DPad-HE: Towards Hardware-friendly Homomorphic Evaluation using 4-Directional Manipulation

Wenxu Tang
University of Science and Technology of China
School of Cyber Science and Technology
Hefei, China
wenxutang@mail.ustc.edu.cn

Fangyu Zheng *
University of Chinese Academy of Sciences
School of Cryptology
Beijing, China
zhengfangyu@ucas.ac.cn

Guang Fan
University of Chinese Academy of Sciences
School of Cryptology
Beijing, China
fanguang328@gmail.com

Tian Zhou
University of Science and Technology of China
School of Cyber Science and Technology
Hefei, China
weekdayzt@mail.ustc.edu.cn

Jingqiang Lin
University of Science and Technology of China
School of Cyber Science and Technology
Hefei, China
linjq@ustc.edu.cn

Jiwu Jing
University of Chinese Academy of Sciences
School of Cryptology
Beijing, China
jwjing@ucas.ac.cn

Abstract

Module Learning with Errors (MLWE) based approaches for Fully Homomorphic Encryption (FHE) have garnered attention due to their potential to enhance hardware-friendliness and implementation efficiency. However, despite these advantages, their overall performance still trails behind traditional schemes based on Ring Learning with Errors (RLWE). This indicates that while MLWE-based constructions hold promise, there remain significant challenges to overcome in bridging the performance gap with RLWE-based FHE schemes.

By uncovering the reasons for the unsatisfactory performance of prior schemes and pinpointing the fundamental differences in the design of MLWE-based FHE compared to traditional approaches, the paper introduces DPad-HE with a novel design incorporating manipulation in the module rank dimension. The newly introduced operations, rank-up, and rank-down, effectively regulate the scale of gadget decomposition, reducing the computational workload of key-switching by several times. Taking CKKS as a case study, the evaluation showcases the comprehensive advantages of DPad-HE over the state-of-the-art MLWE-based scheme, resulting in a performance boost of 1.26× to 5.71×, a reduction in key size from 1/3 to 3/4, with enhanced noise control. To test the hardware-friendliness of the solution, DPad-HE is also implemented on GPU. Notably, DPad-HE demonstrates that, for the first time, the execution latency of MLWE-based schemes can achieve comparable performance with traditional RLWE ones, especially on the GPU platform where a speedup up to 1.41× is witnessed. Additionally, this paper provides a lightweight conversion method between RLWE and MLWE ciphertexts, allowing for flexible selection of RLWE and MLWE settings during a single complete evaluation process. This opens up new possibilities for both RLWE-based and MLWE-based FHEs.

* Fangyu Zheng is the corresponding author (*E-mail: zhengfangyu@ucas.ac.cn*).

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690280

CCS Concepts

• **Security and privacy** → **Cryptography**.

Keywords

Learning with errors (LWE), Fully homomorphic encryption, Module-LWE, Ring-LWE, Hardware friendliness

ACM Reference Format:
Wenxu Tang, Fangyu Zheng *, Guang Fan, Tian Zhou, Jingqiang Lin, and Jiwu Jing. 2024. DPad-HE: Towards Hardware-friendly Homomorphic Evaluation using 4-Directional Manipulation. In *Proceedings of Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690280

1 Introduction

Fully homomorphic encryption (FHE) is a groundbreaking cryptographic technique that allows for computations to be performed on encrypted data without the need for decryption. FHE originated in 1978 with Rivest's work [41], but it wasn't realized until Gentry's breakthrough in 2009 with his theoretical blueprint [24] that FHE gained significant attention, despite its impracticality at the time. Over the past decade, substantial efforts have been devoted to advancing FHE towards practical applications. Various representative FHE schemes, including BGV [9], BFV [20], CKKS [12], FHEW [19] and TFHE [14], have emerged and found applications in both industry and academia [6, 35]. However, despite significant algorithmic improvements, poor performance remains a major obstacle, hindering wider adoption of FHE. Consequently, there's a pressing need for FHE accelerations, which have garnered widespread attention.

To address the performance issues, many implementations now employ specialized hardware like GPUs [17, 21, 29] and FPGAs [1, 28]. GPUs, particularly known for accelerating AI tasks, offer remarkable speed gains for conventional cryptographic workloads like RSA and ECC [22, 23], achieving over a 1000× speedup compared to CPUs. However, in the case of FHE, the performance improvement is not as significant. For example, according to findings in the paper by Jung et al. [29], the performance increase of RLWE-based CKKS on V100 GPUs is only approximately 200×.

GPUs offer significant on-chip memory resources. In contrast to the comparatively limited on-chip memory of CPUs, a single Streaming Multiprocessor (SM) typically contains 64k 32-bit registers (256kB) and 48-228kB of shared memory [37]. For instance, considering H100 GPUs equipped with 144 SMs [38], this translates to approximately 36MB of registers and 32MB of shared memory. While computations in cryptographic schemes like RSA and ECC can often be accommodated within such ample on-chip memory, due to their extensive memory operations, FHE schemes struggle to fully exploit GPU architecture. For example, RLWE-based CKKS requires extremely large-scale polynomial computations with $N = 2^{14}, 2^{15}, 2^{16}$ or more. Such large-scale computations must be carried out using off-chip memory, partly because on-chip memory is difficult to accommodate, and more importantly, because global off-chip memory is needed for inter-thread computation collaboration.

With this insight in mind, our main goal is to *create a more hardware-friendly (and hopefully more efficient) FHE solution that efficiently transitions existing schemes.* Therefore, we shift our focus to MLWE, a scheme widely utilized in the standardized PQC Kyber and Dilithium [7, 18] and other lattice-based algorithms.

1.1 The Pros and Cons of MLWE-based FHE

As highlighted by numerous previous researchers [7, 18, 36], MLWE exhibits superior hardware-friendly characteristics compared to RLWE settings, offering greater flexibility, hardware reusability, parallelism, and better security assumptions. Recent research ConvKyber [43] showcased a tenfold acceleration over traditional butterfly operations when employing GPUs' CUDA cores, with the application of GPUs' tensor cores for small-degree NTT utilized in MLWE-based PQC Kyber [7]. ConvKyber utilizes register-bound computations throughout the NTT process, suggesting significant performance enhancements could be attained if similar strategies were applied to FHE. Additionally, In MLWE, a vector of several small ring components is involved, which can be processed in parallel. MLWE's inherent advantages align particularly well with parallel platforms like GPUs, making MLWE-based FHE schemes highly attractive.

Furthermore, homomorphic schemes based on MLWE offer adaptable methods for adjusting dimensions. It is well-known that, due to the necessity of negacyclic NTT, polynomial degrees must be powers of 2. In RLWE settings, increasing the number of multiplicative layers requires doubling the polynomial degree each time, even if additional layers or packing sizes are not needed. MLWE schemes have two adjustable options: the degree of polynomials and the rank r, where r can be any positive integer. This provides people with more flexibility when selecting parameters: doubling the degree to access double ciphertext modulus as well as more message packing capacity, or just increasing the rank to gain ciphertext modulus gradually. This smooth parameter selection may naturally lead to reduced computational complexity in certain cases for MLWE-based FHE schemes.

