ModHE: Modular Homomorphic Encryption Using Module Lattices

Potentials and Limitations

Anisha Mukherjee[1], Aikata Aikata[1], Ahmet Can Mert[1], Yongwoo Lee[2], Sunmin Kwon[2], Maxim Deryabin[2] and Sujoy Sinha Roy[1]

[1]Graz University of Technology, Graz, Austria,
[2]Samsung Advanced Institute of Technology, Suwon, Republic of Korea

Abstract.
The promising field of homomorphic encryption enables functions to be evaluated on encrypted data and produce results that mimic the same computations done on plaintexts. It, therefore, comes as no surprise that many ventures at constructing homomorphic encryption schemes have come into the limelight in recent years. Most popular are those that rely on the hard lattice problem, called the Ring Learning with Errors problem (RLWE). One major limitation of these homomorphic encryption schemes is that in order to securely increase the maximum multiplicative depth, they need to increase the polynomial-size thereby also increasing the complexity of the design. We aim to bridge this gap by proposing a homomorphic encryption (HE) scheme based on the Module Learning with Errors problem (MLWE), ModHE that allows us to break the big computations into smaller ones. Given the popularity of module lattice-based post-quantum schemes, it is an evidently interesting research endeavor to also formulate module lattice-based homomorphic encryption schemes. While our proposed scheme is general, as a case study, we port the well-known RLWE-based CKKS scheme to the MLWE setting. The module version of the scheme completely stops the polynomial-size blowups when aiming for a greater circuit depth. Additionally, it presents greater opportunities for designing flexible, reusable, and parallelizable hardware architecture. A hardware implementation is provided to support our claims. We also acknowledge that as we try to decrease the complexity of computations, the amount of computations (such as relinearizations) increases. We hope that the potential and limitations of using such a hardware-friendly scheme will spark further research.

Keywords: Homomorphic encryption, module lattice, hardware reusability

1 Introduction

The digital world is quite asymmetric: devices like cell phones are compact but computationally challenged, which is why we often need access to large off-site ‘cloud’ servers with greater computing power. However, trust is a major issue with such outsourced computations. For example, a hospital wants statistical analysis on the percentage of patients with certain dominating illnesses and hence wants to send its patients’ medical records to a research facility. Can the hospital be assured of the complete privacy and protection of its data? One solution is that the hospital signs a privacy contract, and relies on the honesty of the research facility. A better solution would be that the hospital encrypts all its data beforehand and not rely on third parties for its own privacy. However for it to be meaningful the research facility must be able to analyze the encrypted data

without any need for decryption and the hospital would still obtain the same analytical results as the plain data would provide them. This is the very essence of Homomorphic Encryption (HE).

In 1978, Rivest, Adleman, and Dertouzos [RAD+78] conjectured that computations could be performed effectively on homomorphically encrypted data without compromising its security. For the next three decades however, the world only knew of partially homomorphic schemes (like RSA [RSA78] and ElGamal [ElG85]) that could support certain fixed types of operations on the ciphertext. The breakthrough came in 2009 when Gentry [Gen09] introduced the first Fully Homomorphic Encryption (FHE) scheme that could perform arbitrary operations on homomorphically encrypted data. He showed that a somewhat homomorphic scheme could be made fully homomorphic with a process he named 'bootstrapping' through which ciphertexts can be 'refreshed', a way to homomorphically evaluate the decryption circuit. His construction, although quite complex, sparked active research and led to the introduction of other simpler homomorphic schemes. [BV11a] proposed an FHE scheme whose security assumption was based on the classical hardness of solving standard lattice problems in the worst-case, which is more well-known as the Learning with Errors (LWE) problem. The dimension of the lattice determines the scheme's extent of security. Then, in 2011, [BV11b] and in 2012 [FV12] (BFV) ported the scheme in [BV11a] from standard LWE setting to LWE over algebraic rings with the Ring Learning with Errors problem (RLWE). The BGV scheme [BGV11] is another popular RLWE-HE candidate. The TFHE scheme [CGGI20] also uses LWE and the RLWE over a torus but adopts a different bootstrapping procedure. CKKS/HEAAN [CKKS17] and its residue number system (RNS) variant, [CHK+18b], are recent RLWE-based schemes that incorporate homomorphic computation of data over the real and complex fields. In spite of a series of significant advancements in the theoretical aspects of homomorphic encryption, present-day homomorphic encryption schemes introduce a huge computation overhead ranging from 10^4 to 10^5 compared to plaintext calculations. As a consequence, software implementations of homomorphic encryption in general-purpose computers are far from usable in the privacy-preserving outsourcing of computation.

To speed up homomorphic encryption significantly, multiple attempts to develop customized hardware accelerators have surfaced in the last few years. These works range from real acceleration works, for example, the GPU and FPGA-based accelerators [JKA+21, BHM+20, MAK+23, AdCY+23, RLPD20, RJV+18, TRG+20, TRV20], to futuristic ASIC designs [FSK+21, GVBP+22, KKK+22, SFK+22, KLK+22]. It becomes clear from their impressive speedup records, that ASIC or FPGA-based hardware accelerators will be fundamental to making homomorphic encryption usable for real-life privacy-preserving computation. Studying the above-mentioned hardware acceleration works, we see that they start with the mathematical representation of a given homomorphic encryption scheme and make hardware-based building blocks to speed up the mathematical steps of the scheme. In contrast to this typical hardware accelerator development cycle, our approach attempts to unify the two (often considered) sequential processes of scheme proposal followed by hardware design. We strive to accommodate the hardware angle and incorporate it during the process of scheme design.

The following two paragraphs present the motivation behind designing a 'hardware-friendly' homomorphic encryption scheme. In RLWE-based somewhat homomorphic encryption schemes e.g., BGV, BFV, CKKS, etc., the parameters (polynomial ring dimension and modulus size) are chosen depending on application complexity – the more complex the application the larger the parameters. The security of the scheme goes hand-in-hand with the size of the ciphertext modulus and the dimension of the polynomial ring: a need for higher multiplicative depth equates to choosing a bigger ciphertext modulus but to keep the level of security intact, a proportional increase in the polynomial ring dimension is equally imperative. As an example, while polynomials of degree 2^{12} and a

110-bit ciphertext modulus would suffice for homomorphic evaluation of a simple quadratic function, an application performing logistic regression would require almost 4 times larger ring dimension and ciphertext modulus sizes. In a software implementation, adjusting parameter sets for different applications is simple. However, in a hardware implementation, the circuits are physically 'hard-wired' and hence the large variations in the polynomial degree of RLWE-based homomorphic encryption schemes make it rather challenging to optimize the circuits for variable parameter sets. Hardware reusability is desired in the cloud as it will process different types of applications. That motivated us to answer the question, *Could homomorphic encryption schemes be designed in a way that they become hardware-friendly by construction?* The term "hardware-friendly" primarily refers to the ability to reuse hardware resources optimally.

In post-quantum public-key cryptography, module lattice-based schemes such as Saber [DKR+21], Kyber [SAB+21], Dilithium [BDK+21], etc., have been successful in mitigating similar problems with varying parameter sets. With a fixed polynomial degree, these schemes only have to change dimensions of the vector or the matrix of such polynomials to incorporate changing security levels, thereby opening up opportunities for flexibility and reusability in hardware. Taking inspiration from these facts, this paper gives the sketch of a module lattice-based homomorphic encryption scheme and discusses its advantages and limitations compared to the state-of-the-art ideal lattice-based schemes.

1.1 Our contributions

- In this work, we explore the prospects of using the module learning with errors (MLWE) problem in the context of designing a homomorphic encryption scheme. As a case study, we port the RLWE-based CKKS [CKKS17] scheme to the MLWE setting, and propose a new scheme which we call `Mod` or `ModRNS`. A proof-of-concept Sage implementation[1] of the module-lattice-based leveled CKKS homomorphic encryption scheme is provided.

- Algorithmically, we try to retain the properties of the ring variant while adapting them for modules, thereby realizing our motivation without having to make any drastic changes to the heuristics of the original scheme. We investigate the consequences of choosing a fixed base ring and then building upon the rank of the associated module depending on the desired parameters for circuit depth and security. We provide detailed algorithmic descriptions for the readers to understand the design decisions and challenges.

- We first highlight the advantages of better security assumptions associated with MLWE when compared with RLWE [AD17], [CDW17].

- Along with the assurance of stronger security, we also explain how MLWE is a good fit for reusing already existing hardware architecture even under varied parameter requirements. The availability of physical resources may not be at par with the changing security caliber. Indeed, with every such change, having to replace a machine's hardware or buying new ones would not be cheap. We show that in such a scenario, accommodating different parameter sets essentially translates to simply adjusting the rank of the module in MLWE, leading to hardware reusability.

- ModHE allows us to continue processing small polynomials even when we require a higher depth. This is because the number of polynomials packed in a module increases, but their size does not increase. We show how these different components can be utilized mostly in parallel and design the hardware accordingly. Our hardware

[1] https://github.com/anonymous-sub-ches/MODHE

design methodology utilizes the multi-dimensional parallel processing opportunities offered by ModHE and presents them in the context of `ModRNS`.

- As an additional interesting feature that comes with MLWE-based HE, we present the rank reduction technique. MLWE allows us to fix the degree of the ring so that corresponding to the required multiplicative depth, the dimension of the lattice problem (still ensuring security) can be adjusted only by changing the rank of the module. This offers us an opportunity to dynamically adjust/reduce the components of a ciphertext once a certain multiplicative depth has been consumed.
- We acknowledge the fact that ModHE has certain limitations in the form of constrained message packing, increased key sizes, reduced performance, and precision loss. While speed-wise MLWE cannot beat RLWE, it is a trade-off for achieving easier hardware reusability and stronger security.
- To support our motivation, we provide area and performance results for a hardware accelerator architecture targeting the Xilinx FPGA Alveo U280 card.

Organisation: In section 2, we provide mathematical details that will be useful to build concepts in the rest of the paper. We also give a brief description of the important sub-routines of the RNS-CKKS scheme. In section 3, we present algorithmic details of `Mod` version of CKKS, the error bounds of important sub-routines, and also discuss the RNS representation. In section 4, we give the potentials and limitations related to a module-based HE construction. The hardware design is proposed in section 5 and section 6 provides food for thought toward future possibilities and modifications of ModHE constructions. Section 7 concludes the paper.

2 Mathematical background

2.1 Notation

Let $N \in \mathbb{N}$ be a power of two. For a number field $\mathbb{Q}[X]/(\phi_{2N}(X))$ we denote $\mathcal{R} = \mathbb{Z}[X]/(\phi_{2N}(X))$ as its ring of integers consisting of polynomials modulo the $2N$-th cyclotomic polynomial, $\phi_{2N}(X) = X^N + 1$. Also, let $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$ be the residue ring of $\mathcal{R}$ modulo an integer q. An element of $\mathcal{R}_q$ is a polynomial of the form, $a(X) = \sum_{i=0}^{N-1} a_i X_i$ with each of its coefficients in $\mathbb{Z}_q$. The Euclidean norm on the coefficient vector (a_i) is denoted simply by $\|a\|$ while the l_∞ norm is $\|a\|_\infty$ such that $\|a\|_\infty = sup_i |a_i|$. The l_∞ norm of a polynomial a under the canonical embedding is denoted by $\|a\|_\infty^{can}$. We denote the module of rank r over $\mathcal{R}_q$ as $\mathcal{R}_q^r$.

Unless stated explicitly, we will use q to denote a ciphertext modulus. In the non-RNS version of our scheme we use $q_l = p^l \cdot q_0$, $0 \leq l \leq L$ for a base $p > 0$ and a modulus q_0 following [CKKS17]. When discussing the RNS version we will shift to a notation Q for the large ciphertext modulus which is a product of the small primes q_i.

Elements named in usual lowercase letters will denote single polynomials unless otherwise explicitly specified to be integers; bold lowercase letters will represent a vector of polynomials (except in section 2.2 where we use this notation for a vector of integers) and bold uppercase letters will denote multi-dimensional matrices. $\langle \cdot, \cdot \rangle$ denotes the inner product between two vectors. The Number Theoretic Transform (NTT) of a polynomial a is represented by $\tilde{a}$. We use $\circ$ to denote composition of two functions. We use $\star$ to represent dyadic multiplication and '$\cdot$' for the following types of multiplications: polynomial-polynomial, matrix-vector of polynomials, integer-polynomial or integer-integer multiplications (without ambiguity, the reference will be clear from the context it is being used in).

2.2 The Learning with Errors problem and its algebraic variants

The Learning with Errors (LWE) problem was introduced by Regev [Reg05] in 2005. The LWE problem is parameterized by two integers $n \geq 1$ and $q \geq 2$, and an error distribution χ_{err}. It has two variants: the search and the decision variant. The decision variant is usually preferred for defining cryptographic primitives.

Learning with Errors (LWE): Sample a secret vector $\mathbf{s}$ in some key distribution $\chi_{key}(\mathbb{Z}_q^n)$ and fix it. Consider i iterations of the following sampling process: at each i-th iteration, sample random vector $\mathbf{a}_i$ from a uniform distribution $\mathcal{U}(\mathbb{Z}_q^n)$ and error e_i from χ_{err}. Compute, $b_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i \in \mathbb{Z}_q$ and output the tuples, $(\mathbf{a}_i, b_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ called LWE samples from the distribution $\text{LWE}_{n,q,\chi_{err}}$. The 'search' LWE problem states that, given a polynomial number of samples from $\text{LWE}_{n,q,\chi_{err}}$, find $\mathbf{s}$. The 'decision' LWE asks to distinguish with non-negligible advantage between a polynomial number of samples drawn from $\text{LWE}_{n,q,\chi_{err}}$ and the same number of samples $(\mathbf{a_i'}, b_i')$ drawn uniformly from $\mathbb{Z}_q^n \times \mathbb{Z}_q$.

Ring Learning with Errors (RLWE): The Ring Learning With Errors (RLWE) [LPR10] was introduced for speeding up cryptographic constructions based on LWE. Let, $R_q = \mathbb{Z}_q[X]/\langle f(X) \rangle$ be a polynomial ring of degree N with the irreducible polynomial $f(X)$. Sample a secret polynomial s with coefficients from some key distribution $\chi_{key}(R)$ and fix it. Consider i iterations of the following sampling process: at each i-th iteration, sample public polynomial a_i from a uniform distribution $\mathcal{U}(R_q)$ and error e_i from χ_{err}. Compute, $b_i = a_i \cdot s + e_i \in R_q$ and output the tuples, $(a_i, b_i) \in R_q \times R_q$ called RLWE samples from the distribution $\text{RLWE}_{N,q,\chi_{err}}$. The 'search' LWE problem states that, given a polynomial number of samples from $\text{RLWE}_{N,q,\chi_{err}}$, find s. The 'decision' RLWE asks to distinguish with non-negligible advantage between a polynomial number of samples drawn from $\text{RLWE}_{N,q,\chi_{err}}$ and the same number of samples (a_i', b_i') drawn uniformly from $R_q \times R_q$.

While the LWE problem is known to be as hard as worst-case problems on Euclidean lattices, RLWE is considered as hard as the problems are restricted over special classes of ideal lattices. The Module Learning with Errors was introduced as a bridge between the general LWE and RLWE and was discussed in detail by Langlois and Stehlé [LS15].

Module Learning with Errors (MLWE): Consider the module $M \subseteq R^r$ over R with rank r, where R and R_q are defined as in RLWE above. Sample $\mathbf{s}$ from some key distribution $\chi_{key}(R^r)$ and fix it. Consider i iterations of the following sampling process: at each i-th iteration, sample public module element $\mathbf{a}_i$ from a uniform distribution $\mathcal{U}(R_q^r)$ and error e_i from χ_{err} over R. Compute, $b_i = \langle \mathbf{a_i}, \mathbf{s} \rangle + e_i \in R_q$ and output the tuples, $(\mathbf{a_i}, b_i) \in R_q^r \times R_q$ called MLWE samples from the distribution $\text{MLWE}_{N,r,q,\chi_{err}}$. The 'search' MLWE problem states that, given a polynomial number of samples from $\text{MLWE}_{N,r,q,\chi_{err}}$, find $\mathbf{s}$. The 'decision' MLWE asks to distinguish with non-negligible advantage between a polynomial number of samples drawn from $\text{MLWE}_{N,r,q,\chi_{err}}$ and the same number of samples $(\mathbf{a_i'}, b_i')$ drawn uniformly from $R_q^r \times R_q$.

For a fixed number of samples, the MLWE problem can also be viewed in terms of linear algebra by considering a matrix $\mathbf{A}$ whose rows consist of all the sampled $\mathbf{a_i}$'s. Interestingly, for a commutative ring R, an R-algebra is an R-module which is also a ring. The set, $M_n(R)$ of all square $n \times n$ matrices over a commutative ring $\mathcal{R}$ with entries in R is an R-algebra. While a multiplication between two module elements is not intrinsic, a multiplication between two elements of $M_n(R)$ will still make sense when perceived as elements in a ring. The security of the proposed MLWE-HE scheme follows from the hardness assumptions of the MLWE problem.

2.3 CKKS and RNS-CKKS

In a typical homomorphic encryption protocol, a client sends encrypted data to a cloud server that performs computations on it and sends the encrypted results back to the client. The client decrypts the received results locally to again obtain meaningful plaintext results. The client has his own secret key to be able to en(de)crypt messages.

We give an intuitive description of an RLWE-based homomorphic encryption scheme in the following part. Let a client's secret-key be $\mathtt{sk} = (1, s) \in \mathcal{R}_q^2$ and the corresponding public-key be $\mathtt{pk} = (b, a) \in \mathcal{R}_q^2$. Client encrypts a message m using $\mathtt{pk}$ and obtains the ciphertext $\mathtt{ct} \leftarrow (c_0 = v \cdot b + e_0 + m, c_1 = v \cdot a + e_1) \in \mathcal{R}_q^2$ where e_i is a Gaussian distributed error-polynomial and v is polynomial sampled from a distribution dictated by the scheme. The client can decrypt a valid ciphertext $\mathtt{ct} = (c_0, c_1)$ using her secret-key $\mathtt{sk}$ and recover the message $m \leftarrow \langle \mathtt{ct}, \mathtt{sk} \rangle$. Assume that a cloud contains two ciphertexts $\mathtt{ct} = (c_0, c_1)$ and $\mathtt{ct}' = (c_0', c_1') \in \mathcal{R}_q^2$ of the client with respect to messages m and m' respectively. The cloud can compute a valid encryption of $m + m'$ simply by adding the two ciphertexts as $\mathtt{ct}_{\text{add}} \leftarrow (c_0 + c_0', c_1 + c_1') \in \mathcal{R}_q^2$. Computing encryption of $m \cdot m'$ is relatively complex and often differs based on the scheme. The basic idea is that the multiplication of two ciphertexts will result in their respective components being multiplied with each other, like, $\mathtt{ct}_{\text{mult}} = (c_0 \cdot c_0', c_0 \cdot c_1' + c_1 \cdot c_0', c_1 \cdot c_1') \in \mathcal{R}_q^3$. This intermediate result has three polynomial components and can be decrypted using $(1, s, s^2)$ but not using $\mathtt{sk} = (1, s)$. To again allow for decryption to happen using $\mathtt{sk}$, a 'Key-Switching' operation is used to transform the three-component ciphertext $\mathtt{ct}_{\text{mult}}$ back into the usual two-component ciphertext $\mathtt{ct}_{\text{relin}}$ decryptable under $(1, s)$. In this context, key-switching is called 'relinearization' as it produces a linear ciphertext.

