 q_1 \cdots q_{\ell-1} \cdot r &= (q_0 \cdots q_{\alpha-1}) \cdots (q_{\alpha(d-1)} \cdots q_{\ell-1}) \cdot r \\ &= Q_0 \cdots Q'_{d-1} \cdot Q'_{\mathsf{dnum}-1}, \end{aligned}$$

where $Q'_{d-1} \mid Q_{d-1}$ and $Q'_{\mathsf{dnum}-1} = r$. The number of gadget blocks of a ciphertext modulus highly affects the key switching cost, which is roughly linear in the number of gadgets. In the above case, a gadget block $Q_{\mathsf{dnum}-1}$ is used to cover only the sprout, and overall, $(d+1)$ blocks were used instead of d blocks, which leads to $\approx (d+1)/d$ slower performance. Note that the ratio is not so small, especially when Q is small.[1]

To achieve optimal efficiency, we put the sprouts into the bottom gadget block to minimize the number of non-empty gadget blocks, as follows.

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

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

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

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

[1]In the earlier version of this paper, we decomposed as above. To optimize, we suggested resurrecting the rightmost block when it is (almost) eliminated, replacing the second-right block. It introduces additional costs compared to the current version.

Following the original RNS-CKKS key-switching procedures described in Section 2.3.2, we describe the procedure with grafted ciphertext moduli in Algorithm 2. The correctness of the key switching procedures in grafted ciphertext moduli follows from the correctness of the rational rescale.

Algorithm 2 Key switching in grafted moduli.

```
Input: ct = $(\mathbf{b}, \mathbf{a}) \in \mathcal{R}_Q^2$ for $Q = Q_0' \cdot Q_1 \cdots Q_{d-2} \cdot Q_{d-1}'$ with a switching key swk = $\{\mathsf{swk}_i\} \in \mathcal{R}_{PQ_{\max}}^{2\times d}$.
Output: $\mathsf{ct}' \in \mathcal{R}_Q^2$ with a switched secret key.
1: $\mathbf{a} \leftarrow \mathsf{Inv\text{-}RS}_{Q_{\mathrm{inter}}/Q}(\mathbf{a})$        ▷ Inv-RS to $Q_{\mathrm{inter}} = Q_0 \cdots Q_{d-1}$
2: $\mathbf{a}_i \leftarrow \mathsf{ModUp}_{Q_i \to PQ_{\mathrm{inter}}}(\mathbf{a} \mod Q_i)$ for $i \in [d]$        ▷ ModUp to $PQ_{\mathrm{inter}}$
3: $\mathsf{ct} \leftarrow \sum_i \mathbf{a}_i \cdot (\mathsf{swk}_i \mod PQ_{\mathrm{inter}})$        ▷ External product in $PQ_{\mathrm{inter}}$
4: $\mathsf{ct}' \leftarrow \mathsf{ModDown}_{PQ_{\mathrm{inter}} \to Q}(\mathsf{ct})$        ▷ ModDown to $Q$
5: $\mathsf{ct}' \leftarrow \mathsf{ct}' + (\mathbf{b}, 0)$
   return $\mathsf{ct}'$
```

3.1.4 Modulus Adjustment

Modulus adjustment is a counterpart of the level adjustment in RNS-CKKS [KPP22]. The scaling factors are not fixed for each modulus, unlike the previous RNS-CKKS implementations, which had fixed scaling factors for each multiplicative depth. We can adjust two ciphertexts with different moduli and scaling factors before adding or multiplying them.

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

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

with a scaling factor Δ and the modulus $Q = q_0 \ldots q_{\ell-1} \cdot r$ for ℓ unit moduli q_i's and a sprout r. We aim to adjust this ciphertext to a ciphertext in modulus $Q' < Q$ with a scaling factor Δ'. We adapted and modified the level-adjusting method in [KPP22] to our scenario, which can be found in Algorithm 3. Note that the adjustments can be fused into a circuit, as in the level adjustment technique.

Algorithm 3 Modulus adjustment.

```
Input: $\mathsf{ct} \in \mathcal{R}_Q^2$ with scaling factor $\Delta$.
Output: $\mathsf{ct}' \in \mathcal{R}_{Q'}^2$ with scaling factor $\Delta'$, where $Q > Q'\Delta$.
1: Choose the smallest $Q_{\mathrm{mid}} \mid Q$ such that $Q \geq Q_{\mathrm{mid}} \gtrsim Q' \cdot \Delta$.
2: $\mathsf{ct} \leftarrow (\mathsf{ct} \mod Q_{\mathrm{mid}})$        ▷ Modulus reduction to $Q_{\mathrm{mid}}$
3: $\mathsf{ct} \leftarrow \mathsf{ct} \cdot \lceil (Q_{\mathrm{mid}} \cdot \Delta')/(Q' \cdot \Delta) \rfloor$        ▷ Integer multiplication in $Q_{\mathrm{mid}}$
4: $\mathsf{ct}' = \mathsf{RS}_{Q_{\mathrm{mid}}/Q'}(\mathsf{ct})$        ▷ Rational rescale to $Q'$
   return $\mathsf{ct}'$
```

We first choose an appropriate modulus Q_{mid} which is a divisor of Q to satisfy $Q \geq Q_{\text{mid}} > Q'$ and $Q_{\text{mid}} > Q' \cdot \Delta$. We reduce the ciphertext modulus to modulus Q_{mid}, by dropping some components. We then multiply an integer constant and rational rescale the ciphertext to the final modulus Q'. We note that the condition $Q_{\text{mid}} > Q' \cdot \Delta$ allows us to manage the error $\mathbf{e}_{\text{adj}}$ to be sufficiently small.

Theorem 3.1.2 (Modulus and Scaling Factor Adjustment Correctness)**.** *For given a ciphertext* $\mathsf{ct} \in \mathcal{R}_Q^2$ *with a scaling factor* Δ*, and a target ciphertext modulus* Q' *and* Δ' *that satisfy* $Q > Q'\Delta$*. Then, the output of Algorithm 3 is decrypted to the same message with* ct *with an additional rescale error.*

Proof. The integer constant can be represented as

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

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

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

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

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

Since $Q \geq Q_{\text{mid}} > Q' \cdot \Delta$ and $|\delta| < 1/2$, the infinite norm of the second and the third terms are bounded by $m_i/2$ and $e_i/(2\Delta)$. Thus, the scaled initial (which is inherent) and rescale errors become the most significant term. □

3.2 Universal Sprouts

As mentioned in Section 3.1.2, choosing an appropriate sprout for evaluating a specific circuit is important. In other words, a given sprout may not be used for other circuits, i.e., it cannot be used universally. For instance, the sprout in Example 1 allows us to rescale by only multiples of 30 bits. Another option is to have three 20-bit NTT primes or 10, 20, and 30-bit NTT primes so that rescaling by only the multiples of 20 or 10 is allowed, respectively.

In this section, we introduce sprouts that can be used universally, independent of the scaling factors and circuits. In Section 3.2.1, we define the universal sprout and suggest three candidates. As the sprouts include some power-of-two factors, we introduce the method exploiting the power-of-two modulus as RNS modulus in Sections 3.2.2. Finally, we suggest efficient methods for implementing the universal sprouts in the machine architectures in Section 3.2.3.

3.2.1 Universal Sprout and Rescalability

To overcome the restrictions on the scaling factors from the choice of sprouts, we introduce the *universal sprouts*, which can be used for any scaling factors. Let us start with the simplest example, which may be seen as a hybrid of the two CKKS instantiations using binary modulus and RNS.

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

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

where the top modulus is $Q_{max} = q_0 \cdots q_{L-1} \cdot 2^{60}$. From a ciphertext modulus of $\approx (60 \cdot \ell + \gamma)$ bits, one can rational rescale by $q_{\ell'} \cdots q_{\ell-1} \cdot 2^{(\gamma-\gamma')}$ to obtain a ciphertext modulus of $\approx (60 \cdot \ell' + \gamma')$ bits. This can be done regardless of whether $\gamma \geq \gamma'$ or not.

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

As in the above example, if the sprouts (i.e., the divisors of r_{top}) can approximately represent all the bit lengths from 1 to ω, we call the sprout *universal*. Universal sprouts allow rescaling roughly by any bit length, allowing it to be used universally, independent of the circuits.

We note that it is hard to efficiently handle a ciphertext modulo 2^{60} on a 64-bit machine. It requires a larger modulus to embed $\mathbb{Z}_{2^{60}}$ into it or a real/complex-valued Discrete Fourier transform (DFT) instead of NTT. Both introduce some implementation difficulties and inefficiencies since they require higher precisions than the machine word size. Roughly speaking, dealing with a ciphertext modulo 2^{60} is three times more expensive than dealing with a ciphertext modulo 60-bit NTT prime.

Thus, we introduce universal sprouts in Examples 3 and 4, which can be seen as practical extensions of power-of-two sprout in Example 2. They also include some power-of-two factors; however, they are much easier to handle as they are much smaller than the machine word size. The cost of handling the sprouts is roughly two times more expensive than a word-sized NTT prime. We refer to Section 3.2.3 for more details.

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

Example 3. *Let q_i's be* 61*-bit NTT primes and $r_{top} = 2^{15} \cdot r_1 \cdot r_2$, where r_1 is a* 16*-bit NTT prime, and r_2 is a* 30*-bit NTT prime. Typically, we can choose $r_1 = 2^{16} + 1$ and $r_2 = 2^{30} - 2^{18} + 1$, which are NTT primes for ring dimension $N \leq 2^{15}$.*[3] *Each sprout r can be represented as $r = 2^{\alpha} \cdot r_1^{\beta_1} \cdot r_2^{\beta_2}$, where $0 \leq \alpha \leq 15$, $\beta_i \in \{0, 1\}$. We note that the sprouts can represent any bit length from* 1 *to* 60 *as*

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

hence, it is a universal sprout.

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

Example 4. *Let q_i's be* 61*-bit NTT primes and $r_{top} = 2^{19} \cdot r_1 \cdot r_2$, where r_1 is a* 19*-bit NTT prime, and r_2 is a* 23*-bit NTT prime. Typically, we can choose $r_1 = 786{,}433$ and $r_2 = 5{,}767{,}169$, the NTT primes for ring dimension $N \leq 2^{17}$. Note, this sprout is not universal since $\log_2 r_1 \approx 19.584$ and $\log_2 r_2 \approx 22.459$. However, we can use this sprout similarly since the top sprout is close enough to 2^{61}. Each sprout r can be represented as $r = 2^{\alpha} \cdot r_1^{\beta_1} \cdot r_2^{\beta_2}$, where $0 \leq \alpha \leq 20$, $\beta_i \in \{0, 1\}$. We note that we can find a sprout that is located within* 0.5 *bits for any real-valued bits. Rational rescale can be done similarly as in the previous examples.*

The following theorem summarizes the rescalable conditions regarding the choice of sprouts. Informally, if the sprouts are universal, the rational rescale can be done by any arbitrary bit.