However, MLWE schemes directly migrated from RLWE schemes face serious performance issues in homomorphic multiplication algorithm. The previous work ModHE [36] reported a 1.7× slowdown compared with RLWE on the FPGA platform when $r = 2$. Even worse, this performance loss will deteriorate dramatically with the increase of r. The main reason is that the MLWE ciphertext, which is in the form of $(b, \boldsymbol{a})$ satisfying that $m = b + \langle \boldsymbol{a}, \boldsymbol{s} \rangle$ where $\boldsymbol{a}$ and $\boldsymbol{s}$ are size-r vectors of polynomials, will expand to size-$O(r^2)$ when a homomorphic multiplication is performed. Since the algorithm is asymptotically much worse than RLWE, even with better hardware-friendly characteristics, the overall performance of MLWE-based schemes remains inferior to RLWE-based schemes.

1.2 Contributions and Paper Organization

Although ModHE's [36] performance compared to existing RLWE settings is significantly disadvantaged, it introduces an intriguing subroutine called "rank reduction". In leveled homomorphic encryption, the ciphertext modulus diminishes as more and more levels are consumed. With a smaller ciphertext modulus, the lattice rank could be reduced to an appropriate r' from r (where $r' < r$) without compromising the security of the homomorphic encryption scheme.

Inspired by the "rank-reduction" concept, this paper introduces DPad-HE based on the MLWE setting. Similar to the directional pad (D-Pad) found on game controllers, DPad-HE features four directional manipulations (modulus-up and modulus-down as well as rank-up and rank-down) during the homomorphic evaluation process. The newly introduced rank-up and rank-down operations enable DPad-HE to significantly outperform ModHE in terms of computational complexity, key size and noise management.

Remarkably, unlike the significant performance disadvantage of ModHE compared to traditional RLWE implementations, DPad-HE demonstrates *for the first time that, the execution latency of MLWE-based schemes can achieve comparable or even superior performance compared to traditional RLWE ones.* More specifically, this work makes several significant contributions.

- **An insightful finding on MLWE-based FHE** is provided to shed light on the underperformance of previous MLWE-based schemes ModHE in Section 3.1. After comprehensive measurements of the overall workload of MLWE-based FHE schemes, we arrived at an observation that is somewhat inconsistent with traditional RLWE setting, namely, in MLWE-based FHE schemes ported directly from RLWE, the computational cost of inner product calculations which suffer greatly from ciphertext expansions caused by the combined effects of Tensor and gadget decomposition, grows significantly with the rank r, compared with the computational cost of NTT.

- **A more efficient MLWE-based FHE**, called DPad-HE is proposed in Section 3.2 to Section 3.5 upon the above insights. DPad-HE introduces a method based on rank manipulation to reduce the overwhelming complexity of inner product operation. To offset ciphertext expansions, we employ a temporary (higher) module rank with a small set of gadget vectors, along with novel rank manipulation operations. These novel operations lie at the core of DPad-HE's design, greatly reducing computational complexity, memory consumption, and noise growth. The evaluation showcases the 1.26× to 5.71× performance advantages of DPad-HE over ModHE.
- **A lightweight toolkit that converts ciphertexts between RLWE and MLWE** is additionally proposed in Section 4, enabling DPad-HE to interact with existing RLWE-based schemes. A micro-benchmark demonstrates the lightweight nature of the conversion, allowing for the flexible selection of RLWE and MLWE settings during a single complete evaluation process, leveraging their respective advantages at different stages.
- **Implementations and evaluations based on CPU and GPU** are provided in Section 5 to validate the effectiveness of DPad-HE. We conducted experiments using a CPU/GPU implementation based on the SEAL framework. Taking CKKS as a case study, the evaluation demonstrates that execution latency of DPad-HE can outperform traditional RLWE ones with up to 1.25× and 1.41× speedup in CPU and GPU, respectively.

It is worth noting that while our insights stem from GPU implementation, DPad-HE is a universal theoretical scheme that offers improvements across various platforms, including CPU, and especially for GPU, FPGA and ASIC. This is where the "hardware-friendly" aspect of DPad-HE originates.

2 Preliminaries

This section provides the notations used in the paper and the background of the MLWE-based FHE schemes.

2.1 Notation

For a power-of-two integer N, we define the polynomial ring $\mathcal{R}_N = \mathbb{Z}[X]/(X^N+1)$. Also, we define $\mathcal{R}_{q,N} = \mathbb{Z}_q[X]/(X^N+1) = \mathcal{R}_N/q\mathcal{R}_N$, the residue ring of $\mathcal{R}_N$ modulo an integer q. When the context is clear, we may omit N in $\mathcal{R}_{q,N}$ and denote it as $\mathcal{R}_q$. We denote a rank-r module over $\mathcal{R}_q$ as $\mathcal{R}_q^r$. An element of $\mathcal{R}_{q,N}$ is a polynomial of the form $a(X) = \sum_{i=0}^{N-1} a_i X^i$ with each of its coefficients in $\mathbb{Z}_q$. We will always choose q as a product of primes of q_i such that $q_i = 1 \bmod 2N$ for all i. This enables fast multiplication over $\mathcal{R}_{q,N}$, based on the Chinese Remainder Theorem (CRT) and the Number Theoretic Transform (NTT). We use $\mathsf{NTT}_m(\cdot)$ to denote the m-point negacyclic NTT operation, and use $\mathsf{INTT}_m(\cdot)$ to denote the inverse operation. We use $\psi_{q,2m}$ to denote the primitive $2m$-th root of unity in $\mathbb{Z}_q$ whenever $m|(q-1)$. When the context is clear, we may omit q in $\psi_{q,2m}$.

We denote single elements (polynomials or numbers) in lower-case italics, e.g. a, and vectors of such elements in lower-case bold face, e.g. $\boldsymbol{a}$, and matrices in upper-case bold face, e.g. $\boldsymbol{A}$. We denote the inner product of two vectors by $\langle\cdot,\cdot\rangle$ or simply $\cdot$. All the vectors are column vectors unless otherwise stated. We use $x \leftarrow D$ to denote that the sampling x according to distribution D, and use $\boldsymbol{x} \leftarrow D^k$ to denote that the vector $\boldsymbol{x}$ is generated by sampling k times from D independently. We use $(\boldsymbol{v}_0, \boldsymbol{v}_1, \cdots, \boldsymbol{v}_m)$ to horizontally stack these vectors to generate a matrix or a tuple, and use $(\boldsymbol{v}_0 \| \boldsymbol{v}_1 \| \cdots \| \boldsymbol{v}_m)$ to vertically stack these vectors to generate a longer vector. We use $\boldsymbol{v}[a:b]$ to represent the left-closed, right-open slice of the vector $\boldsymbol{v}$. We denote the Hadamard multiplication by $*$ and the Kronecker multiplication by $\otimes$.

2.2 The MLWE-based CKKS scheme

We first recap the ModRNS FHE scheme proposed in ModHE [36, §3.2], which is a CKKS-like scheme based on MLWE, and its packing methods.

2.2.1 Module-LWE. Combined the security advantages of LWE [39] and the flexibility of Ring-LWE [34], Module-LWE (MLWE) was introduced in [9, 32]. We denote an MLWE ciphertext with modulus q, rank r and degree n as $\mathsf{MLWE}^r_{q,n}$, which is an element $\mathsf{ct} = (c_0, \cdots, c_r) \in \mathcal{R}^{1+r}_{q,n}$. We also use the notation of $\mathsf{ct} = (b, \boldsymbol{a})$ where $b = c_0$ and $\boldsymbol{a} = (c_1, \cdots, c_r)$. We call the ciphertext ct the encryption of plaintext $m \in \mathcal{R}_{q,n}$ with secret key $\boldsymbol{s} = (s_0, \cdots, s_{r-1}) \in \mathcal{R}^r_{q,n}$, if $b + \langle \boldsymbol{a}, \boldsymbol{s}\rangle = m$. We call $N = r \cdot n$ the dimension of the ciphertext.