Most ideal lattice-based homomorphic encryption schemes such as BGV [BGV11], BFV [FV12], CKKS [CKKS17] and RNS-CKKS [CHK+18b] share a similar underlying protocol structure. We give a brief description of RNS-CKKS [CHK+18b] as we will build our MLWE construction upon it.

Residue Number System (RNS)
The Residue Number System makes use of the Chinese Remainder Theorem to represent an integer as a vector of its *residues* modulo a basis of pairwise co-prime integers. The same can also be applied to polynomials in rings. If a is a polynomial in the cyclotomic ring $\mathcal{R}_Q$ and $\mathcal{C} = \{q_0, \cdots, q_{k-1}\}$ is a basis such that $Q = \prod_{i=0}^{k-1} q_i$ then, there is a ring isomorphism from $a \in \mathcal{R}_Q$ to its representation $(a^{(0)}, a^{(1)}, \cdots, a^{(k-1)}) \in \prod_{i=0}^{k-1} \mathcal{R}_{q_i}$ being applied coefficient-wise. In compact form, the RNS representation of a can be denoted by $[a]_{\mathcal{C}}$. RNS proves useful for implementing HE schemes on hardware and software platforms as it enables the parallelization of computations. Each polynomial involved in the scheme's routines has shares with respect to each small moduli in the basis. In other words, polynomial arithmetic is done among smaller decomposed polynomials instead of big polynomials modulo a big modulus. HE schemes using RNS choose prime moduli q_i's so that they can utilise the advantage of Number Theoretic Transform (NTT) for fast multiplications.

RNS-CKKS
The original CKKS scheme could not support a double-CRT representation since the rounding operation involved in the approximate arithmetic dictated the use of a power-of-two ciphertext modulus. An RNS variant of CKKS appeared in [CHK+18b] by constructing algorithms that could incorporate double-CRT without compromising on the benefits of CKKS. We give a brief description of the important sub-routines of RNS-CKKS. For better legibility, we will use `RingRNS` to denote them. For a security parameter λ let q be the base integer modulus, L be the maximum level of computation, η be the bit precision. Ensuring $q_j/q \in (1 - 2^{-\eta}, 1 + 2^{-\eta})$, choose a prime p and the basis $\mathcal{C} = \{q_0, \cdots, q_L\}$ such

that at a certain level $0 \leq l \leq L$, $\mathcal{C}_l = \{q_0, \cdots, q_l\}$ and Q_l is a product of $q_i's$ up to l. In this section, we use the notation $y[i]$ to select the i-th element from the general L-tuple, y. To denote the selection of the j-th residue polynomial of $y[i]$ we use $y[i][j]$.

- $\texttt{RingRNS.KeyGen}(\texttt{s})$: The secret s is sampled from a secret key distribution χ_{key} (like the set $\mathcal{HWT}(h)$ of signed binary vectors $\{0, \pm 1\}^N$ with Hamming weight h) such that the secret key is $\texttt{sk} = (1, s)$. The public key is, $\texttt{pk} = (-a \cdot s + e, a) \in \mathcal{R}^2_{Q_L}$, and the evaluation key, where the polynomial a is sampled from a uniform distribution $\mathcal{U}$ over $\mathcal{R}_{Q_L}$ and the error e from an error distribution χ_{err}, usually chosen to be discrete Gaussian distribution $\mathcal{DG}(\sigma^2)$ with variance σ^2 and $\sigma > 0$.

- $\texttt{RingRNS.KSGen}(s, s')$: Given two secret polynomials s and s' from χ_{key}, sample the public polynomial a' from a uniform distribution $\mathcal{U}$ over $\mathcal{R}_{pQ_L}$ where p is a prime and error polynomial e' from error distribution χ_{err}. Output the key-switching key, $\texttt{swk}[i] = (swk_0[i], swk_1[i])_{0 \leq i \leq L} \in \mathcal{R}^2_{pQ_L}$ where, $(swk_0[i], swk_1[i]) = (-a'[i] \cdot s[i] + e'[i] + p \cdot B[i] \cdot s'[i], a'[i]) \pmod{pQ_L}$ such that $B[i] = 1 \pmod{q_i}$ and $B[i] = 0 \pmod{q_j}$ for all $j \neq i$. The key-switching key is used to transform a ciphertext decryptable under s into a ciphertext decryptable under s'.

  At first sight the subroutine $\texttt{RingRNS.KSGen}(\texttt{s}, \texttt{s}')$ looks very similar to the other subroutine $\texttt{RingRNS.KeyGen}(\texttt{s})$. However, there is one major difference: the generated switching keys are encryptions of some function $f(s) = s'$ of the secret s itself. This is referred to as 'Key Dependent Message Security' (KDM) or 'Circular Security'. It roughly means that a public-key scheme can be used to securely encrypt its own secret key (self-encryption). The exploration of KDM secure encryption schemes began with [BHHO08] where the authors proposed a decisional Diffie-Hellman based scheme that could securely encrypt linear combinations of the secret key. Later, a number of works [ACPS09, BGK11, BHHI10] showed that this can be extended for more complex functions involving the secret key as well. [MTY11] took a different approach by considering the secret key as an element in a ring. [MTY11] defines KDM security or circular security as follows: for a class of functions $\mathcal{F}$ a public-key encryption scheme is KDM[$\mathcal{F}$] secure if it is secure even against an adversary who is given public keys $\texttt{pk}_1, \cdots, \texttt{pk}_\texttt{n}$ and has access to encryption of key dependent messages, $f(\texttt{sk}_1, \cdots, \texttt{sk}_\texttt{n})$ for adaptively selected functions $f \in \mathcal{F}$. RLWE-based HE schemes have shown to be KDM[$\mathcal{F}$] secure.

- $\texttt{RingRNS.Enc}_\texttt{pk}(m)$: It encrypts a message m into a ciphertext $\texttt{ct} = (c_0, c_1) = v \cdot \texttt{pk} + (m + e_0, e_1) \in \mathcal{R}^2_{Q_L}$, with v sampled from a distribution χ_{enc}, such as the distribution $\mathcal{ZO}(\rho)$ with $0 \leq \rho \leq 1$ which samples each coefficient of the polynomial from $\{0, \pm 1\}$, but with probability $\rho/2$ for each of $-1, +1$, and $1 - \rho$ for zero.

- $\texttt{RingRNS.Dec}_\texttt{sk}(\texttt{ct})$: It returns an approximation of the message m by decrypting the ciphertext $\texttt{ct}$ with respect to $\texttt{sk}$ modulo q_0, which can be written as, $\langle \texttt{ct}, \texttt{sk} \rangle$. This means, $m \approx (c_0 + c_1 \cdot s) \pmod{q_0}$. The complete equation and its corresponding correctness is discussed in the Appendix A.

- $\texttt{RingRNS.KeySwitch}_\texttt{swk}(\texttt{ct})$: For a ciphertext $\texttt{ct} = (c_0, c_1) \in \mathcal{R}^2_{Q_l}$, compute $\pmod{pQ_l}$ the following: $\texttt{ct}'' = \sum_{i=0}^{l} c_1[i] \cdot \texttt{swk}[i]$. Output the ciphertext, $\texttt{ct}' = \big((c_0, 0) + (\texttt{RingRNS.ModDown}(\texttt{ct}''))\big) \in \mathcal{R}^2_{Q_l}$. The $\texttt{RingRNS.ModDown}$ operation is used to scale down the coefficient modulus from pQ_l to Q_l. This operation is used to transform a ciphertext decryptable under s into a ciphertext decryptable under s' with the help of the key-switching key.

- $\texttt{RingRNS.Add}(\texttt{ct}, \texttt{ct}')$: It adds the polynomial components of the two ciphertexts $\texttt{ct} = (c_0, c_1) \in \mathcal{R}^2_{Q_l}$ and $\texttt{ct}' = (c'_0, c'_1) \in \mathcal{R}^2_{Q_l}$, and computes $\texttt{ct}_{\text{add}} = (d_0, d_1)$ where $d_0 = c_0 + c'_0 \in \mathcal{R}^2_{Q_l}$ and $d_1 = c_1 + c'_1 \in \mathcal{R}^2_{Q_l}$.

- $\texttt{RingRNS.Mult}(\texttt{ct}, \texttt{ct}')$: It multiplies two input ciphertexts $\texttt{ct} = (c_0, c_1) \in \mathcal{R}_{Q_l}^2$ and $\texttt{ct}' = (c_0', c_1') \in \mathcal{R}_{Q_l}^2$, and computes $d_0 = c_0 \cdot c_0' \in \mathcal{R}_{Q_l}$, $d_1 = c_0 \cdot c_1' + c_1 \cdot c_0' \in \mathcal{R}_{Q_l}$, and $d_2 = c_1 \cdot c_1' \in \mathcal{R}_{Q_l}$. The output is the non-linear ciphertext $\texttt{d} = (d_0, d_1, d_2) \in \mathcal{R}_{Q_l}^3$.
- $\texttt{RingRNS.Relin}_{\texttt{evk}}(\texttt{d})$: It relinearizes the result of $\texttt{RingRNS.Mult}$ and produces a ciphertext with two polynomial components so that it is decryptable under the secret key. Now, with the help of the evaluation key, $\texttt{evk} = (evk_0, evk_1) = \texttt{KSGen}(s, s^2)$, one needs to compute $\texttt{ct}'' = (c_0'', c_1'')$ where $c_0'' = \sum_{i=0}^{l-1} d_2'[i] \cdot \texttt{evk}_0[i] \in \mathcal{R}_{pQ_l}$ and $c_1'' = \sum_{i=0}^{l-1} d_2'[i] \cdot \texttt{evk}_1[i] \in \mathcal{R}_{pQ_l}$. The final output is the relinearized ciphertext given by $\texttt{ct}_{\text{relin}} = (d_0, d_1) + (\texttt{RingRNS.ModDown}(c_0''), \texttt{RingRNS.ModDown}(c_1'')) \pmod{Q_l}$. The $\texttt{RingRNS.ModDown}$ operation is used to reduce the coefficient modulus from pQ_l to Q_l. We explain the details of the multiplication and relinearization operations in the Appendix A.
- $\texttt{RingRNS.ModDown}(d)$: For a ciphertext component $d \in \mathcal{R}_{pQ_l}$, let $d''[i] = d[l+1] \pmod{q_j}$ for $0 \le i \le l$. Then, compute the 'scaled down' component $d' = p^{-1} \cdot (d - d'') \pmod{q_i} \in \mathcal{R}_{Q_l}$.
- $\texttt{RingRNS.Rescale}(\texttt{ct})$: It takes a ciphertext $\texttt{ct} = (c_0, c_1) \in \mathcal{R}_{Q_l}^2$ with level l and produces a ciphertext $\texttt{ct}' = (c_0', c_1')$ at level $l-1$. Let, $c_0''[i] = c_0[l] \pmod{q_i}$ for $0 \le i \le l-1$. Then, compute $c_0' = c_0 - c_0'' \in \mathcal{R}_{Q_{l-1}}$. Finally, output the rescaled ciphertext element $c_0' = q_l^{-1} \cdot c_0' \in \mathcal{R}_{Q_{l-1}}$. Similarly, compute the other rescaled ciphertext component $c_1' \in \mathcal{R}_{Q_{l-1}}$.
- $\texttt{RingRNS.Rotate}_{\texttt{rtk}}(\texttt{ct})$: The slot rotation operation takes a ciphertext $\texttt{ct} = (c_0, c_1) \in \mathcal{R}_{Q_l}^2$ and rotation key $\texttt{rtk}$ and performs an automorphism and a key-switch of the ciphertext polynomial coefficients. The rotation key is a key-switching key that has an encryption of the rotated secret key under an automorphism. The ciphertext obtained after the key-switching procedure is then encrypted under this rotated secret key.

The ciphertexts and keys are stored in Number Theoretic Transform (NTT) format to make the sub-routines described above more efficient (polynomial multiplication becomes $O(N \log(N))$). To provide, an algorithmic overview of this we also describe the Algorithm 1 RingRNS.Add, Algorithm 2 RingRNS.Mult, Algorithm 3 RingRNS.ModDown, and Algorithm 4 RingRNS.Relin.

Algorithm 1 RingRNS.Add [CHK+18b]

In: $\texttt{c} = (\tilde{c}_0, \tilde{c}_1)$, $\texttt{c}' = (\tilde{c}_0', \tilde{c}_1') \in R_{Q_l}^2$
Out: $\texttt{d} = (\tilde{d}_0, \tilde{d}_1) \in R_{Q_l}^2$
1: $\tilde{d}_0 \leftarrow \tilde{c}_0 + \tilde{c}_0'$
2: $\tilde{d}_1 \leftarrow \tilde{c}_1 + \tilde{c}_1'$

Algorithm 2 RingRNS.Mult [CHK+18b]

In: $\texttt{ct} = (\tilde{c}_0, \tilde{c}_1)$, $\texttt{ct}' = (\tilde{c}_0', \tilde{c}_1') \in R_{Q_l}^2$
Out: $\texttt{d} = (\tilde{d}_0, \tilde{d}_1, \tilde{d}_2) \in R_{Q_l}^3$
1: $\tilde{d}_0 \leftarrow \tilde{c}_0 \star \tilde{c}_0'$, $\tilde{d}_2 \leftarrow \tilde{c}_1 \star \tilde{c}_1'$
2: $\tilde{d}_1 \leftarrow \tilde{c}_0 \star \tilde{c}_1' + \tilde{c}_1 \star \tilde{c}_0'$

Algorithm 3 RingRNS.ModDown [CHK+18b]

In: $\tilde{d} \in R_{pQ_l}$
Out: $\tilde{d}' \in R_{Q_l}$
1: $t \leftarrow \texttt{INTT}(\tilde{d}[l])$
2: **for** $i = 0$ to $l-1$ **do**
3: $\quad \tilde{t} \leftarrow \texttt{NTT}([t]_{q_i})$ $\quad \triangleright$ in $\mathbb{Z}_{q_i}$
4: $\quad \tilde{d}'[i] \leftarrow [p^{-1} \cdot (\tilde{d}[i] - \tilde{t})]_{q_i}$
5: **end for**

Algorithm 4 `RingRNS.Relin` [CHK+18b]

In: $\mathtt{d} = (\tilde{d}_0, \tilde{d}_1, \tilde{d}_2) \in R^3_{Q_l}$, $\tilde{\mathtt{evk}}_0 \in R^l_{pQ_l}$, $\tilde{\mathtt{evk}}_1 \in R^l_{pQ_l}$
Out: $\mathtt{d}' = (\tilde{d}'_0, \tilde{d}'_1) \in R^2_{Q_l}$
1: **for** $j = 0$ to $l-1$ **do**
2: $\quad d_2[j] \leftarrow \mathtt{INTT}(\tilde{d}_2[j])$ ▷ in $\mathbb{Z}_{q_j}$
3: **end for**
4: **for** $j = 0$ to l **do** ▷ Here q_l is used to represent special prime p
5: $\quad (\tilde{c}''_0[j], \tilde{c}''_1[j]) \leftarrow 0$
6: $\quad$ **for** $i = 0$ to $l-1$ **do**
7: $\quad\quad \tilde{r} \leftarrow \mathtt{NTT}(\left[d_2[i]\right]_{q_j})$ ▷ in $\mathbb{Z}_{q_j}$
8: $\quad\quad \tilde{c}''_0[j] \leftarrow \left[\tilde{c}''_0[j] + \tilde{\mathtt{evk}}_0[i][j] \star \tilde{r}\right]_{q_j}$, $\tilde{c}''_1[j] \leftarrow \left[\tilde{c}''_1[j] + \tilde{\mathtt{evk}}_1[i][j] \star \tilde{r}\right]_{q_j}$
9: $\quad$ **end for**
10: **end for**
11: $\tilde{d}'_0 \leftarrow \tilde{d}_0 + \mathtt{RingRNS.ModDown}(\tilde{c}''_0)$, $\tilde{d}'_1 \leftarrow \tilde{d}_1 + \mathtt{RingRNS.ModDown}(\tilde{c}''_1)$

2.4 Encoding using complex embeddings

Since we work with an underlying polynomial ring structure, we would also expect messages to be in the form of polynomials. But more often than not data comes in the form of vectors of plaintexts. [CKKS17] discusses a method to 'pack' multiple messages in one ciphertext.

Let $\mathcal{K}$ be the cyclotomic field such that $\mathcal{R} \subseteq \mathcal{K}$. The complex canonical embeddings are the ring homomorphisms $\tau_j : \mathcal{K} \to \mathbb{C}$ such that for a $2N$-th root of unity ξ, $\tau_j : \xi \mapsto \xi_j$ where $j \in \mathbb{Z}^*_{2N}$. The canonical embedding $\tau : \mathcal{K} \to \mathbb{C}^N$ can be defined as, $\tau(\mathbf{z}) = (\tau_j(\mathbf{z}))_{j \in \mathbb{Z}^*_{2N}}$, $\mathbf{z} = (z_j)_{j \in \mathbb{Z}^*_{2N}}$ where addition and multiplication in $\mathbb{C}^N$ are component-wise. Since $\tau_{-j} = \bar{\tau}_j$, so there can exist a natural projection $\pi : \mathbb{H} \to \mathbb{C}^{N/2}$ where, $\mathbb{H} = \{(z_j)_{j \in \mathbb{Z}^*_{2N}} : z_{-j} = \bar{z_j}, j \in \mathbb{Z}^*_{2N}\}$. The encoding process thus involves transforming a vector $\mathbf{z}$ under the inverse of π and then using the inverse of the canonical embedding with a rounding operation (usually accompanied with the multiplication of a scaling factor $\Delta \geq 1$ to maintain precision) to finally obtain a polynomial in $\mathcal{R}$. Hence, a plaintext is the polynomial $m(X) = \tau^{-1} \circ \pi^{-1}(\mathbf{z}) \in \mathcal{R}$. The decoding procedure is simply, $\mathbf{z} = \pi \circ \tau(m(X)) \in \mathbb{C}^{N/2}$.