Theorem 3.2.1 (Universal Rescalability)**.** *Let the ciphertext modulus $q_i \in [2^{\omega}(1-\eta), 2^{\omega}(1+\eta)]$ for some $\eta > 0$ for $i \in [L-1]$. Let the maximum sprout modulus $r_{top} = r_0 \cdots r_s$. Assume for any positive integer $\gamma \leq w$, 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 $n = \lceil \delta / w \rceil$.*

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

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

[3]For larger dimensions, we can utilize incomplete NTTs instead of complete NTTs.

which concludes the proof. □

Note that the choice of sprout in Examples 2 and 3 are universal sprouts satisfying the assumption of Theorem 3.2.1 with $\epsilon < 2^{-13}$. Also, there are plenty of 61-bit primes in $[2^{\omega}(1-2^{-20}), 2^{\omega}(1+2^{-20})]$. Thus, these sprouts have *universal rescalability*, i.e., the ciphertexts are rational rescalable by $2^{\delta}\cdot(1\pm 2^{-12})$ for any $\delta \in \mathbb{N}$ smaller than the current ciphertext modulus, where the ciphertext moduli are grafted with the sprout.

More generally, universal sprouts can be used independent of the operations or the choice of parameters such as message precision or ring dimension. Given the unit moduli chosen following the assumptions of Theorem 3.2.1, the output grafted ciphertext moduli Q' with the universal sprouts in Algorithm 1 can be chosen as follows. The universality ensures that $|\log_2 r' - \log_2 \Delta|$ is sufficiently small.

Algorithm 4 Selecting grafted ciphertext moduli for multiplication.

```
Input: $Q = q_0 \cdots q_{\ell-1} \cdot r$ for input ciphertext with scaling factor $\Delta$ and word size $w = 2^{\omega}$.
Output: $Q' = q_0 \cdots q_{\ell'-1} \cdot r'$ for output ciphertext in Algorithm 1.
1: Choose $r' \mid r_{\text{top}}$ such that $|\log_2 r' - \log_2 \Delta|$ achieves the minimum.
2: $\ell' \leftarrow \lfloor(\omega\cdot\ell + \lfloor\log_2 r\rceil - \lfloor\log_2 r'\rceil)/\omega\rfloor$
3: return $Q' = q_0 \cdots q_{\ell'-1} \cdot r'$
```

As a final note, we can also utilize the sprout in Example 4, even though it is not universal, since it can roughly represent every integer or integer ± 0.5 bits.

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

As demonstrated in Example 2, 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 the later examples in Section 3.2.1, the more general universal sprouts also require some power-of-two factors to achieve their universality. In these cases, we can seamlessly leverage the advantages of the previous power-of-two-modulus CKKS scheme [CKKS17] in the RNS manner.

To do so, we need to consider the procedures involving the power-of-two moduli, specifically the procedures with a changing modulus: FBC, Inv-RS, ModUp, ModDown, and RS. They were originally performed only between the relatively prime bases. Thus, in the following, we extend the existing algorithms to ensure that grafted RNS-CKKS can be implemented in the RNS with the power-of-two factors. We start with the cases when the extensions are natural.

First, the FBC is defined and proved based on the Chinese Remainder Theorem (CRT) between the pair-wise relatively prime factors of the input modulus, and between that of P. That is, P and Q are not required to be relatively prime. More precsiely, if we factorize P and Q as in the previous sections, e.g., $Q = q_0 \cdots q_{\ell-1}\cdot 2^{\beta}$, when applying FBC on a polynomial $\mathbf{a} \in \mathcal{R}_Q$

computes $\sum_{j\in[\ell+1]}[(\mathbf{a} \mod q_j)\cdot \hat{q}_j^{-1}]_{q_j}\cdot \hat{q}_j$ modulo each CRT factor of P, where $\hat{q}_j = Q/q_j$ and $q_\ell := 2^\beta$.

Secondly, the other four modulus switching procedures were also defined and proved based on the CRT, but now, based on the pair-wise relatively prime factors of a product of the input and output modulus. If the power-of-two parts of the modulus do not change during the modulus switching, all the factors involved in computation remain pair-wise relatively prime and work the same as before. Therefore, we can naturally extend these operations in the following cases: Inv-RS, ModUp, ModDown, and RS (with an integer factor) between moduli Q and Q', where $\mathrm{ord}_2 Q = \mathrm{ord}_2 Q'$.

However, in the grafted RNS-CKKS, Inv-RS, ModDown, and RS are used between the moduli with different power-of-two factors. We note that ModUp is only applied after the Inv-RS, which controls the power-of-two part equal to that of the switching key modulus, filling up the gadgets, as illustrated in Algorithm 2. Since RS with an integer factor is a special case of ModDown, we focus on Inv-RS and ModDown. In Algorithms 5 and 6, we specified them using the above natural extensions to/from a polynomial in an intermediate modulus. We denote the non-power-of-two factors of Q as q_i for $i \in [\ell]$, which may include a part of sprouts.

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

```
Input: $\mathbf{a} \in \mathcal{R}_Q$ for $Q = \tilde{Q}\cdot 2^\beta$ and $R = \tilde{R}\cdot 2^\gamma$, where $\tilde{Q}$ and $\tilde{R}$ are odd integers and $\beta,\gamma \in \mathbb{Z}\geq 0$.
Output: $\mathsf{Inv\text{-}RS}_R(\mathbf{a}) \in \mathcal{R}_{QR}$.
1: $\mathbf{b} \leftarrow \begin{cases} [2^\gamma]_{q_i}\cdot[\mathbf{a}]_{q_i} & \pmod{q_i} \text{ for } i\in[\ell]\ , \\ 2^\gamma\cdot[\mathbf{a}]_{2^\beta} & \pmod{2^{\beta+\gamma}}\ , \end{cases}$      ▷ $\mathbf{b} \in \mathcal{R}_{\tilde{Q}\cdot 2^{\beta+\gamma}}$
2: $\mathbf{c} \leftarrow \mathsf{Inv\text{-}RS}_{\tilde{R}}(\mathbf{b})$      ▷ $\mathbf{c} \in \mathcal{R}_{QR}$
   return $\mathbf{c}$
```

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

Algorithm 6 ModDown with power-of-two sprouts.

```
Input: $\mathbf{a} \in \mathcal{R}_{QR}$ for $Q = \tilde{Q}\cdot 2^\beta$ and $R = \tilde{R}\cdot 2^\gamma$, where $\tilde{Q}$ and $\tilde{R}$ are odd integers and $\beta,\gamma \in \mathbb{Z}\geq 0$.
Output: $\mathsf{ModDown}_{QR\to Q}(\mathbf{a}) \in \mathcal{R}_Q$.
1: $\mathbf{b} \leftarrow \mathsf{ModDown}_{QR\to\tilde{Q}\cdot 2^{\beta+\gamma}}(\mathbf{a})$      ▷ $\mathbf{b} \in \mathcal{R}_{\tilde{Q}\cdot 2^{\beta+\gamma}}$
2: $\mathbf{c} \leftarrow \begin{cases} [2^{-\gamma}]_{q_i}\cdot\left([\mathbf{b}]_{q_i} - [\mathbf{b}]_{2^\gamma}\right) & \pmod{q_i} \text{ for } i\in[\ell]\ , \\ \left([\mathbf{b}]_{2^{\beta+\gamma}} - [\mathbf{b}]_{2^\gamma}\right)/2^\gamma & \pmod{2^\beta}\ , \end{cases}$      ▷ $\mathbf{c} \in \mathcal{R}_Q$
   return $\mathbf{c}$
```

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

Finally, we are able to follow the key switching in Algorithm 2 and the homomorphic multiplication in Algorithm 1 with the universal sprouts. We remark that the key switching procedure can be slightly modified to omit the $\mathsf{Inv\text{-}RS}$ by the power-of-two factors. That is, key switching in Algorithm 2 for the power-of-two modulus part can be performed modulo $2^{\mathrm{ord}_2(Q)}$ by reducing the modulus, rather than modulo $2^{\mathrm{ord}_2(Q_{\max})}$ with an additional $\mathsf{Inv\text{-}RS}$.

3.2.3 Efficient Sprout Arithmetic

In this subsection, we discuss hardware-level arithmetic strategies, mostly focusing on standard 64-bit processors. In other words, we figure out efficient instantiations of the methods described in the previous section. For simplicity, we stick to a single-threaded Central Processing Unit (CPU) case.

Basics for 64-bit processors

Given a (universal) sprout for standard 64-bit processors, a naive approach requires three 64-bit moduli to handle any polynomial multiplications via NTT. To check this, let $r < 2^{64}$ be a product of all moduli in the sprout. In order to deal with polynomial multiplication over modulo r, one may use *emulated NTT*: embed modulo r into a larger modulus $p > Nr^2$ so that NTT over modulo p emulates polynomial multiplication modulo r without modular reduction by p ever occurring.[4] In this case, as p should typically be larger than 2^{128} ($\because$ r is close to 2^{64}), we need three 64 bit NTT primes.

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

Example 5. *We follow the settings in Example 3 so that $r_{top} = 2^{15} \cdot r_1 \cdot r_2$ where r_1 is a 16-bit NTT prime and r_2 is a 30-bit NTT prime. The key observation is that the modulus $r_1 r_2 \simeq 2^{46}$ can be handled with a composite NTT. As both r_1 and r_2 are NTT primes, one can easily find $2N$-th primitive roots of unity modulo $r_1 r_2$. Then, we may construct one NTT modulus to be $r_1 r_2$ and the other one to be a larger modulus $g > N \cdot 2^{30}$ so that we can emulate modulo 2^{15} arithmetic in modulo g arithmetic. This results in using two NTT moduli instead of three, as desired.*

Note that the strategy described in Example 5 is compatible with RLWE switching keys: if one has only either r_1 or r_2 in the sprout modulus at some point, then this can be naturally embedded into the modulus $r_1 r_2$ without any costly transformations. Notably, we may properly define the embedding $\mathsf{Embed}_{r_1 \to r_1 r_2} : \mathcal{R}_{r_1} \to \mathcal{R}_{r_1 r_2}$ to be compatible with NTT. That is, we have

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

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

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

If the sprout modulus is sufficiently small, then we may even use one word-sized NTT for the sprout. In particular, the following types of sprouts from Example 3 support single-word arithmetic.

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

Other machine word sizes

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

Example 6. *The strategy for 2^{ω}-bit word size for $\omega \geq 7$ (i.e., 128-bit or larger size architectures) can easily be generalized from the 64-bit case. That is, we choose $r_{top} = 2^{15} \cdot r_1 \cdot r_2 \cdot \cdots \cdot r_{\log_2(\omega)-4}$ where r_i is a 2^{i+3}-bit prime for each $1 \leq i \leq \log_2(w) - 4$. Then, this sprout can be computed with two words, as in the standard 64-bit case, by embedding the power-of-two part to a larger prime and using composite NTT for the rest.*

Example 7. *We describe a strategy for a 32-bit word size. Let $r_{top} = 2^{15} \cdot r$ be a sprout where $r = 2^{16} + 1$ is an NTT prime. We embed 2^{15} to $g_1 g_2$ where g_1, g_2 are sufficiently large word-sized NTT primes such that $2^{30} \cdot N < g_1 g_2$. Then we may handle the sprout arithmetic with three words g_1, g_2, and r. If the power-of-two part 2^t become sufficiently small such that $2^{2t} \cdot N < g_1$, we can use two words.*

3.3 Applications

As Grafting breaks the linkage between the scaling factors and the RNS moduli, it can solve many restrictions that previous implementations had. Grafting is effective in both low- and high-precision scenarios, as the size of the scaling factor is often far from the multiples of the word size. We do not need to find small NTT primes (which sometimes do not exist) to accelerate such parameters, as well as the parameters for composite rescaling [AAB+23] or Tuple-CKKS multiplications [CCKS23]. Also, the parameters are not required to reserve the special RNS moduli for a fixed precision bootstrapping. Such characteristics are also hardware and compiler-friendly, as the computations are not tied to the choice of homomorphic operations for different circuits. Selecting FHE parameters has traditionally been an obstacle in implementing FHE due to its application-dependent nature. We expect that Grafting lowers the hurdle of HE parameter selection, leading to further optimization techniques in HE applications.