2.2.2 Encoding and decoding. Like CKKS, the scheme bases its encoding and decoding functions on the canonical embedding $\tau : \mathbb{R}[X]/(X^n+1) \to \mathbb{C}^{n/2}$, which is an isomorphism mapping $m(X)$ into $\boldsymbol{m} \in \mathbb{C}^{n/2}$ by evaluating $m(X)$ at the primitive $2n$-th roots of unity $\xi_j = \xi^{5^j}$ for $0 \le j < n/2$. Therefore, the scheme is able to provide SIMD property. Also, a real scaling factor $\Delta \ge 1$ is used to preserve the precision.

- **Encoding**: $m(X) \leftarrow \mathsf{Ecd}(\boldsymbol{m}, \Delta)$. Given a complex message $\boldsymbol{m} \in \mathbb{C}^{n/2}$ and a scaling factor Δ, return the plaintext $m(X) = \lfloor \Delta \cdot \tau^{-1}(\boldsymbol{m}) \rceil \in \mathcal{R}_n$.
- **Decoding**: $\boldsymbol{m} \leftarrow \mathsf{Dcd}(m(X), \Delta)$. Given a plaintext $m(X) \in \mathcal{R}_n$ and a scaling factor Δ, return the message $\boldsymbol{m} = \tau(\Delta^{-1} m(X)) \in \mathbb{C}^{n/2}$.

2.2.3 Basic operations.

- **Setup**: params $\leftarrow$ $\mathsf{MLWE.Setup}(1^\lambda)$. Given the security parameter λ, return the public parameters including the degree n, the rank r, two distributions χ_{key} and χ_{err} over $\mathcal{R}_{2,n}$, the chain of coprime RNS modulus $\{q_0, q_1, \cdots, q_L\}$, the auxiliary modulus $P = \prod_{i=0}^{k} p_i$ and some precomputed constants. Set $Q_\ell = \prod_{i=0}^{\ell} q_i$ for $0 \le \ell \le L$.
- **Key generation**: $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(\mathsf{params})$. Return a pair of secret and public keys.
  - Sample $\boldsymbol{s} \leftarrow \chi^r_{\text{key}}$ where the Hamming weight of s_i is h for each i. Set the secret key as $\mathsf{sk} = (1, \boldsymbol{s})$.
  - Sample $\boldsymbol{A} \leftarrow \mathcal{U}(\mathcal{R}^{r\times r}_{Q_L})$ and $\boldsymbol{e} \leftarrow \chi^r_{err}$. Set the public key as $\mathsf{pk} = (\boldsymbol{b}, \boldsymbol{A}) \in \mathcal{R}^r_{Q_L} \times \mathcal{R}^{r\times r}_{Q_L}$ where $\boldsymbol{b} = [-\boldsymbol{A}\cdot\boldsymbol{s} + \boldsymbol{e}]_{Q_L}$.

- **Switching key generation**: $\mathsf{swk} \leftarrow \mathsf{KSGen}_{\mathsf{sk}}(\boldsymbol{s}')$. Sample $\boldsymbol{A}' \leftarrow \mathcal{U}(\mathcal{R}_{PQ_L}^{r\times r})$ and $\boldsymbol{e} \leftarrow \chi_{\mathsf{err}}^{r}$, return key-switching key $\mathsf{swk} = (\boldsymbol{b}', \boldsymbol{A}') \in \mathcal{R}_{PQ_L}^{r} \times \mathcal{R}_{PQ_L}^{r\times r}$ where $\boldsymbol{b} = -\boldsymbol{A}' \cdot \boldsymbol{s} + \boldsymbol{e} + P \cdot \boldsymbol{s}' \bmod PQ_L$.
- **Encryption**: $\mathsf{ct} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m(X))$. Given a plaintext $m(X) \in \mathcal{R}_n$, sample $\boldsymbol{v} \leftarrow \chi_{\mathsf{enc}}^{r}$ and $(e, \boldsymbol{\epsilon}) \leftarrow \chi_{\mathsf{err}}^{1+r}$, output the ciphertext $\mathsf{ct} = [(\langle \mathsf{pk}[0], \boldsymbol{v}\rangle, \mathsf{pk}[1]^{\top} \cdot \boldsymbol{v}) + (m(X) + e, \boldsymbol{\epsilon})]_{Q_L} \in \mathcal{R}_{Q_L}^{1+r}$.
- **Decryption**: $m(X) \leftarrow \mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct})$. Given a ciphertext $\mathsf{ct} \in \mathcal{R}_{Q_\ell}^{1+r}$ for some $0 \le \ell \le L$, return $m(X) = [\langle \mathsf{ct}, \mathsf{sk}\rangle]_{q_0}$ where q_0 is the 0-level modulus.
- **Rescale**: $\mathsf{ct}_{\mathsf{rs}} \leftarrow \mathsf{RS}_Q(\mathsf{ct})$. Given a ciphertext $\mathsf{ct} \in \mathcal{R}_{Q_\ell}^{1+r}$, return $\mathsf{ct}_{\mathsf{rs}} = \lfloor \frac{Q_{\ell-1}}{Q_\ell} \mathsf{ct} \rceil \bmod Q_{\ell-1} \in \mathcal{R}_{Q_{\ell-1}}^{1+r}$. The purpose of RS is to reduce the error and maintain the scaling factors.
- **Addition/subtraction**: $\mathsf{ct}_{\mathsf{add}}/\mathsf{ct}_{\mathsf{sub}} \leftarrow \mathsf{Add/Sub}(\mathsf{ct}, \mathsf{ct}')$. Given two ciphertext $\mathsf{ct}, \mathsf{ct}' \in \mathcal{R}_{Q_\ell}^{1+r}$, return the ciphertext $\mathsf{ct}_{\mathsf{add}}/\mathsf{ct}_{\mathsf{sub}} = [\mathsf{ct} \pm \mathsf{ct}']_{Q_l}$ satisfying that $\mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct}_{\mathsf{add}}) = \mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct}) + \mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct}')$ (resp. $\mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct}_{\mathsf{sub}}) = \mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct}) - \mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct}')$).