3 Proposed MLWE-HE scheme

Let λ be a security parameter that governs the dimension of the underlying ring N, the rank of the module r, the maximum level L and the ciphertext modulus $q_l > 0$ for $0 \leq l \leq L$, and an integer $P > 0$. For realizing a module lattice-based homomorphic encryption scheme, we port the RLWE-based CKKS to the MLWE setting with RNS optimization. We use the prefix `Mod` to describe its subroutines. Let $\mathbf{A} \in \mathcal{R}^{r \times r}_{q_L}$ consist of polynomials a sampled uniformly from $\mathcal{U}(\mathcal{R}_{q_L})$, $\mathbf{s}$ be the secret whose components are chosen randomly from r copies of the set $\mathcal{HWT}(h)$ of signed binary polynomials $\{0, \pm 1\}^N$ with Hamming weight h and $\mathbf{e}$ be the error each of whose polynomial coefficients is sampled independently from a discrete Gaussian distribution $\mathcal{DG}(\sigma^2)$, $\sigma > 0$. We also consider another distribution $\mathcal{ZO}(\rho)$ with $0 \leq \rho \leq 1$ which samples each entry in the vector also from $\{0, \pm 1\}^N$, but with probability $\rho/2$ for each of $-1, +1$, and $1 - \rho$ for zero. Let any message $m \in \mathcal{R}$ be encoded under some encoding scheme, like the one described in section 2.

- `Mod.KeyGen`(1^λ): Generate a secret key $\mathtt{sk} = (1, \mathbf{s})$ with each secret polynomial $s_i \leftarrow \mathcal{HWT}(h)$. Sample the random matrix $\mathbf{A}$ from $\mathcal{U}(\mathcal{R}^{r \times r}_{q_L})$ and each error polynomial $e_i \leftarrow \mathcal{DG}(\sigma^2)$ for the error vector $\mathbf{e}$. Generate public key $\mathtt{pk} = (\mathbf{b}, \mathbf{A}) = (-\mathbf{A} \cdot \mathbf{s} + \mathbf{e}, \mathbf{A}) \pmod{q_L} \in \mathcal{R}^r_{q_L} \times \mathcal{R}^{r \times r}_{q_L}$.

- $\texttt{Mod.KSGen}(\mathbf{s}, \mathbf{s}')$: Given two secrets $\mathbf{s}$ and $\mathbf{s}'$, output the key-switching key $\texttt{swk} = (\mathbf{b}_{swk}, \mathbf{A}_{swk}) = (-\mathbf{A}_{swk} \cdot \mathbf{s} + \mathbf{e}_{swk} + P \cdot \mathbf{s}', \mathbf{A}_{swk}) \pmod{P \cdot q_L} \in \mathcal{R}^r_{Pq_L} \times \mathcal{R}^{r\times r}_{Pq_L}$. Here $\mathbf{A}_{swk} \in \mathcal{U}(\mathcal{R}^r_{Pq_L} \times \mathcal{R}^{r\times r}_{Pq_L})$. Each error polynomial in $\mathbf{e}_{swk}$ is sampled from $\mathcal{DG}(\sigma^2)$.
- $\texttt{Mod.Enc}_{\texttt{pk}}(m)$: We obtain a ciphertext encrypting a message m, $\texttt{ct} = (c_0, \mathbf{c_1}) = (\texttt{pk} \cdot \mathbf{v} + (m + e, \mathbf{e}')) \pmod{q_L} \in \mathcal{R}_{q_L} \times \mathcal{R}^r_{q_L}$, $\mathbf{v} \in \mathcal{R}^r_{q_L}$ where the polynomials v_i of vector $\mathbf{v}$ are sampled as $v_i \leftarrow \mathcal{ZO}(0.5)$.
- $\texttt{Mod.Dec}_{\texttt{sk}}(\texttt{ct})$: We obtain an approximation of the message after decryption under $\texttt{sk}$, $(c_0 + \mathbf{c_1} \cdot \mathbf{s}) \pmod{q_l} \approx m$.
- $\texttt{Mod.KeySwitch}_{\texttt{swk}}(\texttt{ct})$: For a ciphertext $\texttt{ct} \in \mathcal{R}_{q_l} \times \mathcal{R}^r_{q_l}$, output the ciphertext, $\texttt{ct}' = (c_0, 0) + \lfloor P^{-1} \cdot \mathbf{c_1} \cdot \texttt{swk} \rceil \in \mathcal{R}_{q_l} \times \mathcal{R}^r_{q_l}$.
- $\texttt{Mod.Add}(\texttt{ct}, \texttt{ct}')$: Given two ciphertexts $\texttt{ct}$ and $\texttt{ct}' \in \mathcal{R}_{q_l} \times \mathcal{R}^r_{q_l}$, their sum is a sum of their corresponding components.

$$\texttt{ct}_{\texttt{add}} = \texttt{ct} + \texttt{ct}' = (c_0 + c_0', \mathbf{c_1} + \mathbf{c_1'}) \pmod{q_l} \in \mathcal{R}_{q_l} \times \mathcal{R}^r_{q_l}.$$

- $\texttt{Mod.Mult}(\texttt{ct}, \texttt{ct}')$: For the multiplication operation $'*'$ between two ciphertexts $\texttt{ct}$ and $\texttt{ct}' \in \mathcal{R}_{q_l} \times \mathcal{R}^r_{q_l}$ to be homomorphic we would like to have,

$$\begin{aligned} m \cdot m' &\approx \texttt{Mod.Dec}((c_0, \mathbf{c_1}) * (c_0', \mathbf{c_1'})) \\ &\approx \texttt{Mod.Dec}(c_0, \mathbf{c_1}) * \texttt{Mod.Dec}(c_0', \mathbf{c_1'}) \\ &\approx ((c_0 + \mathbf{c_1} \cdot \mathbf{s}) \cdot (c_0' + \mathbf{c_1'} \cdot \mathbf{s})) \pmod{q_l} \\ &\approx c_0c_0' + c_0 \sum_{i=0}^{r-1} c_{1i}' s_i + c_0' \sum_{i=0}^{r-1} c_{1i} s_i + \sum_{i=0}^{r-1}\sum_{j=0}^{r-1} c_{1i}c_{1j}' s_i s_j. \end{aligned}$$

We write the resultant ciphertext of the multiplication as, $\texttt{ct}_{\texttt{mult}} = \texttt{d} = (d_0, \mathbf{d_1}, \mathbf{d_2}, \mathbf{d_3}) \in \mathcal{R}_{q_l} \times \mathcal{R}^r_{q_l} \times \mathcal{R}^r_{q_l} \times \mathcal{R}^{r(r-1)/2}_{q_l}$ where:

$$\begin{aligned} d_0 &= c_0 \cdot c_0' & d_{1i} &= c_0 \cdot c_{1i}' + c_0' \cdot c_{1i} \\ d_{2i} &= c_{1i} \cdot c_{1i}' & d_{3_{ij}} &= c_{1i} \cdot c_{1j}' + c_{1i}' \cdot c_{1j},\ i < j \end{aligned}$$

with all the above arithmetic done modulo q_l. Note that $\mathbf{c_1}$ and $\mathbf{c_1'}$ are in $\mathcal{R}^r_{q_l}$ and the notation c_{1i} refers to the i-th component of $\mathbf{c_1}$ for $0 \leq i \leq r-1$. The notation $d_{3_{ij}}$ refers to the component of $\mathbf{d_3}$ that is obtained as a result of multiplications between the '$1i$'-th component and the '$1j$'-th component of the ciphertexts $\texttt{ct}$ and $\texttt{ct}'$.

The ciphertext thus obtained is 'extended' in the sense that it now has more components than a usual ciphertext. For example, in the case of rank $r = 2$, the extended decryption equation would look like,

$$m \cdot m' \approx (d_0 + \mathbf{d_1} \cdot \mathbf{s} + \mathbf{d_2} \cdot \mathbf{s}^2 + d_3 \cdot s_0 s_1) \pmod{q_l}. \tag{1}$$

So just after one multiplication, there is a growth in the number of ciphertext components from 2 to 4. These changes are not desirable in the design and we would want to find a way to get $\texttt{ct}_{\texttt{mult}}$ to mimic a usual ciphertext so that it can be decrypted by the usual secret key components, $\texttt{sk}$. Like, for eqn. 1 it would mean that we would want to look for expressions d_0' and $\mathbf{d_1'}$ such that it can be rewritten approximately as, $d_0' + \mathbf{d_1'} \cdot \mathbf{s} \approx d_0 + \mathbf{d_1} \cdot \mathbf{s} + \mathbf{d_2} \cdot \mathbf{s}^2 + d_3 \cdot s_0 s_1$.

The concepts of relinearization and rescaling come in as a method for converting a degree 2 ciphertext again into a degree 1 ciphertext that can be decrypted under the secret key $\texttt{sk} = (1, \mathbf{s})$.

- $\texttt{Mod.Relin}_{\texttt{evk}}(\texttt{ct}_{\texttt{mult}})$: Let $P = P(\lambda, q_L)$ be an integer that will be used to control the increase in error after multiplication.
  Continuing with our example of $r = 2$, in order to do away with the non-linear terms (in $\mathbf{s}$) from eqn. (1) with respect to $\mathbf{d_2}$ and d_3, we take help of the following evaluation keys: $\texttt{evk}_{\texttt{d}_2} = \texttt{Mod.KSGen}(\mathbf{s}, \mathbf{s^2})$ and $\texttt{evk}_{\texttt{d}_3} = \texttt{Mod.KSGen}(\mathbf{s}, s_0 s_1)$, where for a rank of two the polynomials of the public matrix are sampled uniformly from $\mathcal{R}_{P \cdot q_L}$.

  We can now define the relinearization components as,

$$d_0' = (d_0 + P^{-1} \cdot \mathbf{d_2} \cdot \texttt{evk}_{\texttt{d}_2}[0] + P^{-1} \cdot d_3 \cdot \texttt{evk}_{\texttt{d}_3}[0]) \pmod{q_l}$$
$$\mathbf{d_1'} = (\mathbf{d_1} + P^{-1} \cdot \mathbf{d_2} \cdot \texttt{evk}_{\texttt{d}_2}[1] + P^{-1} \cdot d_3 \cdot \texttt{evk}_{\texttt{d}_3}[1]) \pmod{q_l}$$

  where components of $\mathbf{d_2}$ and d_3 have been assumed to have been lifted (*modulus switch*) from residue ring modulo q_l to residue ring modulo $P \cdot q_l$ as is done in [CKKS17].
  Hence, the central idea here is that by using evaluation keys we adjust the terms that correspond to secret key components of the form $\sum_{i=0}^{r-1} \sum_{j=0}^{r-1} s_i s_j$, that is, relinearization keys are encryptions of these secret key components.

  For a general rank r module, these keys have the form, $\texttt{evk} = \texttt{Mod.KSGen}(\mathbf{s}, \mathbf{s}')$ where $\mathbf{s}'$ is a vector of polynomials of the form $s_i s_j$ for $0 \le i, j < r$ and $j \ge i$. More specifically, $\mathbf{s}'$ is $\mathbf{s^2}$ in the case of $\texttt{evk}_{d_2}$ and in $\texttt{evk}_{d_3}$ consists of terms $s_i s_j$ for $0 \le i, j < r; j > i$ respectively. So, $\texttt{ct}_{\texttt{relin}} = (d_0', \mathbf{d_1'}) = \big(d_0 + P^{-1}(\mathbf{d_2} \cdot \texttt{evk}_{d_2}[0] + \mathbf{d_3} \cdot \texttt{evk}_{d_3}[0]), \mathbf{d_1} + P^{-1}(\mathbf{d_2} \cdot \texttt{evk}_{d_2}[1] + \mathbf{d_3} \cdot \texttt{evk}_{d_3}[1])\big) \pmod{q_l} \in \mathcal{R}_{q_l} \times \mathcal{R}_{q_l}^r$.

- $\texttt{Mod.Rescale}(\texttt{ct})$: Since every message has an inherent scaling factor, say $\Delta \ge 1$ to preserve a certain degree of precision, multiplications between ciphertexts also result in an exponential increase in the scaling factor size. Therefore, to keep the scale constant and also to reduce the noise, we can define the rescaling operation of a ciphertext $\texttt{ct}$ in a level l to a level $l-1$ as, $\texttt{ct}' = \left\lfloor \frac{q_{l-1}}{q_l} \cdot \texttt{ct} \right\rceil \pmod{q_{l-1}}$.

Additionally, we also mention the two subroutines of CKKS called rotation and conjugation. It is intuitive to understand how messages in the same i-th plaintext slot of a ciphertext could be added or multiplied. However, to operate between messages in two different plaintext slots, there should be a way to permute data easily across slots of the ciphertext. In this regard, permutations and rather automorphisms of the associated algebraic ring can be efficiently used for the process of rotation of slots. Like, for a polynomial a in a power-of-two cyclotomic ring $\mathcal{R}$ replacing it with $a^{(k)}(X) = a(X^k) \pmod{\phi_{2N}(X)}$, $k \in \mathbb{Z}_{2N}^*, k > 1$ would result only in a permutation of the coefficients. In particular, $\kappa_k : a(X) \to a(X^k) \pmod{\phi_{2N}(X)}$ would serve as the suitable rotation map. So a ciphertext $\texttt{ct}$ encrypting a message m with respect to a secret key $\mathbf{s}$ would have a 'rotated' ciphertext $\kappa_k(\texttt{ct})$ which is a valid encryption of $\kappa_k(m)$ with secret key $\kappa_k(\mathbf{s})$. This operation is equivalent to a key-switching technique similar to the ring setting and requires a switching key $\texttt{swk}$ where s' takes the form of $\kappa_k(\mathbf{s})$ and for brevity, we denote the corresponding public matrix by $\mathbf{A}_{rot}$. Notice that one of the generating sets of the group of units $\mathbb{Z}_{2N}^*$ is $\{5, -1\}$ as the integer 5 has order $N/2$ in $\mathbb{Z}_{2N}^*$ (by Euler's Totient Theorem, we atleast have, $5^N \equiv 1 \pmod{2N}$ and order of the element 5 must also divide the order of the group so order of 5 is an integer less than or equal to N. Observe that for a power-of-two integer N, $5^{N/2} \equiv 1 \pmod{2N}$ by induction, but no integer less than $N/2$ would satisfy the congruence). If $k = 5^{i-j} \pmod{2N}$ for $0 \le i, j < N/2$, then $m(\xi_j^{5^{i-j}}) = m(\xi_i)$, which resembles a permutation of the plaintext slots.

- $\texttt{Mod.Rotate}_{\texttt{rtk}}(\texttt{ct})$: The associated rotation key is given by the key-switching key generation subroutine by, $\texttt{rtk} = \texttt{Mod.KSGen}(\mathbf{s}, \kappa_k(\mathbf{s}))$. The rotated ciphertext is $\texttt{ct}' = (c_0, 0) + (P^{-1} \cdot \mathbf{c_1} \cdot \texttt{rtk}) \pmod{q_l} \in \mathcal{R}_{q_l} \times \mathcal{R}_{q_l}^r$.

- $\texttt{Mod.Conjugate}_{\texttt{cjk}}(\texttt{ct})$: Another property to note here is that for $2N$-th primitive roots of unity ξ_j, we know $\bar{\xi_j} = \xi_j^{-1}$. Then for a polynomial a in $\mathcal{R}$, $\overline{a(\xi_j)} = a(\bar{\xi_j}) = a(\xi_j^{-1})$. Considering $k = -1$ for the mapping κ_k, the ciphertext $\bar{\texttt{ct}}$ can be seen as the encryption of the conjugate $\bar{\mathbf{z}}$ of a vector $\mathbf{z}$. The conjugate key is a switch-key with s' as $\kappa_{-1}(\mathbf{s})$ and can be written as, $\texttt{cjk} = \texttt{Mod.KSGen}(\mathbf{s}, \kappa_{-1}(\mathbf{s}))$ and $\bar{\texttt{ct}} = (c_0, 0) + (P^{-1} \cdot \mathbf{c_1} \cdot \texttt{cjk}) \pmod{q_l} \in \mathcal{R}_{q_l} \times \mathcal{R}_{q_l}^r$.

In the end, we include a new subroutine which we refer to as 'rank-reduction' that also works similar to the key-switching operation. In leveled homomorphic encryption the ciphertext modulus diminishes as more and more levels get consumed (see the rescaling operation). For a smaller ciphertext modulus, the rank of the lattice could be reduced to an appropriate r' from r (where $r' < r$) without lowering the security of the homomorphic encryption scheme. An example is used to explain the above fact. Let, the initial ciphertext modulus size be $log_2(q_L) = 220$ and the rank be $r = 2$ for 128-bit security. Let, after several homomorphic multiplications the ciphertext modulus size reduces to $log_2(q_l) = 110$. With the current modulus size and rank $r = 2$ the lattice instance would be almost 256-bit secure. The proposed rank reduction procedure could be used to obtain an equivalent ciphertext at a lower rank e.g., $r = 1$ while achieving around 128-bit security. Reducing the rank significant improves the speed and memory requirement. We discuss its usefulness in section 4.

More formally, the rank reduction procedure takes an MLWE rank-r ciphertext $(c_0, \mathbf{c_1}) \in \mathcal{R}_{q_l} \times \mathcal{R}_{q_l}^r$ which is decryptable under the secret $\mathbf{s} = (s_0, \cdots s_{r-1}) \in \mathcal{R}^r$. It produces an equivalent ciphertext $(c_0', \mathbf{c_1'}) \in \mathcal{R}_{q_l} \times \mathcal{R}_{q_l}^{r'}$ of a lower rank $r' < r$ which is decryptable under the secret $\mathbf{s}' = (s_0', \cdots s_{r'-1}') \in \mathcal{R}^{r'}$. Let, $\mathbf{s}'' = \{s_{r'}, \cdots, s_{r-1}\} \in \mathcal{R}^{r-r'}$.

- $\texttt{Mod.RedKeySwitch}(\mathbf{s}', \mathbf{s}'')$: Generate the public matrix $\mathbf{A}_{red} \in \mathcal{R}_{Pq_l}^{(r-r') \times r'}$ by sampling its polynomials from $\mathcal{U}(\mathcal{R}_{Pq_l})$. Generate $\mathbf{e}_{red} \in \mathcal{R}^{r-r'}$ by sampling its error polynomials from $\mathcal{DG}(\sigma^2)$. Compute the rank reduction key, $\texttt{redk} = (b_{red}, \mathbf{A}_{red}) = (-\mathbf{A}_{red} \cdot \mathbf{s}' + \mathbf{e}_{red} + P \cdot \mathbf{s}'', \mathbf{A}_{red}) \pmod{P \cdot q_l} \in \mathcal{R}_{Pq_l}^{r-r'} \times \mathcal{R}_{Pq_l}^{(r-r') \times r'}$.