In this section, we consider some examples in such a manner that can be considered as Grafting-specific applications. We will go through some straightforward properties of Grafting. For instance, the naturally supporting low and high precision without changing the low-level parameters. It allows us to compute arbitrary circuits with arbitrary precision, independent of the choice of the ciphertext modulus. So, we do not need to reserve the special primes or modulus for optimized bootstrapping or special-purposed circuits. Hence, we showcase the implementation results[5] of bootstrapping and the homomorphic DFT with arbitrarily-sized output modulus among the possible ciphertext moduli. As a next step, we demonstrate the efficiency gains from Grafting when applied to applications that use small scaling factors for specific reasons. We demonstrate the applicability of Grafting to Tuple-CKKS multiplication [CCKS23] and its efficacy compared to a prior state.

3.3.1 Straightforward Properties and Gains

Low and High Precision Arithmetic

As the magnitude of the scaling factors can be freely chosen independent of the ciphertext modulus, we have the freedom to choose how precise the operations we would evaluate. There is no need to change the low-level parameters (except for bootstrappings, where the available modulus may be insufficiently large). What we can do is multiply and rescale by the desired amount. The only requirement for high-precision computation is to encode and decode the messages with large scaling factors. More interestingly, the high-precision and lower-precision computations can be done using the same parameters, possibly interchangeable during the computation. Also, if the modulus is sufficiently large for the high-precision bootstrapping, the homomorphic DFTs and $\mathsf{EvalMod}$ can be done with arbitrary precisions. The $\mathsf{ModRaise}$ procedure may become slightly more expensive since the modulus before $\mathsf{ModRaise}$ should be sufficiently large to deliver the message with high precisions. It may require iCRT on several moduli, then embed it into a larger modulus Q, applying modulus q_i operation on the iCRT result for all i.

On the other hand, the low precision computation can benefit from Grafting for its efficiency.

[5]In fact, there is nothing to do in terms of implementation except changing the modulus of the input ciphertext.

The low-precision computation can be done using the parameters for higher-precision parameters or requiring more numbers of moduli since each is smaller than usual (sometimes lacking the NTT primes), but in an inefficient manner. It can be accelerated by Grafting, just rescaling by a smaller amount, not requiring preparation of the special-purpose low-precision parameters.

Flexible Output Modulus

Bootstrapping is generally designed to maximize the multiplicative budget for the upcoming computations. However, the maximized budget may not be required depending on the circuit it evaluates. Especially for circuits with a shallow multiplicative depth but involving wide parallel computations, it becomes rather a burden as the high-modulus computations are more expensive. For instance, assume we are computing a neural network layer consisting of large matrix multiplication followed by a polynomial evaluation, approximating an activation function. Depending on the size of the matrix, it is often more efficient to evaluate the matrix multiplication using the smallest possible modulus, then refresh the modulus to evaluate the rest of the circuit, i.e., activation function, rather than keeping a larger modulus in evaluating the matrix multiplication to skip a single bootstrap. In such cases, the modulus of the ciphertext before a matrix multiplication can be set for a few depths of multiplications rather than the largest possible modulus for better efficiency.

However, this was not so feasible in the original bootstrappable CKKS parameters. Since the precision of the bootstrapping sub-procedures must be larger than that of other multiplications for the same output precision, the scaling factors were chosen differently to optimize the implementation. This restricted accelerating the bootstrapping procedure by lowering down the modulus to arbitrarily desired modulus. In the Grafting scenario, however, this constraint no longer applies, allowing a flexible choice of the output modulus, with a corresponding speed-up proportional to the modulus size.

3.3.2 Tuple-CKKS, Revisited

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

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

Compatibility

We check the compatibility of our method with the multiplication of [CCKS23]. For simplicity, we stick to the pair multiplication - the general tuple multiplication should be checked almost the same. Let us recall the definitions of the components of the pair multiplication in [CCKS23, Definition 4.1, 4.3, 4.5]. In the definitions, $\otimes$ denotes the CKKS tensor operation, Relin denotes the CKKS relinearization, RS_q denotes the rescaling by q, DCP and RCB denote the decomposition of CKKS ciphertext into quotient and remainder and their recombination as defined in [CCKS23, Definition 3.3].

Definition 3.3.1 (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 3.3.2 (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 3.3.3 (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*

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

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

When applying the concept of Grafting to the double multiplication framework, we may perform all the operations except $\mathsf{Relin}(q_{\mathsf{div}} \cdot \hat{\mathsf{ct}})$ 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 [CCKS23, Lemma 4.4].

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

Definition 3.3.4 (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 in* $R^2_{Q^{(\ell-1)}} \times R^2_{Q^{(\ell-1)}}$ *as*

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

We also give a generalized version of [CCKS23, Lemma 4.6] in Lemma 3.3.1, showing the correctness of the pair multiplication after changing the pair rescale definition.

Lemma 3.3.1. *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* $\mathsf{sk} = (1, \mathbf{s}) \in R^2$ *be a secret key with* $\mathbf{s}$ *of Hamming weight* h*. Then the following quantity has an infinity norm of* $\leq (h+1)/2$.

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

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

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

Now, to complete the proof, note that

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

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

The Theorem 3.3.2 is an analog of [CCKS23, Theorem 4.8], guaranteeing the correctness of pair multiplication. The correctness proof follows from the Lemma 3.3.1. We define Mult^2 as a composition of tensor, relinearize, and rescale.

Theorem 3.3.2. *Let* $\mathsf{CT} = (\hat{\mathsf{ct}}_1, \check{\mathsf{ct}}_1), \mathsf{CT}_2 = (\hat{\mathsf{ct}}_2, \check{\mathsf{ct}}_2) \in R^2_{Q^{(\ell)}} \times R^2_{Q^{(\ell)}}$ *be ciphertext pairs. Let* $\alpha_\ell/\beta_\ell = Q^{(\ell)}/Q^{(\ell-1)}$ *where* $(\alpha_\ell, \beta_\ell) = 1$ *and* $\mathsf{sk} = (1, \mathbf{s}) \in R^2$ *be a secret key with* 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 the following quantity has an* *infinity norm of* $\leq (N\check{M}^2/q_{\mathsf{div}} + E_{\mathsf{Relin}} + h)\beta_\ell/\alpha_\ell + (h+1)/2$.

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

Efficiency

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

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

We note that the efficiency gain could be even more significant for lower precision or with fewer slots that require smaller division primes and rescaling factors. For instance, we may use $\log_2(q_{\mathsf{div}}) \leq 10$ and $\log_2\left(Q^{(\ell)}/Q^{(\ell-1)}\right) \leq 20$ for roughly less than 15-bit precision, resulting in an improvement factor of at least 3, compared to the naïve double-CKKS implementation.

3.4 Experimental Results

We implement Grafting and universal sprouts using C++ and create a non-grafted (simple) and grafted variant of RNS-CKKS. We report and compare execution times, ciphertext and key sizes, and the bootstrapping precision of our simple and grafted implementations of RNS-CKKS, and some applications.

In the following, a modulus denoted as $X \times Y$ implies that it can be decomposed into Y RNS moduli of size X bits each. We run our experiments on Ubuntu 24.04.1 LTS with an AMD Ryzen 7 3700X CPU and 66 GB of available memory. We disable CPU scaling, use only a single thread, and pin execution to one CPU core.

3.4.1 Outsourced Linear Transformation

As a warm-up, let us introduce the implementation result of the outsourced linear transformation technique explained in Section 3.1.1, particularly on the CtS step of CKKS bootstrapping. We target the FTa parameter of the HEaaN library [Cry22]. Table 3.1 illustrates the parameters for the original and the suggested parameters, especially focusing on the moduli chains. Instead of using primes with different sizes, we used primes of size $\approx 2^{60}$ to utilize the 64-bit word size fully. The ring dimension is $N = 2^{16}$, and the secret key Hamming weight is $h = 192$.

Table 3.1: Parameter for CtS implementation and the reported runing time.

	$\log q$					$\log p$	dnum	CtS (sec)
	Base	StC	Mult	EvalMod	CtS			
HEaaN.FTa	38	$32 + 28 \times 2$	28×5	38×8	41×3	42×2	10	2.44
Proposed	59×10				$61 + 62$	62	12	1.71

The measured running time of CtS in the FTa and the proposed parameters took 2.44 and 1.71 seconds, respectively. The acceleration ratio roughly meets the expectation, a product of the ratios between 1) the number of unit moduli and 2) the gadget rank dnum,

$$\frac{22}{13} \times \frac{10}{12} \simeq 1.41 \approx \frac{2.44 \text{ sec}}{1.71 \text{ sec}} \simeq 1.43 \ .$$

3.4.2 Grafted RNS-CKKS Implementation

We now present our Proof-of-Concept C++ implementation result of Grafting.

We base our library on the HEaaN library [Cry22]. For the simple and grafted variants, we use the same algorithms for polynomial arithmetic over native integers which allows a fair comparison of execution times. We test and benchmark our library using the FTa parameters from the HEaaN library and its grafted variant using the universal sprout from Example 3. We verify bootstrapping correctness for a precision of up to eight bits using a comprehensive test suite.

In Table 3.2, we display our adaption of the FTa parameters to the grafted case. Our strategy in adapting the parameter aims to keep the maximum ciphertext modulus $Q_{\max}$ and the switching key modulus $PQ_{\max}$ similar to the original parameters. However, this is not always successful, especially when the rank of the gadget decomposition is too large, restricting splitting $Q_{\max}$ into P-sized moduli; each is a product of word-sized unit moduli. This is also the case for our parameters. We change the gadget rank and slightly reduce the ciphertext modulus $Q_{\max}$.

In such a situation, we estimate the theoretical performance gain of each operation we will benchmark in order to make a fair comparison. The tensor products and rescale procedures (despite the additional costs of rational rescale instead of original rescale) are directly influenced by

Table 3.2: FTa parameters for the simple variant [Cry22] and our grafted adaptation.

$N = 2^{15}$	$\log q_i$					$\log p_i$	#	dnum
$\log PQ_{\max} = 777$	Base	StC	Mult	EvalMod	CtS			
simple (FTa)	38	$32 + 28 \times 2$	28×5	38×8	41×3	42×2	22	10
grafted	$61 \times 10 + 45$					61×2	13	6

the number of RNS moduli in the ciphertext modulus. Thus, the speedup factor is approximately the ratio between the number of RNS moduli, which is $\approx 20/11 \approx 1.82$, sorely by the Grafting.