2.3 The Multiplication in MLWE-based CKKS

The CKKS-like MLWE multiplication procedure can be described in three sub-procedures:

(1) **Tensor**: $\mathsf{ct}_{\mathsf{ten}} \leftarrow \mathtt{Tensor}(\mathsf{ct}_1, \mathsf{ct}_2)$. Given two ciphertext $\mathsf{ct}_1 = (b_1, \boldsymbol{a}_1)$ and $\mathsf{ct}_2 = (b_2, \boldsymbol{a}_2) \in \mathcal{R}_{Q_\ell}^{1+r}$, return $\mathsf{ct}_{\mathsf{ten}} = (b_1 b_2, b_1\boldsymbol{a}_2 + b_2\boldsymbol{a}_1, \boldsymbol{a}_1 \otimes \boldsymbol{a}_2) \in \mathcal{R}_{Q_\ell}^{1+r+\binom{1+r}{2}}$. Writing $\mathsf{sk} = (1, \boldsymbol{s})$, we have:
$$\mathsf{ct}_{\mathsf{ten}} \cdot (1, \boldsymbol{s}, \boldsymbol{s} \otimes \boldsymbol{s}) = \mathsf{Dec}(\mathsf{ct}_1) \cdot \mathsf{Dec}(\mathsf{ct}_2)$$

(2) **Relinearize**: $\mathsf{ct}_{\mathsf{relin}} \leftarrow \mathtt{Relin}(\mathsf{ct}_{\mathsf{ten}}, \mathsf{rlk})$. Given a ciphertext $\mathsf{ct}_{\mathsf{ten}} = (\mathsf{ct}, \mathsf{ct}', \mathsf{ct}'') \in \mathcal{R}_{Q_\ell}^{1+r+\binom{1+r}{2}}$ and a relinearization key $\mathsf{rlk} \in \mathcal{R}_{PQ_\ell}^{(1+r)\times\binom{1+r}{2}}$, return $\mathsf{ct}_{\mathsf{relin}} \in \mathcal{R}_{Q_\ell}^{1+r}$ defined as follows:
$$\mathsf{ct}_{\mathsf{relin}} = (\mathsf{ct}, \mathsf{ct}') + \left\lceil \frac{\mathsf{ct}'' \cdot \mathsf{rlk}}{P} \right\rfloor$$

(3) **Rescale**: $\mathsf{ct}_{\mathsf{rs}} \leftarrow \mathsf{RS}_{q_\ell}(\mathsf{ct}_{\mathsf{relin}})$ where $q_\ell = Q_\ell / Q_{\ell-1}$. The output $\mathsf{ct}_{\mathsf{rs}}$ is an element of $\mathcal{R}_{Q_{\ell-1}}^{1+r}$. The purpose of RS in homomorphic multiplication is to reduce the error and maintain the scale factor.

We define the homomorphic multiplication $\mathtt{HMult} = \mathtt{Relin} \circ \mathtt{Tensor}$. The relinearization step needs a relinearization key rlk which is generated by KSGen but in a different manner: Set $r_{\mathsf{ten}} = \binom{r+1}{2}$, sample $\boldsymbol{A}_{\mathsf{rlk}} \leftarrow \mathcal{U}\left(\mathcal{R}_{PQ_L}^{r_{\mathsf{ten}}\times r}\right)$ and $\boldsymbol{e} \leftarrow \chi_{\mathsf{err}}^{r_{\mathsf{ten}}}$, return the relinearization key $\mathsf{rlk} = (\boldsymbol{b}_{\mathsf{rlk}}, \boldsymbol{A}_{\mathsf{rlk}})^{\top} \in \mathcal{R}_{PQ_L}^{(1+r)\times r_{\mathsf{ten}}}$ where $\boldsymbol{b}_{\mathsf{rlk}} = -\boldsymbol{A}_{\mathsf{rlk}} \cdot \boldsymbol{s} + \boldsymbol{e} + P \cdot (\boldsymbol{s} \otimes \boldsymbol{s}) \bmod PQ_L$.

3 The DPad-HE Scheme based on Module-LWE

This section will analyze previous work ModHE [36], uncovering an intriguing discovery regarding MLWE-based FHE, which differs notably from RLWE-based settings: in key-switching, inner product computations closely tied to gadget decomposition may dominate over NTT, consuming considerable time. To address this, the section will introduce DPad-HE, which manipulates an extra variable, the *module rank*, and provides thorough complexity and noise analysis.

3.1 Generalization and Analysis of MLWE-based FHE

In this section, we first go through a thorough analysis of key-switching-related operations in MLWE-based CKKS. We will demonstrate that the inner product part has become the major concern of the overall performance.

3.1.1 Generalized key-switching method. In this section, we first generalize the key-switching method proposed in ModHE to analyze its performance and to serve as a building block for our subsequent algorithms. To build a unified framework for Relin and RankReduce, our generalization encompasses two main aspects: the size of RNS-based gadget decomposition, as described in [8, 27], and the rank of the secret keys before and after key-switching procedure.

Given params = MLWE.$\mathtt{Setup}(1^\lambda)$ described in Section 2.2.3 with the same notations, we define the gadget decomposition size[1] as $d = \lceil(\ell+1)/k\rceil$. With $\{Q_j'\}_{0\le j<d} = \{\prod_{i=jk}^{\min((j+1)k-1,\ell)} q_i\}_{0\le j<d}$, we partition Q_ℓ into d composite-numbers Q_j', each composed of a maximum of k different primes. Also, let $\hat{Q}_j = \prod_{i\ne j} Q_i'$, and assume that $|P| \ge \max_{0\le j<d} Q_j'$. We employ a gadget vector $\boldsymbol{g} = (g_0, \cdots, g_{d-1}) \in \mathcal{R}_{Q_\ell}^{d}$ where $g_j = \hat{Q}_j[\hat{Q}_j^{-1}]_{Q_j'}$ for $0 \le j < d$.

The key-switching related procedure in ModHE can be redefined as the following:

- $\mathsf{KSGen}(\boldsymbol{s}_1, \boldsymbol{s}_2, \boldsymbol{g})$. Given the secret polynomial vectors $\boldsymbol{s}_1 \in \mathcal{R}^{r_1}$ and $\boldsymbol{s}_2 \in \mathcal{R}^{r_2}$, sample $\boldsymbol{A}' \leftarrow \mathcal{U}\left(\mathcal{R}_{PQ_L}^{r_1\times r_2}\right)$ and an error vector $\boldsymbol{e}' \leftarrow \chi_{\mathsf{err}}$. Output switching keys $\{\mathsf{swk}_j\}_{0\le j<d}$ where $\mathsf{swk}_j = \left([-\boldsymbol{A}'\boldsymbol{s}_2 + \boldsymbol{e}' + Pg_j \cdot \boldsymbol{s}_1]_{PQ_L}, \boldsymbol{A}'\right)^{\top} \in \mathcal{R}_{PQ_L}^{(1+r_2)\times r_1}$ for $0 \le j < d$.
- $\mathsf{KeySwitch}_d\left(\boldsymbol{c}, \mathsf{swk}_{(\boldsymbol{s}_1\to\boldsymbol{s}_2)}\right)$. Given a encrypted vector $\boldsymbol{c} = (c_0, \cdots, c_{r_1-1}) \in \mathcal{R}_{Q_\ell}^{r_1}$ and a switching key $\mathsf{swk} \leftarrow \mathsf{KeyGen}(\boldsymbol{s}_1, \boldsymbol{s}_2, \boldsymbol{g})$, perform the followings and output the ciphertext $\mathsf{ct} \in \mathcal{R}_{Q_\ell}^{1+r_2}$ satisfying that $[\mathsf{ct}[0] + \langle \mathsf{ct}[1], \boldsymbol{s}_2\rangle]_{Q_\ell} \approx [\langle \boldsymbol{c}, \boldsymbol{s}_1\rangle]_{Q_\ell}$. Let $d = \lceil(\ell+1)/k\rceil$.
  - a) **Decomposition**: Compute the gadget decomposition $\boldsymbol{C} = \{\boldsymbol{c}_0', \cdots, \boldsymbol{c}_{d-1}'\}$ where $\boldsymbol{c}_i' = [[\boldsymbol{c}]_{Q_i'}]_{PQ_\ell} \in \mathcal{R}_{PQ_\ell}^{r_1}$ for $0 \le i < d$.
  - b) **Inner Product**: Compute $\mathsf{ct}_{up} = \left[\sum_{i=0}^{d-1} \mathsf{swk}_i \cdot \boldsymbol{c}_i'\right]_{PQ_\ell}$.
  - c) **Modulus Down**: Compute and return $\mathsf{ct} = \lfloor P^{-1} \cdot \mathsf{ct}_{up}\rceil$.
- $\mathtt{Relin}(\mathsf{ct}_{\mathsf{ten}}, \mathsf{rlk})$. Given a ciphertext $\mathsf{ct}_{\mathsf{ten}} = (c_0, \boldsymbol{c}_1, \boldsymbol{c}_2) \in \mathcal{R}_{Q_\ell}^{1+r+\binom{1+r}{2}}$ and a relinearization key $\mathsf{rlk} \leftarrow \mathsf{KSGen}(\boldsymbol{s}\otimes\boldsymbol{s}, \boldsymbol{s}, \boldsymbol{g})$, output the ciphertext $\mathsf{ct} \leftarrow (c_0, \boldsymbol{c}_1) + \mathsf{KeySwitch}_d(\boldsymbol{c}_2, \mathsf{rlk})$.
- $\mathtt{Rotate}(\mathsf{ct}, \gamma, \mathsf{rotk}_\gamma)$. We define the automorphism $\phi_i : m(X) \mapsto m(X^{5^i})$. Given a ciphertext $\mathsf{ct} = (b, \boldsymbol{a}) \in \mathcal{R}_{Q_\ell}^{1+r}$ a rotation step γ, and a rotation key $\mathsf{rotk}_\gamma \leftarrow \mathsf{KSGen}(\boldsymbol{s}^{5^i}, \boldsymbol{s}, \boldsymbol{g})$, output the ciphertext $\mathsf{ct} \leftarrow (\phi_\gamma(b), 0) + \mathsf{KeySwitch}_d(\phi_\gamma(\boldsymbol{a}), \mathsf{rotk}_\gamma)$.