  In line with circular security as mentioned in section 2.3, we adopt the notion of 'extended circular security' where some components of the secret are securely encrypted under the remaining secret components.

- $\texttt{Mod.Rankred}_{\texttt{redk}}(\texttt{ct})$: For a ciphertext $\texttt{ct} \in \mathcal{R}_{q_l} \times \mathcal{R}_{q_l}^r$, the components of the rank-reduced ciphertext $\texttt{ct}' = (c_0', \mathbf{c_1'}) \in \mathcal{R}_{q_l} \times \mathcal{R}_{q_l}^{r'}$ are then given by $\big(c_0' = c_0 + P^{-1} \cdot \mathbf{c}_{\text{rem}} \cdot \texttt{redk}[0]\big) \pmod{q_l}$ and, $\big(\mathbf{c_1'} = \mathbf{c}_{\text{red}} + P^{-1} \cdot \mathbf{c}_{\text{rem}} \cdot \texttt{redk}[1]\big) \pmod{q_l}$, where, $\mathbf{c}_{\text{red}} = (c_{1k})_{0 \le k < r'}$ and $\mathbf{c}_{\text{rem}} = (c_{1j})_{r' \le j < r}$, are respectively the 'reduced' and the 'removed' components of $\mathbf{c_1}$.

We note here that for $r \ge 3$ many successive rank reductions from $r \to (r-1)$ can lead to linear combinations between secret components but with respect to different ciphertext moduli. Thorough analysis remains to be performed to understand whether or not such combinations could be exploited by new cryptanalysis. However, as a simple solution, one could always perform rank reduction securely from $r \to r/2$.

3.1 Noise estimations

Note that in a ring setting, the error bound grows in proportion to the size of the ring. Instead, in a module setting, the size of the ring remains fixed and the change in the error bound follows the rank of the module. For a similar security level, the lattice dimension $N \cdot r$ of an MLWE instance with the rank r and ring dimension N will be roughly equal to the ring dimension $N' = N \cdot r$ of the RLWE instance but with N fixed. Therefore the

error growth in an MLWE instance with lattice dimension $N \cdot r$ will depend on r along with a smaller N.

We discuss the following error estimations along the lines of [CKKS17], [CHK+18b], and [CS16]. First, we mention a few facts about the various distributions that the polynomials have been sampled from: Let all sampled coefficients be independent and identically distributed, and let σ^2 be the variance of each such coefficient. Then, a polynomial sampled from a uniform distribution $\mathcal{U}$ over $\mathcal{R}_q$ has a variance of $q^2N/12$, a polynomial sampled from the discrete Gaussian distribution $\mathcal{DG}(\sigma^2)$ of mean centered around zero has variance $\sigma^2 N$ and a polynomial sampled from $\mathcal{ZO}(\rho)$ has variance ρN. For a distribution $\mathcal{HWT}(h)$ over signed binary integers $\{0, \pm 1\}$ the variance is just its Hamming weight, h. In the case of a multiplication of two independent random variables sampled from Gaussian distributions with variances σ_1^2 and σ_2^2, the high-probability bound is set to $16\sigma_1\sigma_2$. As a consequence of the *law of large numbers*, the high-probability bound on the (ring) canonical embedding norm is taken to be 6σ.

Lemma 1. *The error induced during encryption is bounded by* $B_{enc} = 16r\sigma(N/\sqrt{2} + \sqrt{hN}) + 6\sigma\sqrt{N}$.

Proof. Consider the decryption equation of a ciphertext $\texttt{ct}$ encrypting a message m.

$$\begin{aligned} c_0 + \mathbf{c_1} \cdot \mathbf{s} &= ((\texttt{pk}[0] \cdot \mathbf{v} + m + e) + (\texttt{pk}[1] \cdot \mathbf{v} + \mathbf{e'}) \cdot \mathbf{s}) \pmod{q_l} \\ &= (m + \mathbf{e_{pk}} \cdot \mathbf{v} + e + \mathbf{e'} \cdot \mathbf{s}) \pmod{q_l} \\ &= (m + \sum_{i=0}^{r-1} e_{i_{pk}} \cdot v_i + e + \sum_{i=0}^{r-1} e'_i \cdot s_i) \pmod{q_l} \end{aligned}$$

Let, $E = (\sum_{i=0}^{r-1} e_{i_{pk}} \cdot v_i + e + \sum_{i=0}^{r-1} e'_i \cdot s_i)$. The upper bound can then be established with the following inequality:

$$\begin{aligned} \|E\|_\infty^{can} &\leq r \cdot \frac{16 \cdot \sigma \cdot N}{\sqrt{2}} + 6\sigma\sqrt{N} + r \cdot 16\sigma\sqrt{hN} \\ &\leq 16r\sigma(N/\sqrt{2} + \sqrt{hN}) + 6\sigma\sqrt{N} \end{aligned}$$

□

If $\mathbf{z} \in \mathbb{C}^{N/2}$ and Δ is the scaling factor being used, an encryption of the encoded message m is also an encryption of $\Delta \cdot \tau^{-1} \circ \pi^{-1}(\mathbf{z})$ which would have an error upper bound equal to $B_{enc} + N/2 = B'$ (say). Then for the equality, $\pi(\tau(\lfloor \Delta^{-1}(m + B') \rceil)) = \pi(\tau(\lfloor \Delta^{-1}(m) \rceil))$ to hold we need to have $\Delta^{-1}B' < 1/2$. Thus, if $\Delta > N + 2B_{enc}$, the decoded message (after decryption) will correctly return $\mathbf{z}$.

Lemma 2. *Intuitively, the error bound after one addition is the sum of the error bounds corresponding to the individual ciphertexts.*

Lemma 3. *The error induced in the rescaling step is bounded by* $B_{res} = 6\sqrt{N/12} + r \cdot 16\sqrt{hN/12}$.

Proof. The error induced during rescaling is because of the fact that we try to approximate a ciphertext $\texttt{ct}$ with $\texttt{ct}'$ using $\frac{q_{l-1}}{q_l}\texttt{ct}$. Thus an error bound E_{res} can be found on the error vector using the expression,

$$\texttt{ct}' - \frac{q_{l-1}}{q_l}\texttt{ct} \pmod{q_{l-1}} = (\epsilon_0, \boldsymbol{\epsilon_1}).$$

Assuming that each coefficient of the polynomials in the error vector has an approximate variance of $1/12$, we write the error bound during the decryption of this vector by the

following inequality:

$$\begin{aligned}((\epsilon_0, \epsilon_{\mathbf{1}}), \mathbf{s}) &= (\epsilon_0 + \sum_{i=0}^{r-1} \epsilon_{1i} \cdot s_i) \\ &\leq \|\epsilon_0\|_\infty^{can} + \sum_{i=0}^{r-1} \|\epsilon_{1i} \cdot s_i\|_\infty^{can} \\ &\leq 6\sqrt{N/12} + r \cdot 16\sqrt{hN/12}.\end{aligned}$$

□

Lemma 4. *The cumulative error bound after a homomorphic multiplication is the sum of the bounds of* $B_{res} + B_{mult} + B_{relin}$ *where* B_{mult} *and* B_{relin} *are the upper bounds of errors induced during the actual multiplication steps and relinearization respectively.*

Proof. Homomorphic multiplication involves a series of operations, starting with actual multiplications, relinearization, and rescaling. Each of these steps contributes to the error growth. First, note that the multiplication $m \cdot m'$ is approximated by $(c_o + \mathbf{c_1} \cdot \mathbf{s})(c'_o + \mathbf{c'_1} \cdot \mathbf{s}) \pmod{q_l}$ with respect to the two ciphertexts $\mathtt{ct}$ and $\mathtt{ct}'$. Let $\langle \mathtt{ct}, \mathbf{s} \rangle = m + E \pmod{q_l}$ and $\langle \mathtt{ct}', \mathbf{s} \rangle = m' + E' \pmod{q_l}$ such that $\|E\|_\infty^{can}$ and $\|E'\|_\infty^{can}$ have error bounds B and B' respectively, then we may write the error expression corresponding to multiplication as:

$$\begin{aligned}(m + E)(m' + E') \pmod{q_l} &= (mm' + mE' + m'E + EE') \pmod{q_l} \\ \|E_{mult}\|_\infty^{can} &\leq \|mE' + m'E + EE'\|_\infty^{can} \\ &\leq \mu B' + \mu' B + BB'\end{aligned}$$

where, μ and μ' are respectively the upper bounds of the message space of m and m'. Next, some error is also induced during relinearization when $\mathbf{d_2}$ and $\mathbf{d_3}$ are multiplied with their respective evaluation keys. More precisely we can write the error during relinearization with the following inequality:

$$\begin{aligned}E_{relin} &= P^{-1} \cdot ((\mathbf{d_2} \cdot \mathbf{e_{d_2}} + \mathbf{d_3} \cdot \mathbf{e_{d_3}}) \pmod{P \cdot q_l}) \\ \|E_{relin}\|_\infty^{can} &\leq P^{-1} \cdot r(r+1)/2 \cdot 16\sqrt{\frac{Nq_l^2}{12}}\sigma\sqrt{N}\end{aligned}$$

□

Lemma 5. *The error induced in the rank-reduction step is bounded by* $B_{red} + B_{res}$ *where,* $B_{red} = P^{-1} \cdot (r - r') \cdot 16\sqrt{\frac{Nq_l^2}{12}}\sigma\sqrt{N}$.

Proof. The error induced due to this process comes from the terms of the form $\sum_{j=r'}^{r-1} c_{1j} \cdot e_j$ and one rescaling. If B_{red} denotes the upper bound of the error induced just during the key-switching phase, $\|E_{red}\|_\infty$ and B_{res} is the upper bound of the error introduced during rescaling then the total error bound would be $B_{red} + B_{res}$ where we have the following expression bound for $\|E_{red}\|_\infty$:

$$\|E_{red}\|_\infty \leq P^{-1} \cdot (r - r') \cdot 16\sqrt{\frac{Nq_l^2}{12}}\sigma\sqrt{N}$$

□

3.2 RNS representation for ease of implementation: ModRNS

In section 2.3, we described how the RNS representation can improve the efficiency of HE operations. A natural extension would be also to define an RNS variant of Mod. We can define the module isomorphism between $\mathcal{R}_Q^r$ and the module $\prod_{i=0}^{k-1} \mathcal{R}_{q_i}^r$ which is the direct product of the modules $\mathcal{R}_{q_i}^r$, $i \in \{0, \cdots, k-1\}$ such that,

$$\mathbf{a} = (a_t)_{t=0}^{r-1} \mapsto ([a_0]_{\mathcal{C}}, [a_1]_{\mathcal{C}}, \cdots, [a_{r-1}]_{\mathcal{C}}) = [\mathbf{a}]_{\mathcal{C}}$$

and $[a_t]_{\mathcal{C}} = (a_t^{(0)}, a_t^{(1)}, \cdots, a_t^{(k-1)}) \in \prod_{i=0}^{k-1} \mathcal{R}_{q_i}$. For simplicity, we will use either the notation $\mathbf{a}^{(j)}$ or $\mathbf{A}^{(j)}$ as per context to specify that each component of the module element has its corresponding j residue polynomials. The RNS variants of all Mod algorithms follow. We denote them by ModRNS. We use the representation $\prod_{j=0}^{l} \mathcal{R}_{q_j}^r = \mathcal{R}_{Q_l}^r$ in the subroutines below as well as in their pseudo-codes (Algorithms 5, 6, 7, 8, 9, 10).

- $\texttt{ModRNS.Setup}(q, L, \eta; 1^\lambda)$: For security parameter λ we choose an integer q which determines the approximate basis, maximum levels L as before, a bit precision η, ensuring $q_j/q \in (1-2^{-\eta}, 1+2^{-\eta})$ for $1 \leq j \leq L$. We choose a prime p and the basis $\mathcal{C} = \{q_0, \cdots, q_L\}$ such that at a certain level $0 \leq l \leq L$, $\mathcal{C}_l = \{q_0, \cdots, q_l\}$. For simplicity of equation and notations, we also refer to p as q_{l+1}. Let $\hat{q}^{(j)} = Q/q_j$, $\hat{q'}^{(j)} = (\hat{q}^{-1})^{(j)} \pmod{q_j}$ such that, $B^{(j)} = \hat{q'}^{(j)} \cdot q_j$.

- $\texttt{ModRNS.KeyGen}(1^\lambda)$: Generate a secret key $\texttt{sk} = (1, \mathbf{s})$ where $\mathbf{s}$ consists of polynomials $s_i \leftarrow \mathcal{HWT}(h)$. For the public matrix $\mathbf{A} \in \mathcal{U}(\prod_{j=0}^{L} \mathcal{R}_{q_j}^{r\times r})$ comprising of residue polynomials $\mathbf{a} \in \prod_{j=0}^{L} \mathcal{R}_{q_j}$, sample each $\mathbf{a} = [\mathbf{a}]_{\mathcal{C}}$'s such that the residues are in $\mathcal{U}$ over $\prod_{j=0}^{L} \mathcal{R}_{q_j}$. Generate public key $\texttt{pk} = \big(\texttt{pk}^{(j)} = (\mathbf{b}^{(j)} = -\mathbf{A}^{(j)} \cdot \mathbf{s} + \mathbf{e}, \mathbf{A}^{(j)}) \in \mathcal{R}_{q_j}^r \times \mathcal{R}_{q_j}^{r\times r}\big)_{0\leq j\leq L}$.

- $\texttt{ModRNS.KSGen}(\mathbf{s}, \mathbf{s}')$: Given two secrets $\mathbf{s}$ and $\mathbf{s}'$, output the key-switching key $\texttt{swk} = (\texttt{swk}^{(j,i)} = \mathbf{b}_{swk}^{(j,i)}, \mathbf{A}_{swk}^{(j,i)})_{0\leq j<L, 0\leq i\leq L}$ such that, $\mathbf{b}_{swk}^{(j,i)} = \big(-\mathbf{A}_{swk}^{(j,i)} \cdot \mathbf{s} + \mathbf{e}_{swk} + p \cdot B^{(j)} \cdot \mathbf{s}'\big) \in \mathcal{R}_{q_i}^r$ for $\mathbf{A}_{swk}^{(j,i)}$ uniformly random in $\mathcal{R}_{q_i}^{r\times r}$.

- $\texttt{ModRNS.Enc}_{\texttt{pk}}(m)$: We obtain a ciphertext, $\texttt{ct} = (\texttt{ct}^{(j)})_{0\leq j\leq L}$ such that each $\texttt{ct}^{(j)} = (\texttt{pk}^{(j)} \cdot \mathbf{v} + (m+e, \mathbf{e}')) \pmod{q_j} \in \mathcal{R}_{q_j} \times \mathcal{R}_{q_j}^r$ for $0 \leq j \leq L$.

- $\texttt{ModRNS.Dec}_{\texttt{sk}}(\texttt{ct})$: For a ciphertext $\texttt{ct} = (\texttt{ct}^{(j)})_{0\leq j\leq l}$ retrieve an approximation of the message m under the secret key $\texttt{sk}$, $\langle \texttt{ct}^{(0)}, \texttt{sk}\rangle \pmod{q_0} \approx m$.

- $\texttt{ModRNS.KeySwitch}_{\texttt{swk}}(\texttt{ct})$: For a ciphertext $\texttt{ct} = (\texttt{ct}^{(j)})_{0\leq j\leq l}$, output the ciphertext, $\texttt{ct}' = (\texttt{ct}'^{(j)})_{0\leq j\leq l}$ such that each of its residue component is given by, $\texttt{ct}'^{(j)} = \big((c_0^{(j)}, 0) + (\texttt{RingRNS.ModDown}(\mathbf{c_1'}^{(j)})\big) \pmod{q_j} \in \mathcal{R}_{q_j} \times \mathcal{R}_{q_j}^r$, where, $\mathbf{c_1'}^{(j)} = \sum_{i=0}^{l+1} (\mathbf{c_1}^{(j)} \cdot (\texttt{swk}^{(j,i)}[0], \texttt{swk}^{(j,i)}[1]) (\bmod q_i)) \in \mathcal{R}_{pQ_l} \times \mathcal{R}_{pQ_l}^r$.

- $\texttt{ModRNS.Add}(\texttt{ct}, \texttt{ct}')$: Given two ciphertexts $\texttt{ct} = (\texttt{ct}^{(j)})_{0\leq j\leq l}$ and $\texttt{ct}' = (\texttt{ct}'^{(j)})_{0\leq j\leq l}$ both in $\prod_{j=0}^{l} \mathcal{R}_{q_j}^{r+1}$ at some arbitrary level l, their sum is given by, $\texttt{ct}_{\texttt{add}} = (\texttt{ct}_{\texttt{add}}{}^{(j)})_{0\leq j\leq l} \in \prod_{j=0}^{l} \mathcal{R}_{q_j}^{r+1}$ where $\texttt{ct}_{\texttt{add}}{}^{(j)} = \texttt{ct}^{(j)} + \texttt{ct}'^{(j)} \pmod{q_j}, 0 \leq j \leq l$ following the algorithm Mod.Add.

- $\texttt{ModRNS.Mult}(\texttt{ct}, \texttt{ct}')$: For ciphertexts $\texttt{ct} = \big(\texttt{ct}^{(j)} = c_0^{(j)}, \mathbf{c_1}^{(j)}\big)_{0\leq j\leq l}$ and $\texttt{ct}' = \big(\texttt{ct}'^{(j)} = c_0'^{(j)}, \mathbf{c_1'}^{(j)}\big)_{0\leq j\leq l}$ both in $\prod_{j=0}^{l} \mathcal{R}_{q_j}^{r+1}$, we write each residue of the result

$\mathtt{ct_{mult}} = \mathtt{d}$ of $\mathtt{Mod.Mult}$ in the RNS form as $(\mathtt{d}^{(j)} = d_0^{(j)}, \mathbf{d_1}^{(j)}, \mathbf{d_2}^{(j)}, \mathbf{d_3}^{(j)} \in \mathcal{R}_{q_j} \times \mathcal{R}_{q_j}^r \times \mathcal{R}_{q_j}^r \times \mathcal{R}_{q_j}^{r(r-1)/2}),$

$$d_0^{(j)} = c_0^{(j)} \cdot {c'_0}^{(j)} \qquad d_{1t}^{(j)} = c_0^{(j)} \cdot {c'_{1t}}^{(j)} + {c'_0}^{(j)} \cdot c_{1t}^{(j)}$$
$$d_{2t}^{(j)} = c_{1t}^{(j)} \cdot {c'_{1t}}^{(j)} \qquad d_{3_{tt'}}^{(j)} = c_{1t}^{(j)} \cdot {c'_{1t'}}^{(j)} + {c'_{1t}}^{(j)} \cdot {c_{1t'}}^{(j)},\ t < t', 0 \le t, t' < r$$

with all arithmetic done $\pmod{q_j}$, for $0 \le j \le l$. For each j-th residue component of $\mathbf{d_3}$, the notation $d_{3_{tt'}}^{(j)}$ refers to the component that is obtained as a result of multiplications between the '$1t$'-th component and the '$1t'$'-th component of the ciphertexts $\mathtt{ct}$ and $\mathtt{ct}'$ with respect to each j-th residue.