The key switching and the relinearization procedures require NTT of $\mathcal{O}((d+2)(K+L+1))$, which is a significant part of their costs. Thus, the speedup factor can be roughly estimated as $\approx (10+2)/(6+2)\times(22/13) \approx 2.54$, with a factor of $22/13 \approx 1.69$ coming solely from Grafting.

For bootstrapping, assuming that the costs are 60% for CtS, 30% for EvalMod, and 10% for StC, we can approximate the performance gain as $\approx 0.6 \times 2.54 + 0.3 \times 1.82 \approx 2.07$ for the bootstrapping speedup. We note that each sub-procedure is assumed to be accelerated as the key switching and the tensor product, respectively, while StC computed in the smallest modulus is not accelerated much.

The ciphertext and key sizes can be similarly computed, expecting $1 - 12/20 = 40.0\%$ and $1 - (14/22) \times (6/10) \approx 61.82\%$ reduced sizes, respectively.[6]

Table 3.3: Execution times (in milliseconds) for tensoring, rescaling, relinearization, and bootstrapping, as well as memory sizes (in KiB) for ciphertexts and key switching keys. We also include rough estimates for the theoretical gains based on the parameters in Table 3.2. Note that we use the same code for polynomial arithmetic with native integer coefficients.

	Tensor	Rescale	Relin.	Bootstrap.	Ciphertext	Switching key
simple	9.86	28.15	260.03	13908.11	10240.1	112640.2
grafted	5.49	20.86	108.96	6794.56	6144.1	43008.2
Measured gain	1.80×	1.35×	2.39×	2.05×	−40.0 %	−61.8 %
Theoretical gain	1.82×	1.82×	2.54×	2.07×	−40.0 %	−61.8 %

In Table 3.3, we present our benchmark results for execution times and memory measurements. Both measured and theoretically predicted gains are given. The rough estimates do not capture all the details of our implementation, such as the memory layout with additional metadata, slightly more complex code for the grafted variant, or the specifics of our sprout design and its implications, though our measured gains meet our expectations. The result thus validates the efficacy of Grafting, even though its implementation is complicatedly fused with many other techniques. Overall, Grafting allows us to speed up bootstrapping significantly and reduce memory consumption significantly, as desired.

[6]We extend the above adaptation of the FTa parameter, as well as the theoretical estimations on its performance gain, to other RNS-CKKS parameters in the literature, which can be found in Section 3.4.4.

3.4.3 Straightforward Properties and Gains

Based on the discussions in Section 3.3.1, we implemented and experimented with some Grafting-specific applications. We introduce some implementation details and report the experimental results.

High-precision DFT

We implemented the high-precision encoding and decoding, utilizing the ‘__float128’, the quadruple-precision floating-point data type from `quadmath` C++ library. As the encoded values are embedded into the polynomials in the polynomial ring $\mathcal{R}_Q$, homomorphic computation does not require any floating point arithmetic of high precisions. When decrypting (or decoding) and ModRaise, the modulus should be larger than the desired scaling factor.

Table 3.4: Precision (in $\log_2$ of maximum error) and execution times (in milliseconds) of homomorphic DFT for various scaling factors. Error statics are also given. TODO: Make StC start at the lowest possible mod.

Homomorphic DFT	$\log_2 \Delta$	Prec.	Error Average	Error Std. Dev.	Error Max	Time (sec)
StC (Forward DFT)	20					
	40					
	60					
	80					
	100					
	120					
CtS (Backward DFT)	20					
	40					
	60					
	80					
	100					
	120					

Flexible Output Modulus.

We target the specific scenario where we want to evaluate a plaintext-ciphertext matrix multiplication, abbreviated as PCMM. It was demonstrated in [BCH+24] that evaluating the PCMM with the matrices in coefficient encoding can be done very efficiently, with one depth of available multiplications. We also note that matrix multiplication requires higher precision than usual because the messages and errors, as many as the dimensions of the matrices, are summed up.

Hence, we do not need to make the full modulus available after bootstrapping but raise it to the smallest modulus, roughly a sum of the base modulus (for latter bootstrapping), and a larger-than-usual scaling factor of approximately 74 bits. In such a setting, we report bench-marked speedups in Table 3.5. Some additional time is spent on EvalMod for a larger bootstrapping precision. We note that the speed-up is somewhat restricted since the maximum possible modulus is too small. We expect much larger speed-ups with larger ring dimensions.

Table 3.5: Execution times (in milliseconds) and precision for grafted bootstrapping with 'full' and 'flexible' output modulus and increased precision. In the latter case, we increase precision to allow a large matrix multiplication after bootstrapping. Note that the performance of EvalMod has deteriorated from this but was complemented by gains from modernizing up to a smaller modulus. TODO: One more starting mod, same consumption.

	Bootstrap.	StC	EvalMod	CtS	Precision
Full	6794.56	603.19	1587.81	2938.44	8.7
Flexible	6463.19	615.70	1576.68	2778.47	13.4
Gain	1.05×	0.98×	1.01×	1.06×	4.7

3.4.4 Expected Speed-up of Existing Parameters

In this subsection, we extend the performance estimation in Section 3.4.2 on the FTa parameter to other parameters. We investigate the parameters used in the RNS-CKKS libraries and suggest their Grafted version.

We categorize the default RNS-CKKS parameters in the libraries HEaaN [Cry22], SEAL [SEA23], Lattigo [EL23], and OpenFHE [ABBB^{+}22] into two groups: Somewhat Homomorphic Encryption (SHE) parameters and FHE parameter, where the SHE refers to an HE without bootstrapping.

In Table 3.6, we provide the SHE and FHE parameters in the literature. All sizes are given in logarithms base-two and #mod. denotes the number of NTT primes comprising the switching key modulus $PQ_{\max}$, denoted as PQ due to space limit. The size of the RNS moduli $\log q_i$ and $\log p_i$ are given with the number of moduli of that size. Moduli with fractional sizes are only partially used by the step they are allocated to, as referenced in [BMTH21]. For the FHE parameters, we additionally provide the size of the RNS moduli reserved for each bootstrapping sub-procedure or general homomorphic multiplications.

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

Table 3.6: HE parameters for CKKS scheme in the literature. Here, S and F denote the SHE and FHE parameters, respectively. The RNS moduli for SHE parameters are given for the base prime, the auxiliary modulus, and the rest. The RNS moduli for FHE parameters are given for the base prime, the ones reserved for bootstrapping, the auxiliary modulus, and the rest.

		$\log N$	$\log PQ$	$\log \Delta$	#mod.	$\log q_i$					$\log p_i$
						Base	StC	Mult	EM	CtS	
HEaaN [Cry22]	S	13	217	41	5	47	41×3				47
		14	436	42	10	50	42×7				46×2
		15	866	42	20	48	42×14				46×5
		15	860	40	21	50	40×19				50
	F	15	771	36	17	49	33×3	36×1	49×8	47×3	54×1
		16	1,555	42	30	58	42×3	42×9	58×9	58×3	59×3 $+60 \times 2$
		17	2,070	51	40	61	51×3	51×13	51×10	51×3	53×10
Lattigo [EL23]	S	14	438	34	12	45	34×9				43×2
		15	880	40	21	50	40×17				50×3
	F	15	768	25	16	50	60	$50 + 25$	50×8	49×2	50×2
		16	1,546	40	30	60	39×3	40×9	60×8	56×4	61×5
		16	1,553	30	27	55	60×1.5	60×7.5	55×8	53×4	61×5
OpenFHE [ABBB+22]	S	14	371	50	7	60	50×5				60
		15	675	90	7	105	90×5				119
	F	16	1,579	58	27	60	58×2	58×4	58×13	58×2	60×5
		17	2,910	78	34	89	78×3	78×8	78×13	78×3	119×6

Reduction in the Number of NTT Blocks

From the parameters in the literature, we designed their Grafted variants. Our strategy aims to keep the maximum ciphertext modulus $Q_{\max}$ and the switching key modulus $PQ_{\max}$ similar to the original parameters.

For example, the SHE parameter of 860-bit $PQ_{\max}$ comprise 21 NTT primes from HEaaN [Cry22] library, can be reduced to the product of 14 unit moduli of 61-62 bits:

$$\underbrace{50 + 40 \times 19}_{Q_{\max}} + \underbrace{50}_{P} \rightarrow \underbrace{61 \times 13}_{Q_{\max}} + \underbrace{62}_{P}.$$

When considering the changes in dnum, we roughly expect the speed-up of key-switching as the ratio of $(\mathsf{dnum} + 2)(L + K + 1)$ between the original and the new parameters, which is $\approx$ 105% faster. The speed-up for the addition and tensor product can roughly be the ratio of $(L + 1)$, resulting in $\approx$ 43% speed-up. The resulting ciphertext modulus $Q_{\max}$ slightly decreases from 810 to 793 bits. We consider the universal sprout as two NTT primes during the efficiency comparisons.

The 2,910-bit FHE parameter for high-precision from OpenFHE [ABBB+22], as another example, uses 78-bit scaling factor as default, resulting in one 83-bit base modulus, twenty-seven 78-bit moduli, and six 119-bit moduli, in total of 34 primes. This requires larger arithmetic than the machine word size, e.g., 128-bit, introducing some inefficiency. We can decompose the modulus into 48 number of $\approx$ 61-bit moduli, or 50 number of $\approx$ 59-bit moduli by utilizing the 64-bit machine. When using the 128-bit arithmetic, for efficiency comparison, we can choose the 24 number of $\approx$ 121-bit moduli. For the last case, we can achieve a 90% faster key switching and 47% faster addition and tensor products, with slightly reduced $Q_{\max}$, from 2,196 to 2,178 bits. Also, we expect much higher speed-ups when using 64-bit arithmetic.

In Table 3.7, we provide the re-designed parameters corresponding to the parameters in Table 3.6, based on the policy described in Section 3.4.4. Some of the parameters are not expected to be accelerated well, especially when Δ is close to the machine's word size or when dnum is set not so compatible with the mostly word-sized moduli. In such cases, we may benefit from a slightly larger modulus, with secret keys possibly less sparse.

Table 3.7: Grafted parameters corresponding to the HE parameters in the literature. Here, S and F denote the SHE and FHE parameters, respectively. The number of RNS moduli shows the changes from the original to the Grafted parameters. The scaling factor Δ is unnecessary for the Grafted parameters; however, we give that of the corresponding parameters as a reference. The maximum ciphertext modulus $Q_{\max}$ is set approximately the same as the original parameters for fair comparisons. The expected speed-up ratios for addition and tensor products (Add & T.) and key-switching (KS) are given in the last columns. The ratio is especially small when Δ is close to the machine's word size or when $\log_2 PQ$ is too small to split well into word-sized moduli. An asterisk ($*$) indicates that the machine word size (or the unit arithmetic) is set to be 128 bits, expecting much larger speed-ups when using 64-bit with Grafting.