[1]To avoid excessive symbols, we also use d as the decomposition size for evaluation keys, which should ideally be $\lceil(L+1)/k\rceil$.

- $\mathsf{RdKGen}(\boldsymbol{s}, r_{\text{red}}, \boldsymbol{g})$. Given a secret polynomial vectors $\boldsymbol{s} \in \mathcal{R}^r$ and the number of reduced rank r_{red}, output a rank reduction key[2] $\mathsf{rdk} \leftarrow \mathsf{KSGen}(\boldsymbol{s}[r_{\text{rem}} : r], \boldsymbol{s}[0 : r_{\text{rem}}], \boldsymbol{g})$ where r_{rem} denotes $r - r_{\text{red}}$.
- $\mathsf{RankReduce}(\mathsf{ct}, \mathsf{rdk})$. Given a ciphertext $\mathsf{ct} = (b, \boldsymbol{a}) \in \mathcal{R}_{Q_\ell}^{1+r}$ and a rank reduction key $\mathsf{rdk} \leftarrow \mathsf{RdKGen}(\boldsymbol{s}, r - r_{\text{rem}}, \boldsymbol{g})$, output the ciphertext $\mathsf{ct}_{\text{red}} \leftarrow (b, \boldsymbol{a}[0 : r_{\text{rem}}]) + \mathsf{KeySwitch}_d(\boldsymbol{a}[r_{\text{rem}} : r], \mathsf{rdk}) \in \mathcal{R}_{Q_\ell}^{1+r_{\text{rem}}}$.

Notice that the key-switching method in the prior work ModHE is actually a special case when $k = 1$, so our consequent analysis does not lose generalization when making comparisons with the prior work. Also, when $r = 1$, it degenerates into a normal RLWE-based CKKS scheme. Therefore, the discussion around r can also cover the case of RLWE.

3.1.2 Complexity. In this section, we will analyze in detail how MLWE-based schemes face the tremendous performance drawback. We mainly focus on two major computational workloads: (I)NTT operations and Hadamard products. Table 1 shows the frequency of workload invocations by each sub-procedure. Here we assume that $k | (\ell + 1)$. For brevity, we denote $\ell + 1$ as $\tilde{\ell}$, and denote $\ell + k + 1$ as $\tilde{\ell}'$.

Table 1: Computational complexity of `HMult`. (I)NTT and Hadamard products are performed on the polynomials of degree n. For the asymptotic complexity, an additional multiplication with $O(n \log n)$ (for NTT) or $O(n)$ (for Hadamard products) is required.

Sub-procedure	# of (I)NTT	# of Hadamard Product	
		$k = 1$	$k > 1$
Tensor	-	$(r+1)^2\tilde{\ell}$	
Decomposition	$r(r+1)(\tilde{\ell} + d\tilde{\ell}')/2$	-	$r(r+1)\tilde{\ell}^2/2$
Inner Product	-	$r(r+1)^2 d\tilde{\ell}'/2$	
Modulus Down	$(r+1)(\tilde{\ell} + \tilde{\ell}')$	$(r+1)\tilde{\ell}$	$(r+1)(k+1)\tilde{\ell}$

We primarily explain the two overwhelming parts in this table: the (I)NTT operations in the decomposition step, and the Hadamard products in the inner product step. The former is derived from that RNS-based gadget decomposition has to be done in the CRT representation, while polynomial multiplication has to be done in the NTT representation. Therefore it takes $\binom{r+1}{2}\tilde{\ell}$ unit INTT operations and $\binom{r+1}{2}d\tilde{\ell}'$ unit NTT operations before and after the RNS-based decomposition. The latter, which is an arising problem brought by the module structure, is introduced by repeating the matrix-vector multiplications between $\mathcal{R}_{PQ_\ell}^{(r+1)\times\binom{r+1}{2}}$ and $\mathcal{R}_{PQ_\ell}^{\binom{r+1}{2}}$ for d times.

To compare the asymptotic complexity with RLWE-based schemes (when $r = 1$), we fix the dimension $N = r \cdot n$ and the circuit depth. Recall that in RLWE-based schemes, the NTT operations in the decomposition step often dominates the majority of the computation time. However, it should be noted that while the complexity of NTT operations in MLWE schemes is only $(r+1)/2$ times that of RLWE schemes, the workload of inner products will rapidly escalate with r and become the primary workload—since the complexity of inner products in MLWE is $(r+1)^2/4$ times that in RLWE.

This leads us to an even worse performance estimation: as r increases, the performance gap between MLWE and RLWE schemes will gradually approach $O(r^2)$, far worse than the previous estimate [36] of $(r+1)/2$ times.

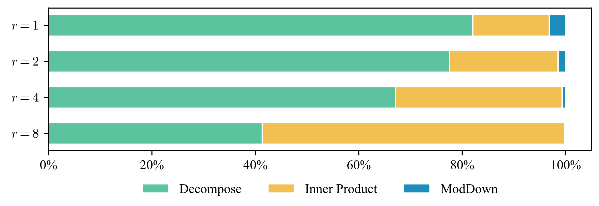


Figure 1: An empirical example of how the module rank affects the performance of MLWE-based homomorphic multiplication given $N = 2^{16}$.

3.2 A New Method with Rank Manipulation

According to the above analysis, Fig. 1 provides a breakdown of computational workload under typical ranks. As we can see, the core computational workload requiring optimization consists of NTT in the decomposition part and the subsequent inner product operations. Clearly, unlike the RLWE case, we must take into account the time consumption of the inner product operations and find ways to reduce their scale.