- $\mathtt{ModRNS.Relin}_{evk}(\mathtt{ct_{mult}})$: We need to relinearize the module components $(\mathbf{d_2}^{(j)})_{0\le j\le l}$ and $(\mathbf{d_3}^{(j)})_{0\le j\le l}$. The evaluation keys are given by, $\mathtt{evk}_{d_2} = \mathtt{ModRNS.KSGen}(\mathbf{s}, \mathbf{s^2})$ and $\mathtt{evk}_{d_3} = \mathtt{ModRNS.KSGen}(\mathbf{s}, \mathbf{s}')$ where $\mathbf{s}' = (s_i s_j)_{0\le i,j<r; j>1}$. As in $\mathtt{RingRNS.Relin}$ and in $\mathtt{Mod.Relin}$ we perform an equivalent of the 'modulus up' operation of each $\mathbf{d_2}^{(j)}$ and $\mathbf{d_3}^{(j)}$ intrinsically using a prime p. Also, recall from $\mathtt{Mod.Relin}$ that the evaluation key, $\mathtt{evk}$ is in the modulo domain of $P \cdot q_L$. In RNS representation we denote each of its residues is given by $\mathtt{evk}^{(j,i)} = (\mathbf{b}_{evk}^{(j,i)}, \mathbf{A}_{evk}^{(j,i)}) \in \mathcal{R}_{q_i}^r \times \mathcal{R}_{q_i}^{r\times r}$ for $0 \le j \le L$ and $0 \le i \le L$. Let, $(d''_0)^{(j)} = \sum_{i=0}^{l+1} \big(\mathbf{d_2}^{(j)} \cdot {\mathtt{evk}_{\mathtt{d_2}}}^{(j,i)}[0] + \mathbf{d_3}^{(j)} \cdot {\mathtt{evk}_{\mathtt{d_3}}}^{(j,i)}[0]\big) \pmod{q_i} \in \mathcal{R}_{pQ_l}$ and $(\mathbf{d_1}'')^{(j)} = \sum_{i=0}^{l+1} \big(\mathbf{d_2}^{(j)} \cdot {\mathtt{evk}_{\mathtt{d_2}}}^{(j,i)}[1] + \mathbf{d_3}^{(j)} \cdot {\mathtt{evk}_{\mathtt{d_3}}}^{(j,i)}[1]\big) \in \mathcal{R}_{pQ_l}^r$. The relinearized ciphertext is then $\mathtt{ct_{relin}}$ such that each of its residue shares is given by, $\big(\mathtt{ct_{relin}}^{(j)} = ({d'_0}^{(j)}, {\mathbf{d'_1}}^{(j)})\big)_{0\le j\le l}$ such that ${d'_0}^{(j)} = (d_0^{(j)} + \mathtt{RingRNS.ModDown}\big((d''_0)^{(j)}\big)) \pmod{q_j} \in \mathcal{R}_{q_j}$ and ${\mathbf{d'_1}}^{(j)} = (\mathbf{d_1}^{(j)} + \mathtt{RingRNS.ModDown}\big((\mathbf{d_1}'')^{(j)}\big)) \pmod{q_j} \in \mathcal{R}_{q_j}^r$ for $0 \le j \le l$.

- $\mathtt{ModRNS.Rescale}(\mathtt{ct})$: A ciphertext $\mathtt{ct} = (\mathtt{ct}^{(j)})_{0\le j\le l} \in \prod_{j=0}^{l} \mathcal{R}_{q_j} \times \prod_{j=0}^{l} \mathcal{R}_{q_j}^r$ is changed into a ciphertext $\mathtt{ct}' = ({\mathtt{ct}'}^{(j)})_{0\le j\le (l-1)} = \big(q_l^{-1} \cdot (c_0^{(j)} - c_0^{(l)}), q_l^{-1} \cdot (\mathbf{c_1}^{(j)} - \mathbf{c_1}^{(l)}) \pmod{q_j}\big)_{0\le j\le (l-1)} \in \prod_{j=0}^{l-1} \mathcal{R}_{q_j} \times \prod_{j=0}^{l-1} \mathcal{R}_{q_j}^r$.

  Lastly, we also provide the RNS version of our proposed rank-reduction technique.

- $\mathtt{ModRNS.RedKeySwitch}(\mathbf{s}', \mathbf{s}'')$: In RNS representation we denote the residues of the reduction key for each $0 \le j < l$ and $0 \le i' \le l$ b,y $\mathtt{redk}^{(j,i')} = (\mathbf{b}_{red}^{(j,i')} = -\mathbf{A}_{red}^{(j,i')} \cdot \mathbf{s}' + \mathbf{e}_{red} + \mathbf{s}'', \mathbf{A}_{red}^{(j,i')}) \in \mathcal{R}_{q_{i'}}^i \times \mathcal{R}_{q_{i'}}^{i\times r'}$.

- $\mathtt{ModRNS.Rankred}_{\mathtt{redk}}(\mathtt{ct})$: The residues corresponding to the rank-reduced ciphertext are written as, $({\mathtt{ct}'}^{(j)} = ({c'_0}^{(j)}, {\mathbf{c'_1}}^{(j)}))_{0\le j\le l}$ such that the j-th residue is given by, $({c'_0}^{(j)}, {\mathbf{c'_1}}^{(j)}) = \big((c_0^{(j)}, \mathbf{c}_{\mathrm{red}}^{(j)}) + \mathtt{RingRNS.ModDown}(\mathbf{c}''^{(j)})\big) \pmod{q_j} \in \mathcal{R}_{q_j} \times \mathcal{R}_{q_j}^{r'}$, such that $\mathbf{c}''^{(j)} = \sum_{i'=0}^{l} (\mathbf{c}_{\mathrm{rem}}^{(j)} \cdot (\mathtt{redk}^{(j,i')}[0], \mathtt{redk}^{(j,i')}[1]) \pmod{q_j} \in \mathcal{R}_{pQ_l} \times \mathcal{R}_{pQ_l}^{r'}$.

Algorithm 5 $\mathtt{ModRNS.Add}$ Algorithm

In: $\boldsymbol{ct} = (\tilde{c}_0, \tilde{\mathbf{c}}_1)$, $\boldsymbol{ct'} = (\tilde{c}'_0, \tilde{\mathbf{c}}'_1) \in R_{Q_l}^{r+1}$
Out: $\boldsymbol{d} = (\tilde{d}_0, \tilde{\boldsymbol{d}}_1) \in R_{Q_l}^{r+1}$

1: $\tilde{d}_0 \leftarrow \tilde{c}_0 + \tilde{c}'_0$
2: **for** $i = 0$ to $r - 1$ **do**
3: $\quad \tilde{d}_1[i] \leftarrow \tilde{c}_1[i] + \tilde{c}'_1[i]$
4: **end for**

Algorithm 6 `ModRNS.Mult` Algorithm

In: $\boldsymbol{ct} = (\tilde{c}_0, \tilde{\boldsymbol{c}}_1)$, $\boldsymbol{ct'} = (\tilde{c}'_0, \tilde{\boldsymbol{c}}'_1) \in R_{Q_l}^{r+1}$

Out: $\boldsymbol{d} = (\tilde{d}_0, \tilde{\boldsymbol{d}}_1, \tilde{\boldsymbol{d}}_2, \tilde{\boldsymbol{d}}_3) \in R_{Q_l} \times R_{Q_l}^r \times R_{Q_l}^r \times R_{Q_l}^{r(r-1)/2}$

1: $\tilde{d}_0 \leftarrow \tilde{c}_0 \star \tilde{c}'_0$
2: **for** $i = 0$ to $r-1$ **do**
3: $\quad \tilde{d}_1[i] \leftarrow \tilde{c}_0[i] \star \tilde{c}'_1[i] + \tilde{c}_1[i] \star \tilde{c}'_0[i]$
4: $\quad \tilde{d}_2[i] \leftarrow \tilde{c}_1[i] \star \tilde{c}'_1[i]$
5: $\quad$ **for** $j = 1$ to $r-1$ **do**
6: $\quad\quad$ **if** $i < j$ **then**
7: $\quad\quad\quad \tilde{d}_3[i,j] \leftarrow \tilde{c}_1[i] \star \tilde{c}'_1[j] + \tilde{c}_1[j] \star \tilde{c}'_1[i]$
8: $\quad\quad$ **end if**
9: $\quad$ **end for**
10: **end for**

Algorithm 7 `ModRNS.SubRelin` Algorithm

In: $\acute{\boldsymbol{d}} \in R_{Q_l}^r$

In: $\tilde{\mathtt{evk}}_0 \in R_{pQ_L}^{r \cdot L}, \tilde{\mathtt{evk}}_1 \in R_{pQ_L}^{r \cdot r \cdot L}$

Out: $\boldsymbol{d} = (\acute{d}_0, \acute{\boldsymbol{d}}_1) \in R_{Q_L}^{r+1}$

1: **for** $k = 0$ to $r-1$ **do**
2: $\quad$ **for** $j = 0$ to $l+1$ **do** $\quad$ ▷ Here q_{l+1} is used to represent special prime p
3: $\quad\quad$ **for** $i = 0$ to l **do**
4: $\quad\quad\quad \tilde{\boldsymbol{u}} \leftarrow \mathtt{NTT}(\mathtt{INTT}([\acute{\boldsymbol{d}}[i][k]])_{q_j})$ $\quad$ ▷ in $\mathbb{Z}_{q_j}$
5: $\quad\quad\quad \acute{d}_0 \leftarrow \acute{d}_0 + \tilde{\mathtt{evk}}_0[i][j][k] \star \tilde{\boldsymbol{u}}_{q_j}$
6: $\quad\quad\quad \acute{d}_1[j,k] \leftarrow \sum_{t=0}^{r-1} \tilde{\mathtt{evk}}_1[i][j][k][t] \star \tilde{\boldsymbol{u}}_{q_j}$
7: $\quad\quad$ **end for**
8: $\quad$ **end for**
9: **end for**

4 Potentials and limitations of MLWE-based homomorphic encryption

We devote this section to identify the potentials of an MLWE-based homomorphic encryption scheme as well as discuss the possible drawbacks that would need further attention from the community. We discuss enhanced security, hardware reusability, increased opportunities for parallel computation and the flexibility to change module rank in the list of potentials. On the other hand, constrained message packing ability and reduced efficiency pose a challenge to the scheme.

4.1 Better security assumptions

While RLWE is known to be hard over restrictions of special classes of ideal lattices, the LWE problem is considered to be as hard as worst-case problems on Euclidean lattices [AD17]. MLWE acts as a bridge between the LWE and RLWE problems. In fact, the hardness of MLWE can be considered equivalent to natural hard problems of lattices and most attacks need to view this problem in terms of the LWE problem (by replacing the ring $\mathcal{R}_q$ by $\mathbb{Z}_q$). Although the MLWE problem is defined over polynomial rings, MLWE would still remain secure under an attack exploiting RLWE. As a result, MLWE can provide a better guarantee of security. The MLWE problem has also been proposed as an alternative to protect against attacks that target the structural characteristics of RLWE [CDW17].

Algorithm 8 `ModRNS.Relin` Algorithm

In: $\boldsymbol{d} = (\tilde{d}_0, \tilde{\boldsymbol{d}}_1, \tilde{\boldsymbol{d}}_2, \tilde{\boldsymbol{d}}_3) \in R_{Q_l} \times R_{Q_l}^r \times R_{Q_l}^r \times R_{Q_l}^{r(r-1)/2}$
In: $\tilde{\texttt{evk}}_0 \in R_{pQ_L}^{(r\cdot(r+1)\cdot L)/2}, \tilde{\texttt{evk}}_1 \in R_{pQ_L}^{(r\cdot r\cdot(r+1)\cdot L)/2}$
Out: $\boldsymbol{d}' = (\tilde{d}'_0, \tilde{\boldsymbol{d}}'_1) \in R_{Q_l}^{r+1}$
1: $t_0, \boldsymbol{t_1} \leftarrow 0$ ▷ $(\boldsymbol{t_0}, \boldsymbol{t_1}) \in R_{pQ_l}^{((r+1)\cdot l)/2} \times R_{pQ_l}^{(r\cdot(r+1)\cdot l)/2}$
2: $(t_0[0], \boldsymbol{t_1}[0]) \leftarrow$ `ModRNS.SubRelin`$(\tilde{\boldsymbol{d}}_2, \tilde{\texttt{evk}}_0[0], \tilde{\texttt{evk}}_1[1])$
3: **for** $j = 1$ to $(r+1)/2$ **do**
4: $\quad (t_0[j], \boldsymbol{t_1}[j]) \leftarrow$ `ModRNS.SubRelin`$(\tilde{\boldsymbol{d}}_3[j-1], \tilde{\texttt{evk}}_0[j], \tilde{\texttt{evk}}_1[j])$
5: **end for**
6: $\tilde{\boldsymbol{d}}'_0 \leftarrow \sum_{j=0}^{(r+1)/2} t_0[j] +$ `RingRNS.ModDown`$(\tilde{\boldsymbol{c}}''_0)$, $\tilde{\boldsymbol{d}}'_1 \leftarrow \sum_{j=0}^{(r+1)/2} \boldsymbol{t}_1[j] +$ `RingRNS.ModDown`$(\tilde{\boldsymbol{c}}''_1)$

Algorithm 9 `ModRNS.RedKeySwitch` Algorithm

In: $\acute{\boldsymbol{c}} \in R_{Q_l}^i$
In: $\tilde{\texttt{redk}}_0 \in R_{pQ_l}^{i\cdot l}, \tilde{\texttt{redk}}_1 \in R_{pQ_l}^{i\cdot r'\cdot l}$ ▷ $r' = r - i$
Out: $(\acute{c}_0, \acute{\boldsymbol{c}}_1) \in R_{Q_l} \times R_{Q_l}^{r'}$
1: **for** $k = r'+1$ to r **do**
2: $\quad$ **for** $j = 0$ to $l+1$ **do** ▷ Here q_{l+1} is used to represent special prime p
3: $\quad\quad$ **for** $m = 0$ to l **do**
4: $\quad\quad\quad \tilde{\boldsymbol{u}} \leftarrow$ `NTT`(`INTT`$(\left[\acute{\boldsymbol{c}}[m][k]\right])_{q_j})$ ▷ in $\mathbb{Z}_{q_j}$
5: $\quad\quad\quad \acute{c}_0 \leftarrow \acute{c}_0 + \tilde{\texttt{redk}}_0[m][j][k] \star \tilde{\boldsymbol{u}}_{q_j}$
6: $\quad\quad\quad \acute{c}_1[j,k] \leftarrow \sum_{t=0}^{r'-1} \tilde{\texttt{redk}}_1[m][j][k][t] \star \tilde{\boldsymbol{u}}_{q_j}$
7: $\quad\quad$ **end for**
8: $\quad$ **end for**
9: **end for**

4.2 Hardware-reusability

RLWE-based homomorphic encryption schemes set their ring dimensions based on the desired level of security and the multiplicative depth. For example, CKKS [CKKS17] sets the degree of polynomial to 2^{14} or 2^{15} for the multiplicative depths 6 and 10 respectively. Note that, an increase in the multiplicative depth increases the ciphertext modulus size which results in diminished security. To compensate for the security loss, the dimension of the lattice is increased by increasing the polynomial degree. The polynomial size changes by a factor of two for different multiplicative depths when cyclotomic rings are used.

Such variations in the polynomial degree in the RLWE-based homomorphic encryption scheme make hardware implementation of a parameter-flexible architecture challenging. Furthermore, when the polynomial degree is 2^{13} or larger, implementation of the large accelerator circuit in hardware becomes very challenging due to low resource utilization, placement, routing, slow clock frequency, and limited on-chip memory, as discussed in [RLPD20, MAK+23].

MLWE-based homomorphic encryption scheme avoids these problems due to its modular nature. This is explained as follows. When working with modules, we can choose a base ring of some suitable (and small) dimension, say $N = 2^k$, and then we can adjust the rank r of the module as per our requirements. This is a good way to restrain the degree of the polynomial from increasing every time we want more depth.

Algorithm 10 `ModRNS.RankRed` Algorithm

In: $\boldsymbol{ct} = (\tilde{c}_0, \tilde{\boldsymbol{c}}_1) \in R_{Q_l} \times R^r_{Q_l}$

In: $\tilde{\texttt{redk}}_0 \in R^{(i \cdot l)}_{pQ_l}, \tilde{\texttt{redk}}_1 \in R^{(i \cdot r' \cdot l)}_{pQ_l}, \tilde{\boldsymbol{c}}_{rem} \in R^i_{Q_l}$ ▷ $r' = r - i$

Out: $\boldsymbol{ct}' = (\tilde{c}'_0, \tilde{\boldsymbol{c}}'_1) \in R_{Q_l} \times R^{r'}_{Q_l}$

1: $t_0, \boldsymbol{t_1} \leftarrow 0$ ▷ $(t_0, \boldsymbol{t_1}) \in R^l_{pQ_l} \times R^{(r' \cdot l)}_{pQ_l}$
2: **for** $j = 1$ to r' **do**
3: $\quad (t_0, \boldsymbol{t_1}[j]) \leftarrow$ `ModRNS.RedKeySwitch`$(\tilde{\boldsymbol{c}}_{rem}[j], \tilde{\texttt{Evk}}_0[j], \tilde{\texttt{Evk}}_1[j])$
4: **end for**
5: $\tilde{c}'_0 c_0 +$ `RingRNS.ModDown`$(\tilde{t}_0)$, $\tilde{\boldsymbol{c}}'_1 \leftarrow \sum_{j=0}^{(r'-1)} \big(\tilde{\boldsymbol{c}}_1[j] +$ `RingRNS.ModDown`$(\tilde{\boldsymbol{t}}_1[j])\big)$

4.3 Increased scope for parallel computations

The proposed MLWE-based homomorphic encryption scheme adds an additional level of abstraction with respect to RLWE-based schemes. A module element is a vector of several small ring components, and these small ring components could be processed in parallel. This additional and high-level parallelism could be exploited in the cloud where many accelerator platforms are generally available. Let us consider the relinearization operation of our `ModRNS` construction where $(\boldsymbol{d}_2, \boldsymbol{d}_3) \in R^r_{Q_l} \times R^{r(r-1)/2}_{Q_l}$ components of the extended ciphertext are linearized (section 3). The relinearization boils down to relinearization of the polynomial ring elements of $\boldsymbol{d}_2$ and $\boldsymbol{d}_3$. As there are no data dependencies between them, the polynomial ring elements could be processed in parallel. As an example, Fig. 1 describes relinearization when rank $r = 2$. Here, $\boldsymbol{d}_2$ has two ring elements, and $\boldsymbol{d}_3$ has only one ring element. All of these ring elements are data and computation-independent of each other throughout the relinearization until the very end when accumulation is performed. Thus, in a multi-accelerator setting (which is common in clouds), dedicated accelerators can be used to process the three ring elements independently in parallel (Fig. 1). We observe that until the very end of the multiplication and key-switching, no accelerator-to-accelerator data exchanges are required. This is particularly advantageous as frequent inter-accelerator data exchanges (which happens when there are data dependencies) are problematic for parallel processing. The same applies to multi-threaded or multi-processor software systems.