		$\log N$	$\log PQ$	$\log \Delta$	#mod.	$\log q_i$	$\log p_i$	Speed-ups	
								Add & T.	KS
HEaaN [Cry22]	S	13	217	41	$5 \to 4$	54×3	55	1.00	1.20
		14	437	42	$10 \to 8$	$54 \times 3 + 55 \times 3$	55×2	1.14	1.33
		15	860	42	$20 \to 16$	$53 \times 4 + 54 \times 8$	54×4	1.15	1.18
		15	855	40	$21 \to 14$	61×13	62	1.43	2.05
	F	15	773	36	$17 \to 13$	$59 \times 8 + 60 \times 4$	61	1.23	1.56
		16	1,545	42	$30 \to 25$	$61 \times 5 + 62 \times 15$	62×5	1.19	1.35
		17	2,070	51	$40 \to 36$	$57 \times 15 + 58 \times 12$	$57 \times 3 + 58 \times 6$	1.07	1.08
Lattigo [EL23]	S	14	428	34	$12 \to 7$	61×6	62	1.43	1.31
		15	886	40	$21 \to 15$	59×12	$59 \times 2 + 60$	1.38	1.75
	F	15	768	25	$16 \to 13$	59×11	$59 + 60$	1.17	1.29
		16	1,546	40	$30 \to 25$	$61 \times 4 + 62 \times 16$	62×5	1.19	1.35
		16	1,553	30	$27 \to 25$	$61 \times 4 + 62 \times 16$	62×5	1.00	1.21
OpenFHE [ABBB+22]	S	14	371	50	$7 \to 6$	$61 + 62 \times 4$	62	1.00	1.14
		15	675	90	$7 \to 6$	$111 \times 4 + 112$	119×2	1.00*	1.14*
	F	16	1,558	58	$27 \to 26$	62×20	53×6	1.05	1.17
		17	2,910	78	$34 \to 24$	121×18	122×6	1.47*	1.90*

Chapter 4

Homomorphic Polynomial Approximation with Adaptive Precisions

4.1 Homomorphic Evaluation using Adaptive Precisions

One of the key features of Grafting arithmetic is its support for homomorphic computations with arbitrary scaling factors. Grafting enables not only the use of scaling factors that exceed the size of the machine word or are too small to have enough NTT primes, but also the ability to change scaling factors during the computation.

In this section, we present a new approach to homomorphic computation with real/complex-valued numbers with reduced modulus consumptions using adaptive precision. That is, the scaling factors can change during the computation, or different scaling factors can be used in parallel.

For instance, assume we have to compute $C(\mathbf{x}, \mathbf{y}) = (\mathbf{A}\mathbf{x}).\mathbf{y} + \mathbf{x}^3$ for $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N$, where $\mathbf{A}$ is a matrix and the cubing and multiplication (.) are defined component-wise. Matrix-vector multiplications generally introduce larger numerical errors since the errors are linearly combined. Therefore, we assume the matrix-vector multiplication requires at least 30 bits of precision while others only require 10. In such cases, we empirically use $\Delta \approx 2^{45}$ for 30-bit multiplication precision, consuming $135 (= 45 \times 3)$ bits of ciphertext modulus for evaluating the circuit C. However, if the scaling factors for $\mathbf{A}\mathbf{x}$ and $\mathbf{x}^3$ can differ, we can consume $90 = \max(45 \times 2, 25 \times 3)$ bits of ciphertext modulus (assuming $\Delta \approx 2^{25}$ gives 10-bit multiplication precision). We can use different scaling factors for each sub-circuit and adjust the results before the addition without wasting the modulus or re-designing the RNS moduli chain.

Another possible scenario is to increase the precision during computation adaptively; for instance, iterative methods like Newton's method usually need small precisions at the beginning, and it increases in the latter iterations. Polynomial approximations on non-polynomial functions give coefficients descending with the degree. Thus, the latter basis (e.g., higher degree monomials in power basis) requires fewer precisions for computing the basis. In such cases, Grafting enables using different scaling factors and reduces the modulus consumption.

In the following, we will focus on homomorphically evaluating a polynomial with adaptive precisions. We first recap some known polynomial approximation methods that are used for approximating the functions for homomorphic evaluations. Based on the properties of the approximation methods, we introduce a homomorphic polynomial evaluation technique that uses adaptive scaling factors, possibly in multi-tracks.

4.1.1 Homomorphic Polynomial Evaluation with Multiple Scaling Factors

We first recap the Baby-Step-Giant-Step (Lazy-BSGS) algorithm using the Chebyshev polynomials and its *lazy* variant from [LLK+22], which operates relinearization before multiplications (if needed).

Algorithm 7 BSGS Algorithm.

```
Input: $\mathsf{ct}_1 = \mathsf{Enc}(\mathbf{x})$ and $\{c_j\}_{j \le d} \in \mathbb{R}^{d+1}$ for $p(x) = \sum_{j \le d} c_j T_j(x)$.
Output: A ciphertext encrypting $p(\mathbf{x})$.
    Let $b$ be the number of baby steps and $g$ be the number of giant steps, satisfying $2^b \cdot 2^g > d$,
    where for given $b$, $g$ is the smallest possible integer.
 1: procedure SETUP($\mathsf{ct}_1, b, g$)
 2:     for $i = 2; i < 2^b; i{+}{+}$ do                                  ▷ Generating $T_2, T_3, \cdots, T_{2^b-1}$
 3:         $\mathsf{ct}_i \leftarrow 2 \cdot \mathsf{ct}_{i_0} \cdot \mathsf{ct}_{i_1} - \mathsf{ct}_{i_0 - i_1}$ where $i_0 = 2^{\lfloor \log_2 i \rfloor}, i_1 = i - i_0$
 4:     end for
 5:     for $i = 2^b; i < d; i \leftarrow 2i$ do                           ▷ Generating $T_{2^b}, T_{2^{b+1}}, \cdots, T_{2^{b+g-1}}$
 6:         $\mathsf{ct}_i \leftarrow 2 \cdot \mathsf{ct}_{i/2}^2 - 1$
 7:     end for
 8: end procedure

 9: procedure BABYSTEP($p, b$)
10:     return $\sum_{j=0}^{2^b-1} c_j \mathsf{ct}_j$
11: end procedure

12: procedure GIANTSTEP($p, b, g$)
13:     if $\deg(p) < 2^b$ then
14:         return BABYSTEP($p, b$)
15:     end if
16:     $\mathsf{ct}_q \leftarrow$ GIANTSTEP($q, b, g - 1$)          ▷ Let $q, r$ be polynomials s.t. $p = q \cdot T_{2^{b+g-1}} + r$.
17:     $\mathsf{ct}_r \leftarrow$ GIANTSTEP($r, b, g - 1$)
18:     return $\mathsf{ct}_q \cdot \mathsf{ct}_{2^{b+g-1}} + \mathsf{ct}_r$
19: end procedure
```

For the *lazy* variant, multiplications are split into tensor products and others (relinearization and rescaling), and the relinearizations are delayed as much as possible. That is, the ciphertexts are relinearized only before multiplications, if needed.

There is some remaining room for further optimizing the modulus consumption. For instance, each baby step evaluation takes $\sum_j^{2^b-1} c_j \mathsf{ct}_j$, where ct_j are precomputed in the SetUp procedure. For instance, the ct_{2^b-1} can be generated with a computation of depth b, resulting in a total computation of depth $d+1$ for the baby step. Instead of doing so, one can compute $c_{2^b-1}\mathsf{ct}_{2^b-1}$ within depth d due to a recurrence relation: $c \cdot T_{2^i-1} = 2 \cdot T_{2^{i-1}} \cdot (c \cdot T_{2^{i-1}-1}) - c \cdot T_1$ for $i \geq 1$. As the constant multiplication by $c = c_{2^i-1}\mathbb{R}$ is done only with T_1, $c_{2^b-1}\mathsf{ct}_{2^b-1}$ can be computed using multiplicative depths of d. This modified baby step computation, BabyStepOpt, requires more computational overhead; however, only a few BabySteps should be replaced by BabyStepOpt. The giant step results in the form of

$$((bs_0 + bs_1 \cdot T_{2^b}) + (bs_2 + bs_3 \cdot T_{2^b})T_{2^{b+1}}) + ((bs_4 + bs_5 \cdot T_{2^b}) + (bs_6 + bs_7 \cdot T_{2^b})T_{2^{b+1}})T_{2^{b+2}},$$

where bs_i are the baby step evaluations. We only require bs_7 to be computed via BabyStepOpt instead of BabyStep since it is the only baby step result that is consecutively multiplied with depth-b, depth-$(b+1)$, and depth-$(b+2)$.

As a final remark for optimization, During SetUp (line 3), it is better to use $i_0 = 2^{\lfloor \log_2 i \rfloor} - 1$ for reducing the error variance amplification, as demonstrated in [LLK+22].

BSGS Algorithm with Two Scaling Factors

In the case using two different scaling factors, $\Delta_1 > \Delta_2$, let assume the components of degree 0 to $(2^{b+g-1} - 1)$ are computed using Δ_1 and the components of degree 2^{b+g-1} to d are computed using Δ_2. The two sub-giant steps (for q and r) can be done with different scaling factors in the giant step and adjusted before addition. It requires repeating the SetUp twice for each scaling factor, which introduces non-negligible overhead. So, having the number of baby steps not too large is beneficial. Omitting the odd (or even) components for odd (or even) functions is desirable, as introduced in [LLK+22].

We give the concrete algorithms in Algorithm 8, where BabyStep may be replaced by BabyStepOpt for the last baby steps, i.e., for degrees $2^{b+g-1}-1$ and $2^{b+g}-1$ (if exists). It consumes $\max((b+g-1)\log_2 \Delta_1, (b+g)\log_2 \Delta_2 + (\log_2 \Delta_1 - \log_2 \Delta_2))$ bits of ciphertext modulus.

BSGS Algorithm with Multiple Scaling Factors

It can be extended to use multiple scaling factors, and we may reduce the modulus consumption further. When considering three different scaling factors, $\Delta_1 > \Delta_2 > \Delta_3$, we can consider three different approaches to reduce the modulus consumption: Among 2^{b+g} components of degree 0 to $2^{b+g} - 1$,

1) evaluate the first quarter with Δ_1 and the second quarter with Δ_2, and the last half with Δ_3,