3.2.1 A initial attempt. At the outset, we aimed to address this issue directly at its source, that is, to avoid ciphertext expansion caused by `Tensor`. We noticed that HERMES [4] proposed a key-switching method by converting a rank-r MLWE ciphertext into RLWE, and exploiting the RLWE key-switching method. If we apply such a key-switching method after the `Tensor` operation, we would need to call RLWE key-switching procedure for at least $(r+1)/2$ times since `Tensor` introduces $r(r+1)/2$ additional polynomials. This implies that it would be at least $(r+1)/2$ times slower than RLWE schemes of the same dimension. A straightforward way to resolve this issue is to perform `Tensor` operation in the RLWE form, since only one additional polynomial needs relinearization. In fact, as is later explained in Section 4.2, in order to endow the converted RLWE ciphertext with multiplicative homomorphism, $\log r$ additional RLWE automorphisms should be evaluated. In this idea, MLWE-based multiplication is only about $(1 + \log r)$ times slower than RLWE schemes, slightly alleviating the aforementioned issue.

However, clearly the MLWE key-switching based on RLWE can never outperform RLWE schemes. More importantly, binding one of the fundamental operations, the homomorphic multiplication, to the RLWE setting goes against the hardware-friendly intention. As a result, we had to seek a more efficient solution for pure MLWE schemes, more specifically, a relinearization algorithm on the expanded MLWE ciphertexts after `Tensor`.

Fortunately, despite the unsatisfactory results of the exploration above, as a side-product of our work, it has provided interesting

[2]Note that the maximum level used in KSGen is not necessarily L. Instead, some $L' < L$ is expected to ensure the security on a reduced module rank r_{rem}.

properties and better versatility for MLWE-based FHE, offering more room for imagination. We will describe this aspect in detail in Section 4.

3.2.2 Rank flexibility for more possibility. As highlighted in [27, 29], the size of gadget decomposition is a primary factor affecting the complexity of KeySwitch. By using more special moduli, we can reduce the decomposition size and effectively lower the computational complexity, at the cost of sacrificing available multiplicative depths. However, in practical applications, the number of special moduli is often set relatively conservatively, typically a small constant, to maintain as many evaluation depths as possible.

If we wish to aggressively decrease the decomposition size by engaging more special moduli, we would have to increase the dimension to maintain security, which is a common issue for both RLWE and MLWE-based schemes. In the situation of MLWE, increasing either n or r is not a wise option: increasing n contradicts the hardware-friendly feature, while increasing r exacerbates the ciphertext expansion brought by Tensor.

Our breakthrough stems from a key insight: compared to RLWE schemes, *MLWE schemes offer high flexibility on module rank.* This flexibility allows us to either elevate the ciphertext's rank by simply appending zero polynomials or reduce its rank by applying a key-switching-like procedure. This key insight reveals that we can have the best of both worlds: conducting the low-complexity relinearization with a small decomposition size, and keeping the same security without altering n or r, bridged by a temporary rank $\hat{r} > r$ to ensure security for both side.

Our relinearization algorithm is quite elegant in concept and can be described in two major steps:

(1) Relinearize the ciphertext with a minimal decomposition size, denoted by a small constant μ, resulting in a ciphertext with rank $\hat{r} > r$;
(2) Reduce the additional rank with a key-switching procedure of normal decomposition size d.

As shown in Fig. 2, the first step is meant to apply a low-complexity key-switching procedure to the expanded ciphertext. To maintain security under a larger set of special moduli brought by the small μ, we temporarily increase the module rank to $\hat{r}$ during relinearization. However, it's important to note that we're not employing the method of adding zero polynomials here; instead, we use a cross-relinearization key to perform relinearization while simultaneously elevating rank. The second step is meant to remove the temporary rank to compress the ciphertext size and to reduce ciphertext expansion when performing Tensor in the future.

To perform such an algorithm, some extra keys should be generated at the preparation phase. First, the KeyGen function should generate an extra vector of secret polynomials $\boldsymbol{s}' \in \mathcal{R}_n^{\hat{r}-r}$ as a part of the temporary secret key for the intermediate ciphertext during the relinearization. Second, the KSGen function should publish a cross-relinearization key crlk and a rank-down[3] key rdk respectively:

- The cross-relinearization key crlk: This key is intended to relinearize the polynomials after Tensor, with the assistance of a new gadget vector $\hat{\boldsymbol{g}} = (\hat{g}_0, \cdots, \hat{g}_{\mu-1})$ of small constant size μ. By introducing a temporary special modulus $\hat{P} = \prod_{i=0}^{K} p_i > P$, we denote $\hat{g}_j$ by $\check{Q}_j[\check{Q}_j^{-1}]_{Q''_j}$ where $Q''_j = \prod_{i=jK}^{\min((j+1)K-1,\ell)}$ and $\check{Q}_j = \prod_{i \neq j} Q''_j$ for $0 \leq j < \mu$. To maintain the similar security for keys of modulus $\hat{P}Q_L$, we employ a temporary rank $\hat{r} > r$. After sampling $\boldsymbol{A}_{\mathsf{crlk}_j} \leftarrow \mathcal{U}\left(\mathcal{R}_{\hat{P}Q_L}^{\binom{1+r}{2} \times \hat{r}}\right)$ and $\boldsymbol{e}_j \leftarrow \chi_{\mathrm{err}}^{\binom{1+r}{2}}$, $\mathsf{crlk} = \{\mathsf{crlk}_j\}_{0 \leq j < \mu}$ is generated by calculating $\mathsf{crlk}_j = (\boldsymbol{b}_{\mathsf{crlk}_j}, \boldsymbol{A}_{\mathsf{crlk}_j})^\top \in \mathcal{R}_{\hat{P}Q_L}^{(1+\hat{r}) \times \binom{1+r}{2}}$ where $\boldsymbol{b}_{\mathsf{crlk}_j} = -\boldsymbol{A}_{\mathsf{crlk}_j} \cdot (\boldsymbol{s} \| \boldsymbol{s}') + \boldsymbol{e}_j + \hat{P} \cdot \hat{g}_j \cdot (\boldsymbol{s} \otimes \boldsymbol{s}) \bmod \hat{P}Q_L$, for $j = 0, 1, \cdots, \mu - 1$.
- The rank-down key rdk: This key is intended to restore the original module rank with normal gadget decomposition $\boldsymbol{g}$ of decomposition size d, thus has small temporary modulus $P = \prod_{i=0}^{k} p_i$ and the original rank r. After sampling $\boldsymbol{A}_{\mathsf{rdk}_j} \leftarrow \mathcal{U}\left(\mathcal{R}_{PQ_L}^{(\hat{r}-r) \times r}\right)$ and $\boldsymbol{e}_j \leftarrow \chi_{err}^{r}$, $\mathsf{rdk} = \{\mathsf{rdk}_j\}_{0 \leq j < d}$ is an element in $\left(\mathcal{R}_{PQ_L}^{(1+r) \times (\hat{r}-r)}\right)^d$ where $\mathsf{rdk}_j = (\boldsymbol{b}_{\mathsf{rdk}_j}, \boldsymbol{A}_{\mathsf{rdk}_j})^\top$ and $\boldsymbol{b}_{\mathsf{rdk}_j} = -\boldsymbol{A}_{\mathsf{rdk}_j} \cdot \boldsymbol{s} + \boldsymbol{e}_j + P \cdot g_j \cdot \boldsymbol{s}' \bmod PQ_L$, for $j = 0, 1, \cdots, d - 1$.