4.4 Ciphertext compression due to module rank reduction

Module-based homomorphic encryption scheme offers additional flexibility in determining the dimension of the lattice problem to work with. This also let us adapt the module rank depending on the size of the ciphertext modulus. In a leveled homomorphic encryption scheme, the ciphertext modulus is initially the largest and then it gradually reduces with the multiplicative depth. Therefore, we may scale down the lattice problem from the initial dimension $N \cdot r$ to a smaller $N \cdot r'$ where $r > r'$ while keeping the security of the scheme in place. Smaller ciphertexts would also mean a reduced decryption complexity, lower computation overhead, and smaller key size.

The procedure for rank-reduction was described in section 3. Although the procedure is a form of key-switching, it is computationally cheaper than a full rank key-switching (e.g., relinearization or rotation).

Rank reduction could be advantageous in applications where majority of homomorphic calculations are performed on ciphertexts of low depths, and fewer calculations are performed ciphertexts of high depths. We give an example of Hybrid Homomorphic Encryption (HHE) [DGH+21]. In HHE the client encrypts her data using a symmetric cipher before sending them to the cloud. HHE saves the communication bandwidth significantly compared to the case where the data is encrypted using a homomorphic encryption scheme.

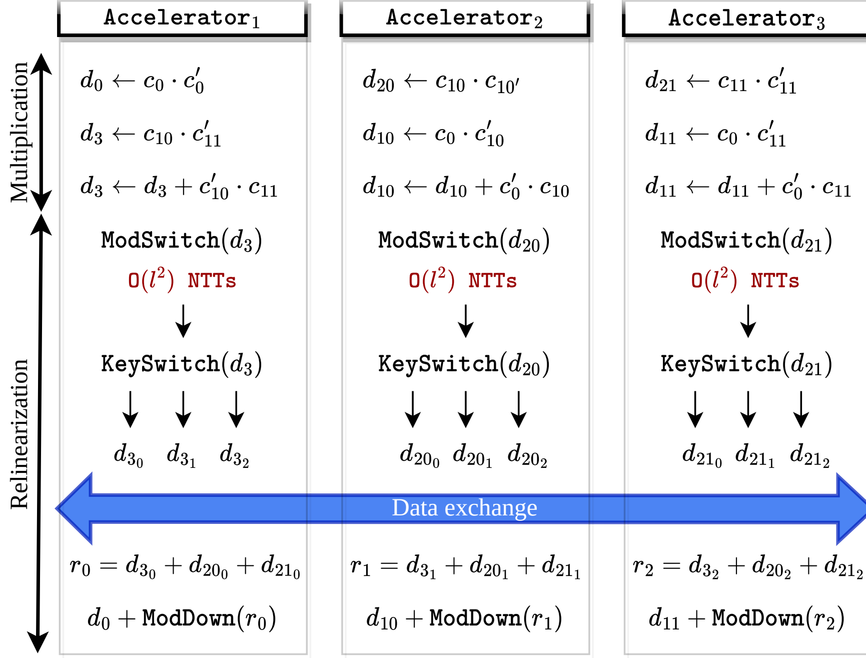

Figure 1: An operation schedule of the relinearization step demonstrating massive parallelism and limited communication for module-dimension $r = 2$

The client also sends a homomorphic encryption of her symmetric key to the cloud. The cloud then evaluates the decryption operation of the symmetric cipher homomorphically to obtain homomorphic encryption of user's data. After that, the cloud is able to evaluate any useful analytical or statistical function on the user's data. A typical scenario in HHE is that the server requires larger parameter sets that can support greater multiplicative depth so as to be able to perform the homomorphic decryption and still be left with enough multiplicative depth to carry out the actual computations. In such a case, ModHE endowed with the rank reduction feature can be well-suited. This can be explained in a better way. Let, the symmetric decryption requires L_{dec} multiplicative levels and the homomorphic analytical or statistical function evaluation requires L_{eval} levels, then the parameter set of the HHE must be chosen to support a total of $L \geq L_{dec} + L_{eval}$ multiplicative levels. With ModHE, the cloud fixes a ring degree N and chooses a module rank r suitable for $L_{dec} + L_{eval}$. The cloud uses L_{dec} for the symmetric decryption and then performs a rank reduction sufficient for L_{eval}. For the special case $L_{dec} \approx L_{eval}$, the cloud can choose a rank $r = 2$, hence perform *dec* in module lattice and *eval* in ring lattice ($r = 1$).

4.5 Reduced message packing

As previously mentioned, the packing of plaintext messages does not increase with an increase in the module rank so `ModRNS` can encode a plaintext of size $N \cdot \log(q_i)/2$ bits as opposed to $N' \cdot \log(q_i)/2$ bits in RNS-CKKS where, $N' = N \cdot r$. By construction, fixing N also fixes the degree of the first polynomial component c_0 in the `ModRNS` ciphertext. This implies that the degree of the message polynomial m that encodes the $N/2$ complex numbers also gets set to N since it resides in the c_0 component. This degree does not increase with an increase in the rank of the module and hence, the amount of plaintext a `ModRNS` ciphertext can encode decreases linearly in r, when compared to `RingRNS`. `ModRNS` trades flexibility of message packing with flexibility of adjusting the module rank in

accordance with security parameters. Indeed, the effect of this constraint would be less palpable in applications that have sparsely packed messages but still require larger security parameters.

4.6 Increased memory consumption and reduced performance

Another limitation of using `ModRNS` is that the number of (small) ring-level operations as well as the number of polynomials in the keys increases quadratically with the module dimension r. More generally, with the module rank r, there will be $O(r^2)$ non-linear components of polynomial degree N to relinearize after a polynomial multiplication. In contrast, in the `RingRNS` case, only one non-linear component of polynomial degree $N \cdot r$ must be relinearized.

Understandably, the expansion rate of the MLWE ciphertext is $(r+1)/2$ as the size of the `ModRNS` ciphertext is $L \cdot N \cdot (r+1) \cdot \log(q_i)$ bits, while the size of the RLWE ciphertext is $2L \cdot N' \cdot \log(q_i)$ bits where $N' = N \cdot r$. The proposed MLWE-based scheme performs more polynomial operations than the corresponding RLWE-based scheme for the same parameter set. Although the number of polynomials is higher in `ModRNS` than in `RingRNS`, the polynomials in `ModRNS` are smaller by r times than those in `RingRNS`. Therefore, choosing a too-small ring size and a very large module rank will not be an ideal design decision for a homomorphic encryption scheme.

Next, to understand how the rank r of the module affects the total number of keys, we recall that for the secret vector $\mathbf{s} = \{s_0, \cdots, s_{r-1}\}$, we require relinearization keys containing the product of every two of its components, that is, $\sum_{i,j=0}^{r-1} s_i s_j$ but with terms of the form $s_i s_j$ and $s_j s_i$ for a pair of (i, j) absorbed into the same relinearization key. Thus the total number of keys would be $\frac{r(r+1)}{2}$ relinearization keys with respect to each distinct quadratic secret component along with the usual r decryption keys. The size of relinearization keys in `ModRNS` is equal to $r \cdot (r+1)^2 \cdot L \cdot (L+1) \cdot N \cdot \log(q_i)/2$ bits when compared to $2 \cdot L \cdot (L+1) \cdot N' \cdot \log(q_i)$ bits in the RLWE setting. Similarly, the rank-reduction keys of `ModRNS` have a size of $(r'+1) \cdot L' \cdot (L'+1) \cdot N \cdot \log q_i$ for all $r' < r$ and $L' < L$.

As stated earlier in the text, the structured variants of LWE over polynomials rings (RLWE) provide advantages in terms of efficiency in speed and keys as well as ciphertext sizes. However, the additional structure makes them susceptible to attacks. On the other hand, LWE benefits from lack of structure but suffers from substantially decreased efficiency. MLWE interpolates between the LWE and RLWE problems [SAB+21]. For similar parameters, MLWE would have a reduced structure compared to its RLWE counterpart (benefiting security) but reduced performance, which we experience in our construction as well.

4.7 Higher precision loss due to increased error

Due to increased computations for relinearization and rank reduction, the error also increases in `ModRNS` as discussed in section 3.1. For `RingRNS`, no rank reduction exists so there is no possibility for comparison. In case of the relinearization step, the number of non-linearized components in `ModRNS` is directly proportional to the module rank r in contrast with `RingRNS` in which it is always one. As mentioned in section 3.1 the error can be quantified with a factor of $r \cdot (r+1)/2$ in `ModRNS`. Thus, we also report an increased precision loss with respect to `RingRNS` atleast by $r \cdot (r+1)/2\times$. This is a trade-off that needs to be performed based on the demands of the specific application. Applications requiring higher precision can choose a bigger word size at the expense of some multiplicative depth L.

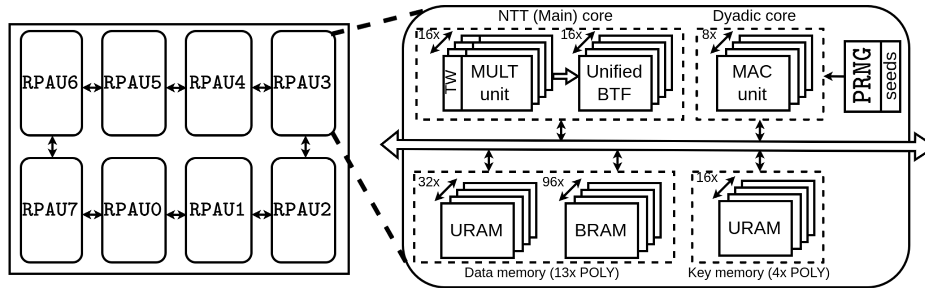

Figure 2: High-level architecture of the proposed design which follows the ring-based interconnect proposed in [MAK+23]. Each RPAU has a 16-core NTT coupled with a 16-core twiddle factor generator, an 8-core dyadic unit, and on-chip BRAM/URAMs.

5 Hardware architecture for ModRNS and its evaluation

In this paper, along with a Sage implementation, we define a proof-of-concept instruction-set hardware architecture for the proposed ModRNS scheme.The architecture is coded using Verilog RTL and implemented using the Xilinx Vivado tool (up to placement and routing) for the Xilinx Alveo U280 accelerator card. While area results of the architecture are obtained from Vivado, computation times for high-level homomorphic evaluation procedures and applications are estimated from the number of instructions.

We do not develop a full-stack prototype of the hardware accelerator as we believe the proposed MLWE-based homomorphic encryption scheme is in its early stages of development. Building a prototype of a homomorphic accelerator requires many months (or years) of rigorous engineering. Hence, it would be more practical to create an accelerator prototype once MLWE-based homomorphic encryption is further improved. The majority of published hardware acceleration studies for homomorphic encryption report simulation results rather than prototypes.

5.1 Hardware architecture

Intuitively, the proposed MLWE-based homomorphic encryption scheme ModRNS adds an additional layer of abstraction with respect to the RLWE-based RingRNS scheme: a module element is a collection of ring elements. Hence, RLWE-based and MLWE-based homomorphic encryption schemes share very similar homomorphic routines and arithmetic operations. For example, the relinearization operation of RingRNS (Algorithm 4) is very similar to the relinearization subroutine of ModRNS (Algorithm 7). Both operations include NTT/INTT and multiplication with evaluation keys. The ModRNS performs the ring-relinearization subroutine several times to perform the module-relinearization operation as shown in Algorithm 8. Thus, a hardware architecture designed for an RLWE-based homomorphic encryption scheme can be re-used for an MLWE-based scheme with very few changes. For example, any accelerator architecture for the RingRNS scheme can be adapted to accelerate the homomorphic operations of the ModRNS. To that end, we followed the design approach of an existing instruction-based accelerator architecture, Medha [MAK+23], and showcase how with minimal changes it can efficiently support the ModRNS scheme.

Target parameter set and platform: As a proof of concept, we selected polynomial degree N as 2^{14}, module dimension r as 2, and the number of small RNS primes as 15 where prime sizes are 54-bit and 60-bit. With the chosen parameters ($N = 2^{14},\ r = 2,\ \log_2 q_i = 54,\ L = 15$), encryption of $\frac{N \cdot \log q_i}{2}$ bits ≈ 0.05 MB of plaintext using ModRNS produces a ciphertext of size $L \cdot N \cdot (r+1) \cdot \log q_i$ bits ≈ 5 MB. These parameters support a multiplicative depth of $L - 1$, so in our case, it is 14. The relinearization keys will

require $\frac{r\cdot(r+1)^2\cdot L\cdot(L+1)\cdot N\cdot\log q_i}{2}$ bits $\approx$ 239 MB of storage. The keys for rank reduction for going from $r = 2$ to $r' = 1$ ($\texttt{ct} \in \mathcal{R}_q \times \mathcal{R}_q^2 \to \texttt{ct}' \in \mathcal{R}_q \times \mathcal{R}_q^1$), when L reduces to $L' = 7$, require $\frac{(r'+1)\cdot L'\cdot(L'+1)\cdot N\cdot\log q_i}{2}$ bits $\approx$ 16 MB of storage. A `RingRNS` ciphertext for the same security parameters consumes $2 \cdot L \cdot N \cdot r \cdot \log q_i$ bits $\approx$ 6.6 MB storage, can encode $\frac{N\cdot r\cdot\log q_i}{2}$ bits $\approx$ 0.1 MB data and the relinearization key size required in this case is $2 \cdot L \cdot (L+1) \cdot N \cdot r \cdot \log q_i$ bits $\approx$ 106 MB. The additional memory requirement for keys in `ModRNS` compared to `RingRNS` is not a problem as present-day accelerator platforms contain on-board DDR or high bandwidth memory (HBM).

As explained in section 4, one limitation of `ModRNS` is that the key size can increase with large module dimensions. Thus, using only on-chip memory resources for storing the evaluation keys can become more challenging for large module dimensions. To that end, for our proof-of-concept design, we selected the Xilinx Alveo U280 card, equipped with an HBM providing up to 460 GB/s, as our target platform.

Low-level arithmetic units: Design decisions for the low-level arithmetic units are borrowed from Medha [MAK+23]. Reader may follow [MAK+23] for detailed description of these units. We used Xilinx multiplier IPs to generate bit-parallel multipliers that utilize DSPs available in Alveo U280 card for implementing integer multiplication operations. An integer multiplier for 54/60-bit inputs consumes 10 DSP units and has a latency of 12 cycles. To simplify modular reduction operation, we used sparse primes that also enable cheap reduction circuit implementations. Instead of using Barrett and Montgomery methods that work for generic primes, we used an add-shift-based modular reduction approach optimized for sparse primes that uses only LUTs and eliminates DSPs [ZYC+20, MAK+23]. An add-shift-based modular reduction unit can support 2 sparse primes, it is fully pipelined and finishes one operation in 5 cycles.

NTT is one of the fundamental and most time-consuming arithmetic blocks in HE schemes. Thus, there are several efforts in literature to implement efficient NTT architectures for HE [MAK+23, RLPD20]. In this work, we implemented a parallel 16-core iterative NTT circuit for $N = 2^{14}$. Each core implements a unified butterfly circuit that can perform both the Cooley-Tukey butterfly configuration for NTT and the Gentleman-Sande butterfly configuration for INTT [CG99]. The same circuit can perform both NTT and INTT, and it finishes one NTT/INTT operation in 7,168 cycles. A butterfly operation during NTT requires several constants called *twiddle factor*. Storing all necessary twiddle factors for each butterfly unit will increase on-chip memory utilization significantly. To reduce the on-chip memory usage, we generated twiddle factors on-the-fly during NTT/INTT operation. Each butterfly unit is coupled with one modular multiplier and a memory block that stores initial twiddle factors. These extra multipliers used in twiddle factor generation are re-used for coefficient-wise multiplication operations as well.

Architecture of `ModRNS` processor: Here we describe how an existing accelerator architecture (in this case Medha [MAK+23]) for `RingRNS` could be turned into an accelerator for the proposed `ModRNS` by incorporating small modifications. Medha [MAK+23] designs and implements residue polynomial arithmetic units (RPAU) of polynomial size 2^{14} for each small RNS prime used in the `RingRNS` scheme. Each RPAU has an NTT (main) unit for NTT/INTT and coefficient-wise operations, a dyadic unit tailored for coefficient-wise polynomial addition and multiplication, and on-chip memory for data and key storage. We followed a similar approach and implemented RPAUs for RNS bases of `ModRNS`. In an RPAU, the NTT and dyadic units can run operations in parallel and this enables efficient execution of the relinearization operation of `ModRNS`. The relinearization operation requires one NTT and $r \cdot (r+1)/2$ evaluation key multiplications as explained in section 4.4. For example, for $r = 2$, it requires three evaluation key multiplications. For optimal performance, the latency of one NTT operation and $r \cdot (r+1)/2$ evaluation key multiplications should be similar so the

execution of both operations can be overlapped efficiently. For example, the relinearization operation of `RingRNS` requires one NTT and two evaluation key multiplications (steps 7,8 of Algorithm 4) and [MAK+23] selected the number of cores in NTT unit and dyadic units accordingly to optimize the performance of relinearization operation. Similarly, we used a 16-core NTT unit and an 8-core dyadic unit in each RPAU. This enables our architecture to parallelize one NTT and three evaluation key multiplications efficiently for $r = 2$ setting. To reduce the computation time, we employ 8 RPAUs in our architecture where each RPAU supports two small RNS primes by employing reconfigurable modular reduction units.

Note that if the rank, r, is increased, memory bandwidth, on-chip memory resources and the number of dyadic units are parameters one must re-evaluate to ensure that the hardware architecture performs optimally without any stall due to communication bottleneck. However, this does not necessarily mean that any change in r would require a new architecture. For example, it is possible to use the proposed architecture for a higher r with slight performance loss due to dyadic core latency. This performance loss can be minimized by utilizing the main core for evaluation key multiplications as well. Another approach would be selecting a range of r values to support and optimizing your architecture for the maximum r parameter. This will enable your architecture to operate with multiple r without any performance loss.