Algorithm 8 BSGS Algorithm with Two Scaling Factors.

```
Input: $\mathsf{ct}_1 = \mathsf{Enc}(\mathbf{x}; \Delta_1)$ and $\{c_j\}_{j \le d} \in \mathbb{R}^{d+1}$ for $p(x) = \sum_{j \le d} c_j T_j(x)$ and $\Delta_1 > \Delta_2$.
Output: A ciphertext encrypting $p(\mathbf{x}; \Delta_1)$.
    Let $b$ be the number of baby steps and $g$ be the number of giant steps, satisfying $2^b \cdot 2^g > d$,
    where for given $b$, $g$ is the smallest possible integer.
 1: procedure SetUp($\mathsf{ct}_1, b, g, \Delta_\alpha$)
 2:     if $\mathsf{ct}_1$.scale_factor $= \Delta_1$ && $\alpha = 2$ then
 3:         $\mathsf{ct}_1 \leftarrow \mathsf{RS}_{\Delta_1/\Delta_2}(\mathsf{ct}_1)$
 4:     end if
 5:     for $i = 2; i < 2^b; i$++ do                        ▷ Generating $T_2, T_3, \cdots, T_{2^b-1}$ w.r.t. $\Delta_\alpha$
 6:         $\mathsf{ct}_i \leftarrow 2 \cdot \mathsf{ct}_{i_0} \cdot \mathsf{ct}_{i_1} - \mathsf{ct}_{i_0 - i_1}$ where $i_0 = 2^{\lfloor \log_2 i \rfloor}, i_1 = i - i_0$
 7:     end for
 8:     for $i = 2^b; i < d; i \leftarrow 2i$ do              ▷ Generating $T_{2^b}, T_{2^{b+1}}, \cdots, T_{2^{b+g-1}}$ w.r.t. $\Delta_\alpha$.
 9:         if $\alpha = 1$ && $i = 2^{b+g-1}$ then                      ▷ Omitting $T_{2^{b+g-1}}$ if $\alpha = 1$.
10:             break
11:         end if
12:         $\mathsf{ct}_i \leftarrow 2 \cdot \mathsf{ct}_{i/2}^2 - 1$
13:     end for
14: end procedure

15: procedure BabyStep($p, b, \Delta_\alpha$)
16:     return $\sum_{j=0}^{2^b-1} c_j \mathsf{ct}_j$                                          ▷ $\mathsf{ct}_j$ w.r.t. $\Delta_\alpha$.
17: end procedure

18: procedure GiantStep($p, b, g, \Delta_1, \Delta_2$)
19:     if $\deg(p) < 2^b$ && $\Delta_1 = \Delta_2$ then
20:         return BabyStep($p, b, \Delta_1$)
21:     end if                               ▷ Let $q, r$ be polynomials s.t. $p = q \cdot T_{2^{b+g-1}} + r$.
22:     $\mathsf{ct}_q \leftarrow$ GiantStep($q, b, g-1, \Delta_2, \Delta_2$)
23:     $\mathsf{ct}_r \leftarrow$ GiantStep($r, b, g-1, \Delta_1, \Delta_1$)
24:     return $\mathsf{ct}_q \cdot \mathsf{ct}_{2^{b+g-1}} + \mathsf{ct}_r$   ▷ Modulus and scaling factor (to $\Delta_1$) adjustment is required,
25: end procedure                          ▷ since $\mathsf{ct}_q, \mathsf{ct}_{2^{b+g-1}}$ w.r.t. $\Delta_2$ and $\mathsf{ct}_r$ w.r.t. $\Delta_1$.
```

2) evaluate the first quarter with Δ_1 and the middle half with Δ_2, and the last quarter with Δ_3,

3) evaluate the first half with Δ_1 and the third quarter with Δ_2, and the last quarter with Δ_3.

In the first case, the modulus consumption used for the first half can be decreased. Focusing on the first half, the first option offers a better modulus consumption because it uses two scaling factors instead of one (Δ_1) for evaluating the first half components. As the Algorithm 8 splits the modulus consumption tracks for the first half and the second half, we can use Δ_1 for depth-$(b+g-2)$, Δ_2 for depth-$(b+g-1)$, and Δ_3 for depth-$(b+g)$. Thus, we have the modulus consumption of

$$\max_{1 \le i \le 3} \left((b+g-3+i) \log_2 \Delta_i + (\log_2 \Delta_1 - \log_2 \Delta_i) \right),$$

where $(\log_2 \Delta_1 - \log_2 \Delta_i)$ is for adjusting before adding them.

In the second and the third cases, the scaling factor Δ_2 is used for depth-$(b+g)$ computation, i.e., the third quarter of the components. Thus, the impact on the modulus consumption is restricted, which become

$$\max\left((b+g-2)\log_2\Delta_1, (b+g)\log_2\Delta_2 + (\log_2\Delta_1 - \log_2\Delta_2)\right),$$

or

$$\max\left((b+g-1)\log_2\Delta_1, (b+g)\log_2\Delta_2 + (\log_2\Delta_1 - \log_2\Delta_2)\right),$$

respectively for the second and the third cases.

When considering more scaling factors, e.g., $\Delta_1 > \Delta_2 > \cdots > \Delta_k$, we can generalize the modulus bit consumption to $\max_{1\le i\le k}\left((b+g-k+i)\log_2\Delta_i + (\log_2\Delta_1 - \log_2\Delta_i)\right)$, using Δ_i for evaluating the components of degrees less than or equal to $2^{b+g-k+i}-1$.

4.1.2 Error Analysis on Polynomial Approximation with Multiple Scaling Factors

In the previous section, we provide the BSGS algorithm with multiple scaling factors. Here, we assume a more general polynomial basis and analyze the error amplification during the homomorphic evaluations.

For a polynomial basis $\mathcal{B} = \{\varphi_i(x)\}$ of $\mathbb{R}[x]$, let $p(x)$ be a degree-d polynomial approximating a real function $f(x)$ in a range $I \subset \mathbb{R}$, where the approximation error is defined as $e_{\text{aprx}} := f(x) - p(x)$. Let $p(x) = \sum_{j=0}^{d} c_j\varphi_j(x)$ for some coefficients $c_j \in \mathbb{R}$ $(0 \le j \le d)$ with respect to the basis $\mathcal{B}$. When homomorphically evaluating the polynomial on an encrypted message x with a scaling factor Δ, one can compute each basis $\varphi_j(x)$ with the scaling factor Δ (or a scaling factor close to it).

However, one can also use smaller Δ when computing the bases and reduce the modulus consumption with a cost of larger errors (thus smaller precision). We found that, however, if the coefficient c_j is especially smaller than other coefficients, the quantity $c_j\varphi_j(x)$ can be computed with a smaller scaling factor, as the increased error from $\varphi(x)$ is multiplied with the smaller c_j, making the impact of the larger error negligible. Precisely, let φ_j (omitting (x) for simplicity) that is computed with a scaling factor Δ as

$$\varphi_j^{\Delta} := \varphi_j + e_{\text{basis},j}^{\Delta}.$$

Note that the dominant term of the basis error is the rescale error, which has a variance of $n(h+1)/(12\Delta^2)$, where h is the secret key Hamming weight, and n is the number of slots.[1] The

[1] Assuming the rescaling error (in coefficient) uniform in $[-1/2, 1/2]$, when multiplied with the secret key, it is

error term of the homomorphic evaluation of f on x consists of errors from different sources, the polynomial approximation error e_{aprx} and the error included in each basis, amplified when multiplied with c_j, resulting in a final error of

$$e^{\Delta} := e_{\text{aprx}} + \sum_j c_j e^{\Delta}_{\text{basis},\, j}.$$

For the maximum integer $k > 0$ satisfying $2^k < d$, let assume $|c_j|$ for $j \geq 2^k$ are much smaller than the average of $|c_j|$ for $0 \leq j < 2^k$, i.e., $|c_j| < \delta \cdot \text{Avg}(\{|c_i|\}_{0 \leq i < 2^k})$ for some $\delta < 1$. In such cases, we can use smaller scaling factor $\Delta' \approx \sqrt{\delta}\Delta$ for $\varphi_j^{\Delta'}$ ($j \geq 2^k$) with an error of

$$e^{\Delta,\Delta'} := e_{\text{aprx}} + \sum_{0 \leq i < 2^k} c_i e^{\Delta}_{\text{basis},i} + \sum_{2^k \leq j \leq d} c_j e^{\Delta'}_{\text{basis},\, j}.$$

The ratio between the variances of the two error terms can roughly be bounded as

$$\begin{aligned} \frac{\text{Var}[e^{\Delta,\Delta'}]}{\text{Var}[e^{\Delta}]} &\approx \frac{\text{Var}[e_{\text{aprx}}] + \sum_{i<2^k} c_i^2 \cdot \frac{n(h+1)}{12\Delta^2} + \sum_{j \geq 2^k} c_j^2 \cdot \frac{n(h+1)}{12\Delta'^2}}{\text{Var}[e_{\text{aprx}}] + \sum_j c_j^2 \cdot \frac{n(h+1)}{12\Delta^2}} \\ &< \frac{\sum_{i<2^k} c_i^2 + \sum_{j \geq 2^k} c_j^2 \cdot \delta^{-1}}{\sum_{i<2^k} c_i^2 + \sum_{j \geq 2^k} c_j^2} \\ &< \frac{1+\delta}{1+\delta^2} \approx 1 + \delta, \end{aligned}$$

since $\sum_{j \geq 2^k} c_j^2 \leq \delta \cdot ((d - 2^k + 1)/2^k) \cdot \sum_{i<2^k} c_i^2 \leq \sum_{i<2^k} c_i^2 \cdot \delta$. Thus, using a larger scaling factor for φ_j ($j \geq 2^k$) has a limited impact on the magnitude of the error, while it reduces the modulus consumption.

This is similar to the analysis of Lee et al. [LLK+22], which finds the approximating polynomial that minimizes the error variance and can be naturally extended. Possibly with multiple scaling factors, Δ_j for φ_j, we have the error term after evaluating the polynomial,

$$e := e_{\text{aprx}} + \sum_j c_j e^{\Delta_j}_{\text{basis},\, j},$$

where the variance is given as

$$\text{Var}[e] := \text{Var}[e_{\text{aprx}}] + \sum_j c_j^2 \cdot \frac{n(h+1)}{12\Delta_j^2},$$

assuming $\text{Var}[e^{\Delta_j}_{\text{basis},\, j}] = n(h+1)/(12\Delta_j^2)$. We want to find the coefficient vector $\mathbf{c} = (c_j)_j$ that

a sum of $(h+1)$ uniform variables. When decoded, the error is amplified by the number of slots and scaled with the scaling factor.

minimize the variance, which is a solution of

$$\nabla_{\mathbf{c}}\left(\mathrm{Var}[e_{\mathrm{aprx}}] + \sum_j c_j^2 \cdot \frac{n(h+1)}{12\Delta_j^2}\right) = 0. \tag{4.1.1}$$

Similar to that of [LLK+22], we may obtain a linear system depending on the target function f and e_{aprx}.

Before moving to the next section to deal with a concrete example, we demonstrate that other polynomial approximations can also use smaller scaling factors for the higher degrees to reduce the modulus consumption. For this, we require $\sup_{j \geq i} |c_j|$ to be decreased, and we can find a finite sequence of positive integers $0 = i_0 < i_1 < i_2 < \cdots < i_k \leq d$ that satisfies

$$\sup_{j \geq i_{\ell+1}} |c_j| < \sup_{j \geq i_\ell} |c_j| \cdot \delta,$$

for some $\delta \ll 1$, $0 \leq \ell \leq k-1$, and $k > 0$.

For instance, in Taylor approximation, for real/complex analytic function $f(x)$, the Taylor expansion at 0 is given as $T(f)(x) = \sum_{j=0}^{\infty}(f^{(j)}(0)/j!)x^j$ and can be decomposed into a d-th Taylor polynomial $T_d(f)(x) = \sum_{j=0}^{d}(f^{(j)}(0)/j!)x^j$ and a remainder $R_d(f)(x) = f(x) - T_d(f)(x)$. The error when evaluating a polynomial $T_d(f)$ instead of f can be efficiently bounded when the input x is centered at 0 (or shifted point), as $|R_d(f)(x)| \leq M \cdot r^{d+1}/(d+1)!$ when $|x| < r$ and $|f^{(d+1)}(x)| < M$ for some $M, r > 0$. When $f^{(j)}(0)$ is bounded from above by K, which is the case for the functions generally required to be evaluated in homomorphic computations, the j-th coefficient $c_j = f^{(j)}(0)/j!$ can be bounded as

$$|c_j| < \frac{K}{j!} \leq \frac{K \cdot e^{j-1}}{j^j} \quad .$$

Thus, depending on the function to approximate, the modulus consumption may be reduced by using smaller scaling factors for the higher-degree basis. Note, however, that it is challenging to analyze elaborately for general functions.