The detailed algorithm is described in Algorithm 1.

Algorithm 1 DPad-HE.Relin

```
Input: Ciphertext $\mathsf{ct} = (c_0, \boldsymbol{c}_1, \boldsymbol{c}_2) \in \mathcal{R}_{Q_l}^{1+r+\binom{1+r}{2}}$, cross-relinearization key $\mathsf{crlk} \in \left(\mathcal{R}_{\hat{P}Q_L}^{(1+\hat{r}) \times \binom{1+r}{2}}\right)^{\mu}$, rank-down key $\mathsf{rdk} \in \left(\mathcal{R}_{PQ_L}^{(1+r) \times (\hat{r}-r)}\right)^{d}$
Output: Ciphertext $\mathsf{ct}' = (c'_0, \boldsymbol{c}_1{}') \in \mathcal{R}_{Q_l}^{1+r}$
1: $(\hat{c}_0, \hat{\boldsymbol{c}}_1) \leftarrow \mathsf{KeySwitch}_{\mu}(\boldsymbol{c}_2, \mathsf{crlk})$    ▷ in $\mathcal{R}_{Q_\ell}^{1+\hat{r}}$
2: $(\check{c}_0, \check{\boldsymbol{c}}_1) \leftarrow \mathsf{KeySwitch}_{d}(\hat{\boldsymbol{c}}_1[r:\hat{r}], \mathsf{rdk})$    ▷ in $\mathcal{R}_{Q_\ell}^{1+r}$
3: $\mathsf{ct}' \leftarrow (c_0 + \hat{c}_0 + \check{c}_0, \boldsymbol{c}_1 + \hat{\boldsymbol{c}}_1[0:r] + \check{\boldsymbol{c}}_1)$
4: return $\mathsf{ct}'$
```

3.3 Complexity Comparison

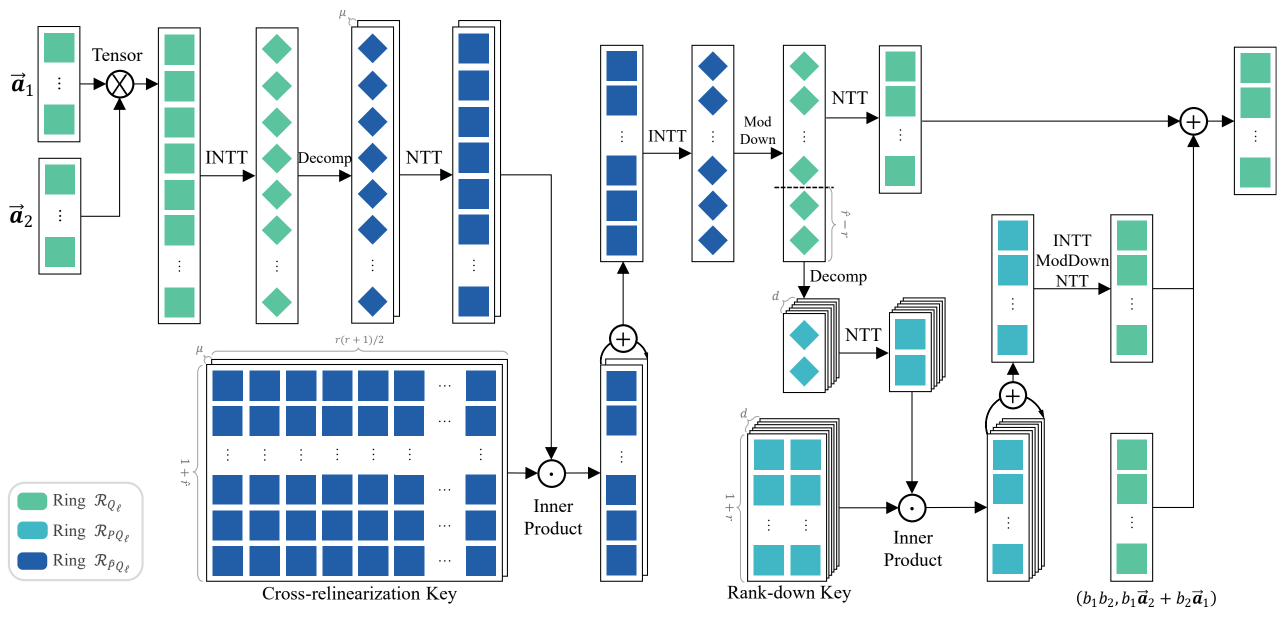
In this section, we will analyze the time complexity as well as the space complexity of our proposed algorithm. Aside from the analysis focusing on the dominant computational workload, we provide an asymptotic time complexity comparison to outline our advantage intuitively.

Note that $\hat{r} \leq 2r$, where the maximum value of $\hat{r}$ is attained when $\mu = 1$, i.e., without using any gadget decomposition in the first stage.

3.3.1 Time Complexity. For the analysis to come, we denote $\ell + 1$ as $\tilde{\ell}$, denote $\ell + k + 1$ as $\tilde{\ell}'$, and denote $\ell + K + 1$ as $\tilde{\ell}''$, for brevity.

In the first stage, we relinearize the tensored part $\boldsymbol{c}_2$ by switching its key from $\boldsymbol{s} \otimes \boldsymbol{s}$ to $\boldsymbol{s} \| \boldsymbol{s}'$. In the decomposition step, it takes $\binom{r+1}{2}(\tilde{\ell} + \mu\tilde{\ell}'')$ (I)NTT operations and $\binom{r+1}{2}\tilde{\ell}^2$ Hadamard products.

[3]Here, we distinguish it from rank reduction to emphasize their different functionalities and use cases.



Tensor
INTT
Decomp
NTT
Mod Down
Inner Product
Cross-relinearization Key
Rank-down Key
INTT ModDown NTT
Ring $\mathcal{R}_{Q_\ell}$
Ring $\mathcal{R}_{PQ_\ell}$
Ring $\mathcal{R}_{\hat{P}Q_\ell}$
$(b_1 b_2, b_1\vec{a}_2 + b_2\vec{a}_1)$

Figure 2: DPad-HE.HMult procedure. Each colored square or diamond represents a polynomial. The green indicates the polynomial is in $\mathcal{R}_{Q_\ell}$, the ultramarine indicates it's in $\mathcal{R}_{\hat{P}Q_\ell}$, and the turquoise indicates it's in $\mathcal{R}_{PQ_\ell}$. Squares represent the polynomial in NTT form, and diamonds represent the polynomial in CRT form.

In the inner product step, it takes $\frac{r(r+1)(\hat{r}+1)}{2}\mu\tilde{\ell}''$ Hadamard products when performing matrix-vector multiplications between $\mathcal{R}_{\hat{P}Q_\ell}^{(\hat{r}+1)\times\binom{r+1}{2}}$ and $\mathcal{R}_{\hat{P}Q_\ell}^{\binom{r+1}{2}}$ for μ times.