Our architecture utilizes HBM and this eliminates the need for excessive on-chip memory consumption. In our architecture, the evaluation keys are assumed to be read from HBM during homomorphic operations without degrading the performance, and fed to the dyadic multipliers for relinearization operation. The host processor sends ciphertexts to the FPGA and ciphertexts are stored in on-chip memory. Our architecture for $r = 2$ uses 8 RPAUs, each having 8-core dyadic units. Thus, during a relinearization operation, $8 \cdot 8 = 64$ coefficient-wise multiplications with 64 coefficients of relinearization keys are performed every cycle. Our architecture needs to read 64 coefficients every cycle from HBM running with an operation frequency of f MHz. It will require an HBM bandwidth of $\approx \frac{64 \cdot 8 \cdot f}{10^3}$ GB/s. For example, for an operating frequency of 210 MHz, the required bandwidth is ≈ 108 GB/s. This shows that Xilinx Alveo U280's HBM is fast enough to feed 64 coefficients of relinearization keys every cycle. Each RPAU in our architecture employs 32 URAMs and 96 BRAMs that can store 13 residue polynomials for ciphertexts. Each RPAU also uses 16 URAMs as a pre-fetch unit that can store evaluation keys read from HBM.

Our target FPGA, Xilinx Alveo U280, consists of three separated SLR regions and there are only a limited number of connections between two neighboring SLRs [Xil]. As shown in Algorithm 4, each RPAU should send or receive residue polynomials from other RPAUs during key switching operation. This poses a significant implementation challenge for SLR-based large FPGAs due to the limited number of SLR-to-SLR connections. In [MAK+23], a *ring* interconnect is proposed to place and connect multiple RPAU units on an SLR-based FPGA. This method connects only neighboring RPAUs and eases the placement and routing of RPAUs. This method is adopted and used in our architecture. We used 2 SLRs to place 8 RPAUs, 4 RPAUs per SLR, and reserved one SLR for HBM-related building blocks for providing a more realistic implementation result. For the rest of the architecture, we followed the same approach as Medha [MAK+23]. A high-level architecture diagram of the proposed accelerator is shown in Fig. 2.

Implementation results: We synthesized and implemented our design targeting the Xilinx Alveo U280 accelerator card using Vivado 2022.2. The implementation results show that our design with 8 RPAU units consumes only 699,513 LUTs, 513,720 FFs, 3,200 DSPs, 1,034 BRAMs, and 305 URAMs. Its implementation achieves a clock frequency of 210 MHz. The implementation includes all components of the proposed processor excluding an HBM

Table 1: Estimated performance figures. When the r is 1, the multiple accelerator setting is not applicable, as we operate on one module component per accelerator.

N	r	**Depth+1** (L)	**Homomorphic Operation**	**Perf. (Cycle)** (1 accelerator)	**Perf. (Cycle)** (3 accelerators)
2^{14}	1	7	Add.	1,152	-
2^{14}	1	7	Mult.	2,560	-
2^{14}	1	7	Mult.+ Relin.	99,448	-
2^{14}	2	15	Add.	3,456	1,152
2^{14}	2	15	Mult.	13,824	4,608
2^{14}	2	15	Mult.+Relin.	1,193,376	397,792
2^{14}	$2 \rightarrow 1$	$7 \rightarrow 7$	Rank reduction	99,448	-

controller. True prototyping in FPGA and functional verification of an HBM-processor interface is not a straightforward task as it could require many months of engineering work [BBTV]. Several works [SYYZ22, SYYT20, FSK+21, KLK+22] in literature report synthesis results. Similar to these works [FSK+21, KLK+22], we use HBM's bandwidth specification for estimating the time requirements for high-level homomorphic procedures and applications.

5.2 Performance benchmarking

Estimated performance results for `ModRNS.Add`, `ModRNS.Mult` and `ModRNS.Relin` operations are presented in Table 1. It should be noted that the proposed accelerator can be used in a multi-FPGA setting where each FPGA implements the accelerator of Fig. 2 to improve the performance as shown in Fig. 1. Cloud providers such as Amazon AWS use FPGA stacks for accelerating computation. Our base ring dimension (when $r = 1$) is 2^{14}, with the packing of 2^{13}, and for higher multiplicative depth, we increase the $r = 2$.

Note that at this point, we have more multiplicative depth; however, the maximum available packing stays fixed at 2^{13}. The fixed amount of packing is one limitation of the `ModRNS` scheme, as we go higher up in the module dimension, the amount of packing does not increase, unlike `RingRNS`. Thus, the `ModRNS` suffers from an $(r+1)/2$ ciphertext expansion rate compared to `RingRNS`. Similarly, owing to quadratic growth in the number of non-linear components after multiplication, `ModRNS` will also require $(r+1)/2\times$ more time compared to an `RingRNS` implementation. Thus, for our example case where $r = 2$, `ModRNS` has 1.5 $\times$ higher ciphertext expansion rate and 1.5$\times$ slower computation. Both `ModRNS` and [MAK+23] use an architecture designed for polynomial degree $N = 2^{14}$. `ModRNS` reuses it for parameters ($r = 2, N = 2^{14}$) and [MAK+23] reuses it to process polynomials of degree $N' = N \cdot r = 2^{15}$. Both these parameters offer the same depth for 128-bit security. Therefore, we just compare the results for [MAK+23] ($N' = 2^{15}$) with `ModRNS` ($r = 2, N = 2^{14}$) on the same architecture. For this case, `ModRNS` consumes $\approx 1.7\times$ more clock cycles compared to [MAK+23].

We also provide an estimated benchmark for the logistic regression inference. For computing the logistic function $g(x) = 1/(1 + e^x)$ homomorphically, we use Taylor's polynomial approximation up to polynomial degree-9 i.e., $g(x) \approx 1/2 + 1/4x - 1/48x^3 + 1/480x^5 - 17/80640x^7 + 31/1451520x^9$. The proposed hardware architecture is estimated to consume 3.8M clock cycles and finish the operation in 18 ms using a three accelerator setting (see Fig. 1) when the FPGA-based accelerator runs at 210 MHz. The same application consumes 1.8 seconds using the SEAL library with polynomial degree $N = 2^{15}$ on one Intel(R) Core(TM) i5-10210U CPU running at 1.60GHz. Although we only provide estimates for logistic regression, similar estimations can be extended to other function approximations, for example, sine, cosine, etc. Comparing the application here with

the one reported in [MAK+23] will not be fair as here we target higher accuracy by expanding to more terms. Instead, comparing the speedup achieved over software can be justified. `ModRNS` running on one hardware accelerator is 2.8× slower than [MAK+23] running on the same hardware accelerator. We also acknowledge that due to added error, `ModRNS` Key-switch operation reports only 34-bit precision compared to the 40-bit precision reported in the `RingRNS` case. The new property of rank reduction also incurs an additional 3-bit precision loss. Thus, `ModRNS` offers ≈ 30-bit precision for the parameters used here. Implementing Medha [MAK+23] in our setting (on Xilinx Alveo U280 card) will significantly reduce its on-chip memory. This might help improve its clock frequency and routing problems to improve performance further. However, we cannot estimate this without performing an actual implementation.

6 Future scopes

In section 4, we discuss a few limitations of an MLWE-based homomorphic encryption. As a future direction of research along ModHE lies in mitigating them. For example, in theory, the canonical embedding map $\tau : \mathcal{K} \to \mathbb{C}^N$ used in the encoding-decoding procedure of RNS-CKKS can be extended to the map $(\tau, \tau, \cdots, \tau)$ which will be an embedding from $\mathcal{K}^d \to \mathbb{C}^{Nr}$, $r \leq d$. For an $\mathcal{R}$-module $M \subseteq \mathcal{K}^d$ (if M is full rank then $r = d$) let us then denote this embedding as τ_M. The set $\tau_M(M)$ is then a module lattice of dimension Nr. This map can then help to extend packing along the rank of a module and the concrete procedure for using this map for applications that necessarily require full packing of plaintexts remains to be explored. In the following paragraphs, we discuss three other interesting avenues of future work.

6.1 Bootstrapping

The next step would be to analyze the bootstrapping procedure of `ModRNS`. [CHK+18a] presents a series of procedures for bootstrapping in the CKKS/HEAAN scheme. It aims at refreshing or recrypting a ciphertext in a low level of modulus q to produce an equivalent ciphertext with a larger modulus $Q \gg q$ such that both of these ciphertexts decrypt approximately to the same message over their respective moduli. This approximate decryption would also involve a modular reduction resulting in a polynomial of a substantial degree and evaluating it during refreshing would again consume a lot of levels. To avoid this problem, [CHK+18a] instead proposed approximating the modular reduction function with a scaled sine function, both possessing a periodic behaviour in a given suitable interval and ultimately exploiting its relation with the complex exponential function to arrive at a desired polynomial approximation. The sequence of high-level routines involved in CKKS bootstrapping consists of `ModRaise`, `CoeffToSlot`, `EvalExp`, `ImgExt` and `SlotToCoeff` respectively. Since these routines are part of a homomorphic recryption, they are built upon combinations of fundamental homomorphic functions. For example, `CoeffToSlot` and `SlotToCoeff` involve linear transformations of vectors over plaintext slots, which in turn involve homomorphic multiplication and rotation functions. Likewise, conjugation is required for extracting the imaginary part of the recrypted ciphertext in `ImgExt`. Therefore, bootstrapping in the module variant of CKKS could follow along these lines. It serves as an interesting future scope for a complete analysis of the process as well as its feasible implementation.

6.2 Other HE schemes and MLWE

At the beginning of the paper we listed quite a few popular HE schemes such as the BFV/FV [FV12] and the BGV [BGV11] schemes that rely on the RLWE problem. Although

these schemes differ in their concrete noise management techniques, they share a similar underlying structure for their arithmetic circuits. For example, the polynomials involved in the en(de)cryption routines of the FV [FV12] HE scheme could be replaced by module elements such that the ciphertext ct now becomes a vector of polynomials contained in the module. The decryption is still valid as all modulo operations with respect to the ciphertext modulus or the plaintext modulus can be performed component-wise over the resulting module elements. In fact, homomorphic functions like homomorphic addition or multiplication could be extended to operate over module elements just like we showcased in our MLWE construction. One has to however apply the relinearization techniques with some caution so as to maintain the correctness of the decryption. A similar intuition can be applied to the BGV scheme [BGV11] as well. Therefore, it is not just CKKS but also the other RLWE-based HE schemes that could be ported to the module setting.

6.3 Memory-centric platforms

Various memory-centric platforms like processing-in-memory (PIM) and computing-in-memory (CIM) are emerging as the future paradigms of computing. PIM [AMY+23] platforms are ideal when there are several small computation tasks in an algorithm. Operands for the small tasks could be fetched fast from the near-memory, then processed in the processing elements of the platform, and finally written back in low latency to the near-memory. CIM platforms [VJV+19] offer computation in memory so data movement is minimal and is within the memory. The smaller the computation requirement, the easier it is to perform it in this setting.

The proposed ModRNS is a suitable algorithm for such platforms, owing to its ability to process smaller rings for homomorphic operations instead of one big ring. The NTT/INTT transforms, and the remaining homomorphic operations can be more efficiently evaluated using the ModRNS setting. Expensive operations such as relinearization are done independently on each non-linear module element. Even within a module element that consists of a component equal to the rank of the module, each of the module components is operated upon separately. Once these operations are done, a data share is indeed required to add the result to the remaining elements. The proposed hardware architecture is optimized for module components and features parallel dyadic units that can utilize all three relinearization key components in parallel.

In the emerging memory landscape, Micron Technology Inc. introduced 3D stacked memory [Sch], called Hybrid Memory Cube (HMC) that offers an aggregate bandwidth of 480 GBps at much lower power and a smaller form factor. More recently AMD designed the 3D Chiplets [JP] that can offer up to 2 terabytes of bandwidth. Such Chiplets feature multiple DRAM layers stacked on top of each other and connected through high bandwidth through silicon vias (TSVs) as shown in Figure 3. The bottom layer of the 3D stack is the logic layer which contains the necessary *computation circuits* for the memory controller. As the logic layer could not be arbitrarily large, a small ModRNS processing unit could be placed in the logic layer. All the above memory layers can be used to store all the required keys and ciphertexts. Thus, the data can be fetched fast via TSVs (i.e., high bandwidth), processed fast in the ModRNS processing unit, and then stored back fast into the memory layers. In this way, data will never leave the memory cube and thus the classical memory-processor data transfer bottlenecks will be avoided.

7 Conclusions and future directions

In this paper we discussed the use of module lattices to instantiate homomorphic encryption schemes and as a concrete construction presented a module version of the original ring-based CKKS scheme. We then proposed an RNS variant of the scheme to further improve the

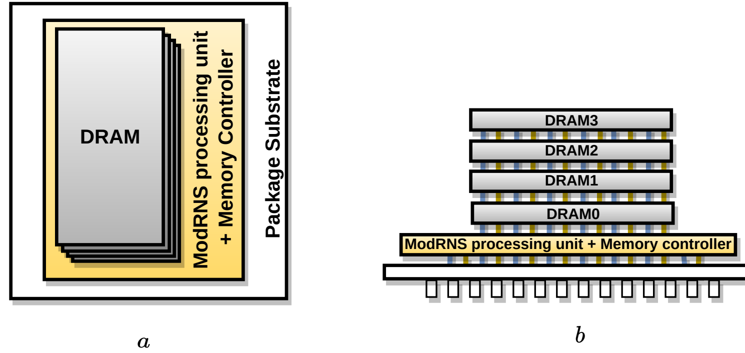

Figure 3: (a) shows the side-view of the 3D stack where the bottom layer after the substrate is the logic layer and the upper layers are memories, and (b) is a cross-sectional view to show the TSV interconnects connecting multiple different layers.

efficiency of HE operations. The original scheme's sub-routines were adapted in a way that requires minimum deviations from its fundamental approaches. In addition to reaping the benefits of the ring-based CKKS, our scheme also aids hardware reusability and long-lived parallel computation opportunities. It eliminates the need to increase the polynomial degree every time we want our scheme to support a higher level of security. With the MLWE feature of rank-reduction, the rank of the module can infact be managed dynamically. We also made a few important observations about module-based HE constructions: while ModHE can flourish in environments that need greater circuit depth, selecting appropriate parameter trade-offs is vital for achieving its full potential. It faces shortcomings in the context of an increased load on memory and the number of computations. We discussed a hardware implementation that is consistent with our primary goal of reusability and flexibility. We then provided results for homomorphic operations in the module setting.

MLWE-based HE schemes are fairly unexplored: there is immense scope for investigation of its properties and designing concrete hardware prototypes. Efficient bootstrapping for ModHE and detailed analysis of MLWE instances of other popular ring-based HE schemes are the major future direction of research in this area. We believe that experimenting with different parameter sets and improving functionalities of module-based homomorphic encryption in order to overcome its limitations would indeed be worthwhile.

A Correctness of `RingRNS` and description of subroutines

We first discuss in brief the correctness proof of CKKS and `RingRNS` after encryption and decryption. Then we give detailed equations for its important subroutines described in section 2.3 so that the reader gets familiar with how they give the desired results.

Let `ct` be an encryption of the message m such that m is obtained as a result of encoding $\mathbf{z} \in \mathbb{C}^{N/2}$. Then the decryption and decoding of `ct` will correctly return $\mathbf{z}$ if the scaling factor Δ satisfies $\Delta > N + 2B_{clean}$ where B_{clean} is the upper bound of the

decryption error [CKKS17]. The decryption $\pmod{q_0}$ works in the following way:

$$\begin{aligned}\texttt{RingRNS.Dec}_{\texttt{sk}}(\texttt{ct}) &= c_0 + c_1 \cdot s \\ &= (v \cdot \texttt{pk}[0] + m + e_0) + (v \cdot \texttt{pk}[1] + e_1) \cdot s \\ &= (v \cdot (-a \cdot s + e) + m + e_0) + (v \cdot a + e_1) \cdot s \\ &= -v \cdot a \cdot s + v \cdot e + m + e_0 + v \cdot a \cdot s + e_1 \cdot s \\ &= m + E\end{aligned}$$

where, $E = (v \cdot e + e_0 + e_1 \cdot s)$. Let the upper bound of E be given by B_{clean} such that $B_{clean} = 8\sqrt{2}\sigma N + 6\sigma\sqrt{N} + 16\sigma\sqrt{hN}$ (the readers may refer to [CKKS17] for details of the derivation). For a vector $\mathbf{z} \in \mathbb{C}^{N/2}$ an encryption of the encoded message m is also an encryption of $\Delta \cdot \tau^{-1} \circ \pi^{-1}(\mathbf{z})$ as given in section 2.4 where Δ is the scaling factor being used during encoding to maintain precision. The total error upper bound for E is equal to $B_{clean} + N/2 = B'$ (say). Then for the equality, $\pi(\tau(\lfloor \Delta^{-1}(m + B') \rceil)) = \pi(\tau(\lfloor \Delta^{-1}(m) \rceil))$ to hold we need to have $\Delta^{-1}B' < 1/2$. Thus, if $\Delta > N + 2B_{clean}$, the decoded message (after decryption) will correctly return $\mathbf{z}$.

The addition subroutine RingRNS.Add adds two ciphertexts ct and ct′ such that the resultant ciphertext $\pmod{Q_l}$ is $\texttt{ct}_{\texttt{add}} = (d_0 + d_1)$ as in section 2.3 and is decryptable under sk in the following way:

$$\begin{aligned}d_0 + d_1 \cdot s &= (c_0 + c_0') + (c_1 + c_1') \cdot s \\ &= (c_0 + c_1 \cdot s) + (c_0' + c_1' \cdot s) \\ &= m + m' + E'\end{aligned}$$

The equations above capture the homomorphic nature of this addition subroutine because $\texttt{RingRNS.Dec}_{\texttt{sk}}(\texttt{ct} + \texttt{ct}') = \texttt{RingRNS.Dec}_{\texttt{sk}}(\texttt{ct}) + \texttt{RingRNS.Dec}_{\texttt{sk}}(\texttt{ct}')$.