4.2 Application to EvalMod

In this section, we focus on EvalMod, the sub-procedure of CKKS bootstrapping that homomorphically evaluates a modulo function on the message.

4.2.1 Remez Approximation for Multiple-Scaling Factor Evaluation

In general, the CKKS bootstrapping is used to lengthen the multiplicative budget for further computations as much as possible. During bootstrapping, we homomorphically decode the ciphertext, raise the modulus, encode the ciphertext again, and apply a reduction modulo the base modulus, referred to as StC, ModRaise, CtS, and EvalMod. Basically, for given an input ciphertext in $\mathcal{R}_q^2$ encrypting a message $\mathbf{m}$, it generates a ciphertext in $\mathcal{R}_{Q_{\max}}^2$ encrypting a message $\mathbf{m} + q\mathbf{I}$, then evaluate a polynomial approximation of mod q.

One popular way of (approximately) applying the modulo q operation. As it is a periodic function, we set a high-probability interval for the message $\mathbf{m} + \mathbf{e}' + q\mathbf{I}$, then approximate the function to a polynomial. Each coefficient of the message is close to a multiple of q. When scaled, the modulo q operation can be approximated to $\frac{q}{2\pi} \cdot \sin(\frac{2\pi x}{q}) \approx x \bmod q$, when x is close to a multiple of q. The state-of-the-art implementation uses Remez mini-max approximation using the Chebyshev basis in the multi-interval consisting of short intervals centered at the multiples of q. They approximate the sine or cosine functions with a scaled interval, then apply the double angle formula and inverse sine function [HK20, BMTH21, BTH22], as

$$x \bmod 1 \approx \frac{1}{2\pi} \cdot \sin(2\pi x) = \frac{1}{2\pi} \cdot \cos\left(2^k \cdot \left(\frac{2\pi}{2^k} \cdot \left(x - \frac{1}{4}\right)\right)\right),$$

or arcsin of it, where $x \in \cup_{-K+1 \le i \le K-1}[i - \epsilon, e + \epsilon]$ for some integer $K > 0$, real number $0 < \epsilon \ll 1$ and the double angle formula can be k times repeatedly applied to $\cos(\frac{2\pi}{2^k} \cdot (x - \frac{1}{4}))$.

For instance, the Remez algorithm provides a mini-max approximation of the cosine function

$$x \mapsto \cos\left(\frac{2\pi}{2^k} \cdot \left(x - \frac{1}{4}\right)\right),$$

over the intervals $\cup_{-K+1 \le i \le K-1}[i - \epsilon, e + \epsilon]$ for a suitable K. It is determined by the target degree (or the approximation precision), the range parameter K, and k the number of double angle formulas to apply. Assume we have a 63-degree polynomial with coefficients for the Chebyshev basis. We can easily observe that the magnitude of the coefficients gets smaller for the higher degree basis, so we can consider sparing modulus by using less precise scaling for the coefficients as described in Section 4.1, Algorithm 8.

As an example, we can evaluate the polynomial with a circuit that is separated into two subordinate circuits using different moduli chains: 1) compute the first half of the polynomial, i.e., for degrees 0 to 31, consuming 5 multiplicative depths with a desired precision, and 2) compute the second half, e.g., for degrees 32 to 63 with smaller coefficients consuming 6 multiplicative depths with lesser precision. As the second half requires more multiplicative depths, we can reduce the total modulus consumed during the polynomial approximation, e.g., 5 depths with $\Delta \approx 42$ bits, and 6 depths with $\Delta \approx 37$ bits in parallel, consuming the modulus of total $\approx$ max(5×42, 6×37) = 222 bits, instead of 6×42 = 252 bits, saving 30 bits of modulus. The cost of evaluating the cosine function increases slightly since the Chebyshev bases are computed in both subordinate circuits (SetUp w.r.t. $\alpha = 1, 2$ in Algorithm 8).

We illustrate the magnitude of the coefficients of the cosine approximation polynomial in Figure 4.1. The magnitude of the coefficients indeed decreases for the increasing degree. However, as the magnitude of some coefficients in the second half is not sufficiently smaller than that of the first half, the gain in the modulus consumption is limited.

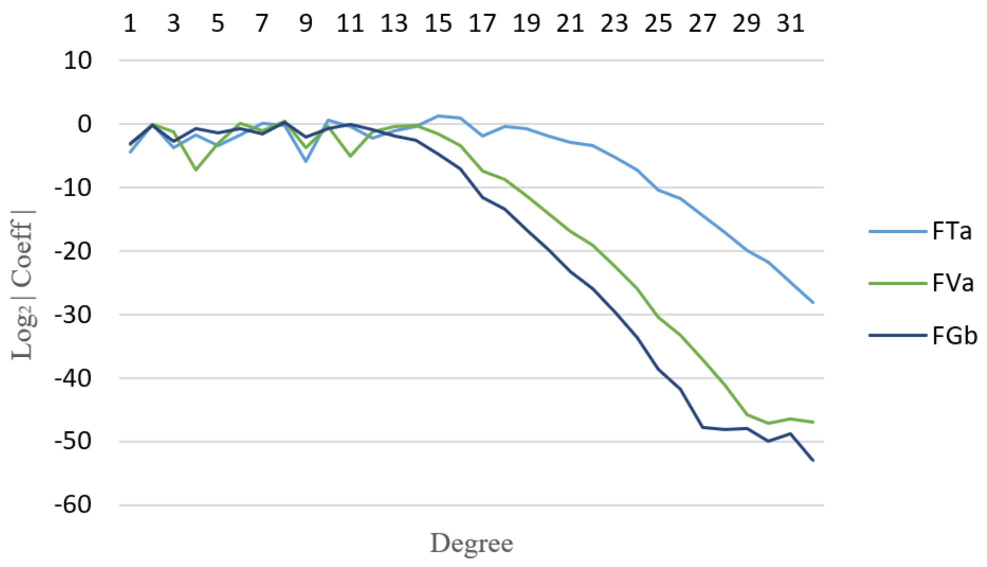

Figure 4.1: Coefficient sizes (in $\log_2$) of the cosine approximation polynomials of degree 63, odd coefficients only, from [Cry22].

We note, however, it does not guarantee the precision and which scaling factors to use, nor the optimality of the approximation for a target precision. Indeed, the Remez algorithm is not friendly to these kinds of analyses; rather, we will use the error variance minimization technique introduced in [LLK+22] to find the optimal approximation of the modulo function for a better EvalMod procedure. In the following, we provide the concrete approximation methods that provide polynomial approximations favorable to the method introduced in Section 4.1, further reducing the modulus consumption.

4.2.2 Error Variance Minimization for Multiple Scaling Factors

The modulo 1 function $\mathsf{Mod1} : \cup_{-K+1 \le i \le K-1} [i - \epsilon, e + \epsilon] \to [-\epsilon, \epsilon]$ can be represented as $\mathsf{Mod1}(x) = x - \lfloor x \rceil$, for some $\epsilon > 0$, and is an odd function. By following the analysis of [LLK+22], we obtain the linear system of equations from Equation 4.1.1 that for the odd coefficients $\mathbf{c} = [c_1, c_3, \cdots, c_d]^t$ as

$$(\mathbf{T} + \mathbf{D}\mathbf{I}) \cdot \mathbf{c} = \mathbf{y},$$

where

$$\mathbf{T} = \begin{bmatrix} \langle T_1, T_1 \rangle & \langle T_1, T_3 \rangle & \cdots & \langle T_1, T_d \rangle \\ \langle T_3, T_1 \rangle & \langle T_3, T_3 \rangle & \cdots & \langle T_3, T_d \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle T_d, T_1 \rangle & \langle T_d, T_3 \rangle & \cdots & \langle T_n, T_d \rangle \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} \langle \mathsf{Mod1}, T_1 \rangle \\ \langle \mathsf{Mod1}, T_3 \rangle \\ \vdots \\ \langle \mathsf{Mod1}, T_d \rangle \end{bmatrix},$$

and $\mathbf{D} = n(h+1)/12 \cdot \mathrm{diag}(\Delta_1^{-2}, \Delta_3^{-2}, \cdots, \Delta_d^{-2})$.

To use the BSGS algorithm with two scaling factors, described in Algorithm 8, we should find the coefficients $\mathbf{c}^*$ multiplied at each baby step. Assume that $d = 2^b \cdot 2^g - 1$, following the notations in Algorithm 8, the coefficients $\mathbf{c}$ and $\mathbf{c}^*$ are in a linear relation that $\mathbf{c} = \mathbf{L} \cdot \mathbf{c}^*$ for

$$\mathbf{L} = \begin{bmatrix} \mathbf{A}_{2^{b+g-1}} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{A}_{2^{b+g-2}} & 0 \\ 0 & \mathbf{A}_{2^{b+g-2}} \end{bmatrix} \cdots \begin{bmatrix} \mathbf{A}_{2^b} & & \\ & \ddots & \\ & & \mathbf{A}_{2^b} \end{bmatrix} \cdot \mathbf{c}^*, \quad \text{where } \mathbf{A}_k = \begin{bmatrix} \mathbf{I}_{k/2} & \frac{1}{2}\mathbf{J}_{k/2} \\ 0 & \frac{1}{2}\mathbf{I}_{k/2} \end{bmatrix}.$$

The diagonal matrix $\mathbf{D}$ becomes

$$\mathbf{D} = \frac{n(h+1)}{12} \cdot \begin{bmatrix} \Delta_1^{-2} \cdot \mathbf{I}_{2^{b+g-2}} & 0 \\ 0 & \Delta_2^{-2} \cdot \mathbf{I}_{2^{b+g-2}} \end{bmatrix}.$$

The resulting linear system would be

$$(\mathbf{T} + \mathbf{D}\mathbf{I}) \cdot \mathbf{L} \cdot \mathbf{c}^* = \mathbf{y}.$$

For a typical parameter set of $\log_2 q_0 = 45$, $h = 128$, and $n = 2^{15}$, $K = 14$, we have a default scaling factor of $\log_2 \Delta_1 = 45$. We can introduce smaller scaling factors and find the polynomial approximations of Mod1 for multi-scaling factor evaluation. As shown in Figure 4.2, the size of the coefficients are in some ranges not to amplify the errors from the basis when multiplied. Note that if Δ_2 is smaller than some threshold, the coefficients corresponding to the scaling factor get too small to be encoded using Δ_2, which is undesirable.

As a remark, it can be easily figured out that Grafting is friendly to the skip-connection type operations [SSKM24]. Also, the EvalMod approximation obtained from the error variance minimization can be applied to their techniques, reducing modulus consumption further.

4.3 Implementation and Experimental Results

4.3.1 Cosine Approximations

We first focus on evaluating the cosine function. As the scaling factors and multiplication precisions can be flexibly changed, we evaluate the Chebyshev approximation of the cosine

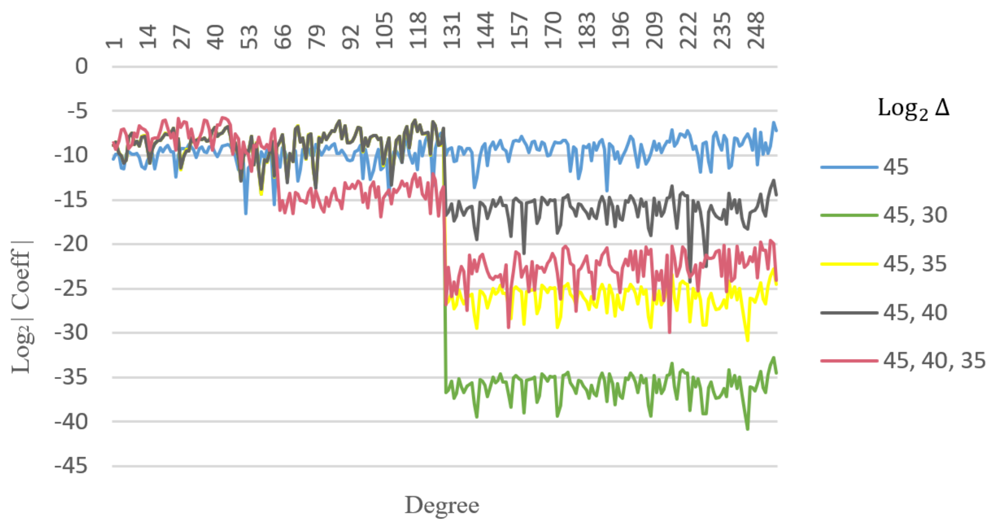

Figure 4.2: Coefficient sizes (in $\log_2$) of the EvalMod approximation polynomials of degree 511, odd coefficients only, and with multiple scaling factors.

function with higher precision than the one used for EvalMod in the previous subsection. We use the grafted FTa parameter, but a different 63-degree cosine function approximation, which is used for the FGb parameter bootstrapping [Cry22]. We use 42-bit Δ instead of 28-bit or 38-bit Δ. We report the precision and modulus consumption in Table 4.1.

Table 4.1: Precision (in $\log_2$ of maximum error) and modulus consumption (in bits) for grafted cosine function evaluation using the approximation polynomial from FGb parameter of [Cry22]. We used $\log_2 \Delta_1 = 42$ and $\log_2 \Delta_2 = 41$ to 37 following Algorithm 8. Statistics for the maximum errors (among the slots) from each execution are also shown.

Cosine approx.	Prec.	Error			Scaling factors		Modulus
		Average	Std. Dev.	Max	$\log_2 \Delta_1$	$\log_2 \Delta_2$	consum.
Original	21.4	2.44e-7	3.10e-8	3.65e-7	42	-	252
Reduced modulus consumption	21.6	2.45e-7	2.69e-8	3.13e-7	42	41	247
	21.5	2.42e-7	2.60e-8	3.33e-7	42	40	242
	21.6	2.45e-7	2.48e-8	3.15e-7	42	39	237
	19.3	1.44e-6	3.08e-8	1.53e-6	42	38	232
	13.2	1.10e-4	3.37e-8	1.10e-4	42	37	227

The modulus consumption can be reduced by 15 bits without damaging the final precision, which shows the basic levels of gain from using multiple scaling factors but, at the same time, shows the need for specific polynomial approximation for the multiple-scaling factors.

4.3.2 EvalMod from Error Variance Minimization

We obtain several approximation polynomials following the EvalMod function approximation described in Section 4.2. We show the implementation results in Table 4.2.

Table 4.2: Precision (in $\log_2$ of maximum error) and modulus consumption (in bits) for grafted EvalMod function evaluation. We use $\log_2 \Delta_1 = 42$ as default and search for the approximation polynomials by changing Δ_2. The evaluation follows Algorithm 8. Statistics for the maximum errors (among the slots) from each execution are also shown. TODO: Add execution times?

Cosine approx.	Prec.	Error			Scaling factors		Modulus
		Average	Std. Dev.	Max	$\log_2 \Delta_1$	$\log_2 \Delta_2$	consum.

Chapter 5

Summary and Future Works

In this work, we introduce a new modulus management system for RNS-CKKS called Grafting. We also introduce universal sprouts which, in combination with Grafting, significantly advance the current state-of-the-art for RNS-CKKS: We efficiently decouple the scaling factors from the ciphertext modulus. Leveraging the new arithmetic introduced by Grafting, we propose adaptive precision computations on encrypted data, utilizing multiple scaling factors and minimizing modulus consumption.

Initially, this approach allows us to separate parameters from specific use cases, facilitating generic parameter selection in libraries and enhancing usability for individuals without specialized knowledge. Secondly, it permits the selection of mostly word-sized primes for the ciphertext modulus, maintaining both precision and modulus space. This adjustment accelerates homomorphic multiplication and bootstrapping by factors of 2.20 and 2.05, respectively, for leading parameters. Furthermore, it notably reduces memory consumption for ciphertexts and key-switching keys. Lastly, this advancement can enhance existing CKKS applications, like bootstrapping, by enabling adaptive precision and versatile output moduli, as well as applications such as Tuple-CKKS.

Overall, our research improves on the two main challenges HE faces today: performance and usability. We foresee that this will encourage further research, particularly in HE applications, and we are keen on seeing these developments.

We conclude by suggesting potential future enhancements related to Grafting, such as: 1) the parallelization of Grafting for optimized implementations on AVX or GPU, incorporating power-of-two segments or possibly introducing new sprouts, and 2) the development of applications utilizing the new arithmetic for extremely low precision through smaller scaling factors or by dynamically adjusting scaling factors.

Bibliography

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

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

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

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

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

[BEHZ16] Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. A full RNS variant of FV like somewhat homomorphic encryption schemes. In Roberto

Avanzi and Howard M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, pages 423–442. Springer, Cham, August 2016.

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

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

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

[CCS19] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 34–54. Springer, Cham, May 2019.

[CHK$^+$18] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 360–384. Springer, Cham, April / May 2018.

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

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

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

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

[EL23] Tune Insight SA EPFL-LDS. Lattigo v5. Online: https://github.com/tuneinsight/lattigo, November 2023. EPFL-LDS, Tune Insight SA.

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

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

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

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

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

[KPK+22] Seonghak Kim, Minji Park, Jaehyung Kim, Taekyung Kim, and Chohong Min. EvalRound algorithm in CKKS bootstrapping. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 161–187. Springer, 2022.

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

[LLK+22] Yongwoo Lee, Joon-Woo Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and HyungChul Kang. High-precision bootstrapping for approximate homomorphic encryption by error variance minimization. In Orr Dunkelman and Stefan

Dziembowski, editors, *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 551–580. Springer, Cham, May / June 2022.

[LLL+21] Joon-Woo Lee, Eunsang Lee, Yongwoo Lee, Young-Sik Kim, and Jong-Seon No. High-precision bootstrapping of RNS-CKKS homomorphic encryption using optimal minimax polynomial approximation and inverse sine function. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 618–647. Springer, Cham, October 2021.

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

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

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

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

한국어 초록

RNS (Residue Number System) 기반의 CKKS (Cheon-Kim-Kim-Song) 동형암호 스킴 (SAC 2018)은 계산 속도 면에서 매우 효율적이기 때문에 널리 구현되어 왔습니다. 그러나 최신 구현들은 머신 워드 크기를 충분히 활용하지 못해 비효율을 초래하고 있습니다. 이는 동형 곱셈 과정에서의 재스케일 계수 (rescale factor)가 스케일링 계수 (scaling factor)와 같은 크기로 선택되기 때문으로, 이 크기가 머신 워드 크기와 가까이 맞춰지지 않으면 계산 자원이 낭비됩니다.

이 문제를 해결하기 위해, 우리는 *Grafting*이라는 새로운 모듈러스 관리 시스템을 제안합니다. 이는 암호문의 모듈러스를 워드 크기의 계수로 채우는 동시에 임의의 재스케일 (rescale)을 허용합니다. Grafting은 계산 속도를 높이고 메모리 사용량을 줄이며, 목표 정밀도와 상관없이 사용할 수 있는 보편적인 파라미터를 제공합니다. Grafting의 핵심 요소는 *sprout*으로, 이는 암호문의 모듈러스를 구성하는 일부분으로서 반복적으로 사용될 수 있습니다. 일반적으로 임의의 암호문 모듈러스를 사용할 경우, 다양한 모듈러스를 갖는 여러 개의 연산 키 (evaluation key)가 필요하지만, sprout를 활용하면 동일한 키로 다양한 재스케일 계수를 이용한 재스케일을 지원할 수 있습니다. 또한, 이는 *universal sprout*으로 확장되어 임의의 크기로 재스케일을 허용합니다. Grafting은 재스케일 계수의 크기, 스케일링 계수, RNS 계수 간의 의존성을 없애 RNS-CKKS 구현의 제약을 해결합니다.

또한 우리는 Grafting을 적용한 새로운 산술 기법을 바탕으로, 동형 계산의 실행 시간과 소비되는 모듈러스 관점에서의 효율성을 재검토하였습니다. Grafting 산술의 간단한 응용으로, 특히 작은 스케일링 계수를 요구하는 Tuple-CKKS 곱셈 (CCS 2023)과 같은 기술에서 계산 효율성을 2배 이상 개선할 수 있음을 확인했습니다. 또한, Grafting은 계산 중 스케일링 계수를 적응적으로 변경할 수 있으며, 하위 레벨의 파라미터인 모듈러스들을 수정하지 않고도 임의의 정밀도를 자연스럽게 지원합니다. 또, 동형 연산을 수행 시, 각 하위 절차에 대해 최적화된 스케일링 계수를 활용하여 계수 소비를 줄일 수 있습니다. 우리는 Baby-Step Giant-Step (BSGS) 다항식 계산 방식을 다중 스케일링 계수를 활용하는 경우에서 일반화하고, Error Variance Minimization 기법 (Eurocrypt 2022)을 기반으로 새롭게 제시한 다항식 연산 방법에 특화된 다항식 근사를 제안하였습니다.

우리는 HEaaN 라이브러리에서 Grafting이 적용된 파라미터를 기준으로 Grafting이 적용된 RNS-CKKS 코드와 그렇지 않은 코드를 C++로 구현하여 비교하는 실험을 진행했습니다. 벤치마크 결과는 Grafting의 효율성을 입증하였으며, 동형 곱셈에서는 2.20×, 부트스트래핑에서는 2.05×의 속도 향상을 달성했습니다. 암호문과 연산 키의 크기도 비슷한 비율로 감소했으며, 정밀도는 동일하게 유지되었습니다. 또한, Grafting 구현을 바탕으로 다중 스케일링 계수를 사용하는 동형 다항식 근사 연산을 EvalMod 연산에 적용하여, 실제로 모듈러스 소비를 줄이면서도 유사한 실행 시간을 유지하는 것을 확인하였습니다.

주 요 어: 암호학, RLWE (Ring Learning With Errors), 동형암호, 나머지수 체계 (RNS), 다항식 근사, 적응형 정밀도 연산.

학　번: 2019-24352

Acknowledgement

Acknowledgement goes here. Hangul in Acknowledgement may use “\kr”: Hong Gildong (홍길동).

TODO: Acknowledgements

ChatGPT says “Both acknowledgment and acknowledgement are correct spellings. The only difference is that acknowledgment is the preferred American English spelling, while acknowledgement is the preferred British English spelling. You can use either one based on your preference or the style guide you’re following.”