Similarly, RingRNS.mult is desired to fulfill the homomorphic property, $\texttt{RingRNS.Dec}_{\texttt{sk}}(\texttt{ct} * \texttt{ct}') = \texttt{RingRNS.Dec}_{\texttt{sk}}(\texttt{ct}) * \texttt{RingRNS.Dec}_{\texttt{sk}}(\texttt{ct}')$ where we use ‘$*$’ to refer to the multiplication operation between ciphertexts. In the following part we show why this is not very straightforward and additionally needs $\texttt{RingRNS.Relin}_{\texttt{evk}}$. Consider the RHS of the desired homomorphic multiplication property mentioned above done $\pmod{Q_l}$:

$$\begin{aligned}\texttt{RingRNS.Dec}_{\texttt{sk}}(\texttt{ct}) * \texttt{RingRNS.Dec}_{\texttt{sk}}(\texttt{ct}') &= (c_0 + c_1 \cdot s) \cdot (c_0' + c_1' \cdot s) \\ &= (-v \cdot a \cdot s + v \cdot e + m + e_0 + v \cdot a \cdot s + e_1 \cdot s) \\ &\cdot (-v \cdot a \cdot s + v \cdot e + m' + e_0' + v \cdot a \cdot s + e_1' \cdot s) \\ &= m \cdot m' + E''\end{aligned}$$

Observe that, $(c_0 + c_1 \cdot s) \cdot (c_0' + c_1' \cdot s) = c_0 \cdot c_0' + (c_0 \cdot c_1' + c_0' \cdot c_1) \cdot s + (c_1 \cdot c_1') \cdot s^2$. So the multiplied ciphertext $\texttt{ct} * \texttt{ct}'$ defined as $\texttt{d} = (d_0, d_1, d_2)$ such that $d_0 = c_0 \cdot c_0' \in \mathcal{R}_{Q_l}$, $d_1 = c_0 \cdot c_1' + c_1 \cdot c_0' \in \mathcal{R}_{Q_l}$, and $d_2 = c_1 \cdot c_1' \in \mathcal{R}_{Q_l}$ would satisfy the above-mentioned homomorphic property. But this means that d is decryptable using $(1, s, s^2)$ and not using $(1, s)$ and the decryption complexity will keep increasing with more multiplications. The subroutine $\texttt{RingRNS.Relin}_{\texttt{evk}}$ is used to avoid this problem as it helps to manage the non-linear terms (with respect to s) in the decryption equation. Recall the forms of the relinearized ciphertext $\texttt{ct}_{\texttt{relin}}$ and evaluation key $\texttt{evk} = (-a' \cdot s + e + s^2, a') \in \mathcal{R}^2_{pQ_L}$ as given in section 2.3. We show that the relinearized ciphertext $\texttt{ct}_{\texttt{relin}}$ would decrypt to approximately the same value as d using the following equations $\pmod{Q_l}$:

$$\begin{aligned}d_0' + d_1' \cdot s &= \big(d_0 + \texttt{RingRNS.ModDown}(d_2 \cdot (-a' \cdot s + e + s^2))\big) \\ &+ \big(d_1 \cdot s + \texttt{RingRNS.ModDown}(d_2 \cdot a \cdot s)\big) \\ &\approx d_0 + d_1 \cdot s + d_2 \cdot s^2\end{aligned}$$

B Correctness of Mod and description of subroutines

In order to demonstrate how the different subroutines work together in the case of Mod, we will continue our choice of module rank 2 for the all the following mathematical descriptions.

Let $\mathtt{ct} = (c_0, \mathbf{c_1}) \in \mathcal{R}_{q_l} \times \mathcal{R}_{q_l}^2$ be an encryption of the message m such that m is obtained as a result of encoding $\mathbf{z} \in \mathbb{C}^{N/2}$. Then the decryption and decoding of ct will correctly return $\mathbf{z}$ if the scaling factor Δ satisfies $\Delta > N + 2B_{enc}$ where B_{enc} is the upper bound of the decryption error as explained in section 3.1. The decryption works in the following way:

$$\begin{aligned}
\mathtt{Mod.Dec}_{\mathtt{sk}}(\mathtt{ct}) &= c_0 + \mathbf{c_1} \cdot \mathbf{s} \\
&= (\mathbf{v} \cdot \mathtt{pk}[0] + m + e_0) + (\mathbf{v} \cdot \mathtt{pk}[1] + \mathbf{e_1}) \cdot \mathbf{s} \\
&= (\mathbf{v} \cdot (-\mathbf{A} \cdot \mathbf{s} + \mathbf{e}) + m + e_0) + (\mathbf{v} \cdot \mathbf{A} + \mathbf{e_1}) \cdot s \\
&= -\mathbf{v} \cdot \mathbf{A} \cdot \mathbf{s} + \mathbf{v} \cdot \mathbf{e} + m + e_0 + \mathbf{v} \cdot \mathbf{A} \cdot \mathbf{s} + \mathbf{e_1} \cdot \mathbf{s} \\
&= m + E
\end{aligned}$$

Note that for module rank 2, $\mathbf{s}$ consists of two ring components, s_0 and s_1. So, $\mathbf{c_1} \cdot \mathbf{s} = c_{10} \cdot s_0 + c_{11} \cdot s_1$ where c_{10} and c_{11} are respectively the two ring components of $\mathbf{c_1}$.

The addition subroutine Mod.Add adds two ciphertexts ct and ct′ in $\mathcal{R}_{q_l} \times \mathcal{R}_{q_l}^2$ such that the resultant ciphertext $\pmod{q_l}$ is $\mathtt{ct}_{\mathtt{add}} = (c_0 + c_0', \mathbf{c_1} + \mathbf{c_1'})$ as in section 3 and is decryptable under sk in the following way:

$$\begin{aligned}
\mathtt{Mod.Dec}_{\mathtt{sk}}(\mathtt{ct}_{\mathtt{add}}) &= (c_0 + c_0') + (\mathbf{c_1} + \mathbf{c_1'}) \cdot \mathbf{s} \\
&= (c_0 + \mathbf{c_1} \cdot \mathbf{s}) + (c_0' + \mathbf{c_1'} \cdot \mathbf{s}) \\
&= m + m' + E'
\end{aligned}$$

Again, through the above-mentioned equations, the homomorphic nature of this addition subroutine can be perceived due to the relation, $\mathtt{Mod.Dec}_{\mathtt{sk}}(\mathtt{ct} + \mathtt{ct}') = \mathtt{Mod.Dec}_{\mathtt{sk}}(\mathtt{ct}) + \mathtt{Mod.Dec}_{\mathtt{sk}}(\mathtt{ct}')$.

Next, the desired homomorphic properties of Mod.Mult has been discussed in section 3 for a module of general rank r. For $r = 2$ and two ciphertexts ct and ct′ in $\mathcal{R}_{q_l} \times \mathcal{R}_{q_l}^2$, we have the following relation for decryption:

$$\begin{aligned}
m \cdot m' = c_0 \cdot c_0' &+ (c_0 \cdot c_{10}' + c_0' \cdot c_{10}) \cdot s_0 + (c_0 \cdot c_{11}' + c_0' \cdot c_{11}) \cdot s_1 \\
&+ (c_{10} \cdot c_{10}') \cdot s_0^2 + (c_{11} \cdot c_{11}') \cdot s_1^2 + (c_{10} \cdot c_{11}' + c_{10}' \cdot c_{11}) \cdot s_0 s_1
\end{aligned}$$

The multiplied ciphertext $\mathtt{ct}_{\mathtt{mult}}$ defined as $\mathtt{d} = (d_0, \mathbf{d_1}, \mathbf{d_2}, d_3)$ such that $d_0 = c_0 \cdot c_0' \in \mathcal{R}_{q_l}$, $\mathbf{d_1} = (c_0 \cdot c_{10}' + c_0' \cdot c_{10}, c_0 \cdot c_{11}' + c_0' \cdot c_{11}) \in \mathcal{R}_{q_l}^2$, $\mathbf{d_2} = (c_{10} \cdot c_{10}', c_{11} \cdot c_{11}') \in \mathcal{R}_{q_l}^2$ and $d_3 = c_{10} \cdot c_{11}' + c_{10}' \cdot c_{11} \in \mathcal{R}_{q_l}$ would satisfy the above-mentioned homomorphic property. This ciphertext is decryptable under $(1, \mathbf{s}, \mathbf{s^2}, s_0 s_1)$ and hence for similar reasons mentioned in the previous section, needs to be relinearized. Using the evaluation keys with respect to $\mathbf{d_2}$ and d_3 mentioned in $\mathtt{RingRNS.Relin}_{\mathtt{evk}}$ in section 3, the relinearized ciphertext would be $\mathtt{ct}_{\mathtt{relin}} = (d_0', \mathbf{d_1'})$ such that it is decryptable again under $(1, \mathbf{s})$ and approximates the actual decryption in the following way:

$$\begin{aligned}
d_0' + \mathbf{d_1'} \cdot \mathbf{s} &= \big(d_0 + P^{-1} \cdot (\mathbf{d_2} \cdot (-\mathbf{A_{d_2}} \cdot \mathbf{s} + \mathbf{e_{d_2}} + \mathbf{s^2}) + d_3 \cdot (-\mathbf{A_{d_3}}[\mathbf{0}] \cdot \mathbf{s} + \mathbf{e_{d_3}} + s_0 s_1)\big) \\
&\quad + \big(\mathbf{d_1} + P^{-1} \cdot (\mathbf{d_2} \cdot \mathbf{A_{d_2}} + d_3 \cdot \mathbf{A_{d_3}}[\mathbf{0}])\big) \cdot \mathbf{s} \\
&\approx d_0 + \mathbf{d_1} \cdot \mathbf{s} + \mathbf{d_2} \cdot \mathbf{s^2} + d_3 \cdot s_0 s_1
\end{aligned}$$

In case of the rank reduction subroutine to decrease the rank of a ciphertext $\mathtt{ct} = (c_0, \mathbf{c_1}) \in \mathcal{R}_{q_l} \times \mathcal{R}_{q_l}^2$ in a module of rank 2 to a module of rank 1 (that is, change it to a ciphertext in the base ring), the rank reduction key required will be $\mathtt{redk} =$

$(-\mathbf{A}_{red}^{1\times1} \cdot s_0 + e_{red} + P \cdot s_1, -\mathbf{A}_{red}^{1\times1})$. Notice that in this special case, the public matrix is actually just a polynomial a_{red}. The rank-reduced ciphertext $\mathtt{ct}' = (c_0', c_1') \in \mathcal{R}_{q_l} \times \mathcal{R}_{q_l}$ thus obtained approximates the ring decryption of the rank-reduced ciphertext with respect to s_0 in the following way:

$$\begin{aligned} c_0' + c_1' \cdot s_0 &= \left(c_0 + P^{-1} \cdot c_{11} \cdot (-a_{red} \cdot s_0 + e_{red} + P \cdot s_1)\right) + \left(c_{10} + c_{11} \cdot a_{red}\right) \cdot s_0 \\ &\approx c_0 + c_{10} \cdot s_0 \end{aligned}$$

The two above-mentioned sections in Appendix show that the structures of both `RingRNS` and `Mod` share similarities by virtue of the homomorphic properties desired by the different operations. However, it also brings out the differences in terms of the increased algebraic manipulations required in the module case as well as the additional feature of rank reduction.

C Acknowledgement

This work was supported in part by the Samsung Electronics co. ltd., Samsung Advanced Institute of Technology and the State Government of Styria, Austria – Department Zukunftsfonds Steiermark.

References

[ACPS09] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009*, pages 595–618, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[AD17] Martin Albrecht and Amit Deo. Large modulus ring-lwe $\geq$ module-lwe. pages 267–296, 11 2017.

[AdCY+23] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Tugce Yazicigil, Anantha P. Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. FAB: An FPGA-based Accelerator for Bootstrappable Fully Homomorphic encryption. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*, pages 882–895. IEEE, 2023.

[AMY+23] Kazi Asifuzzaman, Narasinga Rao Miniskar, Aaron R. Young, Frank Liu, and Jeffrey S. Vetter. A survey on processing-in-memory techniques: Advances and challenges. *Memories - Materials, Devices, Circuits and Systems*, 4:100022, 2023.

[BBTV] Jonas Bertels, Michiel Van Beirendonck, Furkan Turan, and Ingrid Verbauwhede. Hardware acceleration for fhew. https://github.com/FHE-org/fhe-org.github.io/files/11574584/024-Hardware_accelerator_for_FHEW_slides.pdf accessed on 06 July, 2023.

[BDK+21] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium. Proposal to NIST PQC Standardization, Round3, 2021. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions.

[BGK11] Zvika Brakerski, Shafi Goldwasser, and Yael Tauman Kalai. Black-box circular-secure encryption beyond affine functions. In Yuval Ishai, editor,

Theory of Cryptography, pages 201–218, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[BGV11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *Electron. Colloquium Comput. Complex.*, page 111, 2011.

[BHHI10] Boaz Barak, Iftach Haitner, Dennis Hofheinz, and Yuval Ishai. Bounded key-dependent message security. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 423–444, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[BHHO08] Dan Boneh, Shai Halevi, Mike Hamburg, and Rafail Ostrovsky. Circular-secure encryption from decision diffie-hellman. In David Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, pages 108–125, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[BHM+20] Ahmad Al Badawi, Louie Hoang, Chan Fook Mun, Kim Laine, and Khin Mi Mi Aung. Privft: Private and fast text classification with homomorphic encryption. *IEEE Access*, 8:226544–226556, 2020.

[BV11a] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 97–106, 2011.

[BV11b] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 505–524, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[CDW17] Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. Short stickelberger class relations and application to ideal-svp. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 324–348, Cham, 2017. Springer International Publishing.

[CG99] Eleanor Chu and Alan George. *Inside the FFT black box: serial and parallel fast Fourier transform algorithms*. CRC press, 1999.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[CHK+18a] 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, *Advances in Cryptology – EUROCRYPT 2018*, pages 360–384, Cham, 2018. Springer International Publishing.

[CHK+18b] 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, *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*, volume 11349 of *Lecture Notes in Computer Science*, pages 347–368. Springer, 2018.

[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, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.

[CS16] Ana Costache and Nigel P. Smart. Which ring based somewhat homomorphic encryption scheme is best? In Kazue Sako, editor, *Topics in Cryptology - CT-RSA 2016*, pages 325–340, Cham, 2016. Springer International Publishing.

[DGH+21] Christoph Dobraunig, Lorenzo Grassi, Lukas Helminger, Christian Rechberger, Markus Schofnegger, and Roman Walch. Pasta: A case for hybrid homomorphic encryption. Cryptology ePrint Archive, Paper 2021/731, 2021. https://eprint.iacr.org/2021/731.

[DKR+21] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. SABER. Proposal to NIST PQC Standardization, Round3, 2021. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions.

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[FSK+21] Axel Feldmann, Nikola Samardzic, Aleksandar Krastev, Srini Devadas, Ron Dreslinski, Karim Eldefrawy, Nicholas Genise, Christopher Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption (extended version), 2021.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 144, 2012.

[Gen09] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009.

[GVBP+22] Robin Geelen, Michiel Van Beirendonck, Hilder V. L. Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios Dimou, Ingrid Verbauwhede, Frederik Vercauteren, and David W. Archer. Basalisc: Flexible asynchronous hardware accelerator for fully homomorphic encryption, 2022.

[JKA+21] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):114–148, Aug. 2021.

[JP] Cadence John Park, Product Management Group Director for Advanced IC Packaging. Chiplets and heterogeneous packaging are changing system design and analysis. https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/ic-package-design-analysis/chiplets-and-heterogeneous-packaging-are-changing-system-design-and-analysis.pdf accessed on 14 April, 2023.

[KKK+22] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. Bts: An accelerator for bootstrappable fully homomorphic encryption. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 711–725, New York, NY, USA, 2022. Association for Computing Machinery.

[KLK+22] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, John Kim, Minsoo Rhu, and Jung Ho Ahn. Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse, 2022.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 1–23, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptogr.*, 75(3):565–599, 2015.

[MAK+23] Ahmet Can Mert, Aikata, Sunmin Kwon, Youngsam Shin, Donghoon Yoo, Yongwoo Lee, and Sujoy Sinha Roy. Medha: Microcoded hardware accelerator for computing on encrypted data. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(1):463–500, 2023.

[MTY11] Tal Malkin, Isamu Teranishi, and Moti Yung. Efficient circuit-size independent public key encryption with kdm security. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, pages 507–526, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[RAD+78] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, page 84–93, New York, NY, USA, 2005. Association for Computing Machinery.

[RJV+18] Sujoy Sinha Roy, Kimmo Järvinen, Jo Vliegen, Frederik Vercauteren, and Ingrid Verbauwhede. HEPCloud: An FPGA-based multicore processor for FV somewhat homomorphic function evaluation. *IEEE Transactions on Computers*, 67(11):1637–1650, 2018.

[RLPD20] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. Heax: An architecture for computing on encrypted data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1295–1309, New York, NY, USA, 2020. Association for Computing Machinery.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, feb 1978.

[SAB+21] Peter Schwabe, Roberto Avanzi, Joppe Bos, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehle. CRYSTALS-KYBER. Proposal to NIST PQC Standardization, 2021. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions.

[Sch] Andreas Schlapka. Micron announces shift in high-performance memory roadmap strategy. https://www.micron.com/about/blog/2018/august/micron-announces-shift-in-high-performance-memory-roadmap-strategy accessed on 14 April, 2023.

[SFK+22] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. Craterlake: A hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 173–187, New York, NY, USA, 2022. Association for Computing Machinery.

[SYYT20] Yang Su, Bailong Yang, Chen Yang, and Luogeng Tian. Fpga-based hardware accelerator for leveled ring-lwe fully homomorphic encryption. *IEEE Access*, 8:168008–168025, 2020.

[SYYZ22] Yang Su, Bai-Long Yang, Chen Yang, and Song-Yin Zhao. Remca: A reconfigurable multi-core architecture for full rns variant of bfv homomorphic evaluation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(7):2857–2870, 2022.

[TRG+20] Jonathan Takeshita, Dayane Reis, Ting Gong, Michael Niemier, X. Sharon Hu, and Taeho Jung. Algorithmic acceleration of b/fv-like somewhat homomorphic encryption for compute-enabled ram. Cryptology ePrint Archive, Report 2020/1223, 2020. https://ia.cr/2020/1223.

[TRV20] Furkan Turan, Sujoy Sinha Roy, and Ingrid Verbauwhede. HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA. *IEEE Transactions on Computers*, 69(8):1185–1196, 2020.

[VJV+19] Naveen Verma, Hongyang Jia, Hossein Valavi, Yinqi Tang, Murat Ozatay, Lung-Yen Chen, Bonan Zhang, and Peter Deaville. In-memory computing: Advances and prospects. *IEEE Solid-State Circuits Magazine*, 11(3):43–55, 2019.

[Xil] Xilinx. Alveo U200 and U250 Data Center Accelerator Cards Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf.

[ZYC+20] Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, Shaojun Wei, and Leibo Liu. Highly efficient architecture of newhope-nist on fpga using low-complexity ntt/intt. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):49–72, Mar. 2020.