In the second stage, we restore the module rank from $\hat{r}$ to the original r. In the decomposition step, as is depicted in Fig 2, we skip the first INTT operations, therefore requiring $(\hat{r}-r)d\tilde{\ell}'$ unit NTT operations and $(\hat{r}-r)\tilde{\ell}^2$ unit Hadamard products (if $k>1$). In the inner product step, it takes $(r+1)(\hat{r}-r)d\tilde{\ell}'$ unit Hadamard operations when performing matrix-vector multiplications between $\mathcal{R}_{PQ_\ell}^{(r+1)\times(\hat{r}-r)}$ and $\mathcal{R}_{PQ_\ell}^{\hat{r}-r}$ for d times.

To provide an intuitive cognition of our algorithm, we rephrase the complexity analysis in an asymptotic manner. Since $k\in O(1)$ and $K\le\ell+1$, we deduce $\tilde{\ell},\tilde{\ell}',\tilde{\ell}''\in O(\ell)$. Since $d=\lceil(\ell+1)/k\rceil$, we have $d\in O(\ell)$. Since $\hat{r}\le 2r$, we deduce $\hat{r}\in O(r)$.

As is shown in Table 2, our algorithm's concept lies in offsetting the $O(r^2)$ expansion caused by Tensor and the $O(\ell^2)$ expansion resulting from the gadget decomposition. Consequently, our relinearization method provides a considerable speedup over the prior schemes based on MLWE, especially when the multiplicative depth is large or the module rank is large. The former situation is common since an FHE cryptosystem is more often to evaluate a large circuit with a sufficiently large L, for example in the usage of machine learning. Note that when r is set to a modest value and ℓ is sufficiently large, our algorithm is likely to outperform RLWE-based schemes, since we have better asymptotic complexity in the relatively expensive NTT operations, while the complexity of Hadamard products has not expanded to much. We must stress that these estimates are quite approximate as they neglect lower-order terms.

Table 2: Asymptotic computational complexity comparison of RNS-CKKS, the prior work and our `HMult` methods. Here, NTT and Hadamard Product indicate the corresponding word-size operations.

Methods	(I)NTT	Hadamard Product
RNS-CKKS [11, 27]	$O(\ell^2 N\log N)$	$O(\ell^2 N)$
ModHE [36]	$O(r\ell^2 N\log n)$	$O(r^2\ell^2 N)$
DPad-HE	$O((r+\ell)\,\ell N\log n)$	$O((r\ell^2+r^2\ell)\,N)$

3.3.2 *Space Complexity.*

In regard to the space complexity, we mainly discuss the size of the relinearization key. In the prior work, the relinearization key is a size-d set of $(r+1)\times\binom{r+1}{2}$ matrix in $\mathcal{R}_{PQ_L,n}$, of which the bit-size is $\frac{r(r+1)^2dn}{2}\log PQ_L$. In our algorithm, we use two key-switching keys. The first key `crlk` is a size-μ set of $(\hat{r}+1)\times\binom{r+1}{2}$ matrix in $\mathcal{R}_{\hat{P}Q_L,n}$, and the second key rdk is a size-d set of $(r+1)(\hat{r}-r)$ matrix in $\mathcal{R}_{PQ_L,n}$. The overall space complexity is $\frac{r(r+1)(\hat{r}+1)n\cdot\mu}{2}\log\hat{P}Q_L+(\hat{r}-r)(r+1)dn\log PQ_L$. So the ratio between ours and the prior work is approximately

$$\frac{2(\hat{r}-r)}{r(r+1)}+\frac{(\hat{r}+1)k\mu}{(r+1)(L+1)}\le\frac{2}{r+1}+\frac{2k\mu}{L+1}$$

as $\hat{r}\le 2r$. Since k,μ are small constants and $r\ge 2$ in the MLWE setting, our relinearization key-pair is expected to consume less space compared to the prior work, especially when the circuit depth is sufficiently large.

3.4 Noise Estimations

In this section, we will provide an upper bound of the noise introduced by our DPad-HE.Relin algorithm. Compared to the naive Relin algorithm, we will demonstrate that our algorithm exhibits smaller noise growth in common scenarios.

We first bound the noise of our generalized key-switching subprocedure that switches a vector of polynomials $\boldsymbol{c} \in \mathcal{R}_{Q_\ell}^{r_1}$ to a ciphertext $(b, \boldsymbol{a}) \in \mathcal{R}_{Q_\ell}^{1+r_2}$ satisfying $b + \langle \boldsymbol{a}, \boldsymbol{s}_2 \rangle \approx \langle \boldsymbol{c}, \boldsymbol{s}_1 \rangle$, with the help of a switching key $\mathsf{swk}_{(s_1 \to s_2)}$.

Lemma 3.1. *For the generalized MLWE key-switching algorithm described in §3.1.1, we have the following heuristic bound for key-switching noise:*

$$\|e_{KS}\|_\infty^{can} \le \frac{8 \cdot d \cdot \sigma r_1 n}{\sqrt{3}P} \max_{0 \le i < d} Q'_i + \sqrt{3n} + \frac{8\sqrt{2} r_2 n}{3}$$

Lemma 3.2. *For our DPad-HE.*Relin *algorithm Alg.1, we have the following heuristic bound for our relinearization noise:*

$$\|e_{DPad.Relin}\|_\infty^{can} \le \frac{4 \cdot \mu \cdot \sigma r(r+1) n}{\sqrt{3}\hat{P}} \max_{0 \le i < \mu} Q''_i + \frac{8 \cdot d \cdot \sigma(\hat{r} - r) n}{\sqrt{3}P} \max_{0 \le i < d} Q'_i + 2\sqrt{3n} + \frac{8\sqrt{2}(\hat{r} + r) n}{3}$$

For the detailed proofs of the above lemmas, please see Appendix A.1.1 and A.1.2. For ModHE, we can derive its relinearization noise bound by applying Lemma 3.1:

$$\|e_{\mathsf{ModHE.Relin}}\|_\infty^{\mathsf{can}} \le \frac{4 \cdot d \cdot \sigma r(r+1) n}{\sqrt{3}P} \max_{0 \le i < d} Q'_i + \sqrt{3n} + \frac{8\sqrt{2} r n}{3}$$

In the common situation when $P \ge \max_{0 \le i < d} Q'_i$ and $\hat{P} \ge \max_{0 \le i < \mu} Q''_i$, our noise bound holds an advantage of

$$B_{\mathsf{ModHE.Relin}} - B_{\mathsf{DPad.Relin}} \approx \frac{4}{3} n \left(\sqrt{3}(d - \mu) r(r+1) - 2\sqrt{2}\hat{r} \right) \ge \frac{4}{\sqrt{3}} n r^2 \left((d - \mu)(1 + r^{-1}) - 4\sqrt{6}/3 \right)$$

since $\hat{r} \le 2r$. Because d is grown by ℓ, while μ is a small constant, our algorithm achieves better noise growth most of the time.

3.5 Rank Hoisting for Rotation

With regard to the homomorphic rotation, a unique key should be generated for each distinct rotation step. However, the possible rotation steps are equivalent to the number of slots packed, which is generally a large number, leading to a significantly large key size. In some homomorphic libraries such as SEAL [42] and HEaaN [16], the key-generation function only generates keys for the rotations with all the power-of-two steps. When a certain step of HRotate operation is needed, they represent the step in non-adjacent form (NAF) [40] and perform multiple power-of-two rotations to achieve the desired step. This comes at a trade-off between the increased rotation time and the reduced key size.

In such a situation, we note that MLWE-based schemes hold an advantage in decreasing the overall rotation time. Remember that in the DPad-HE.Relin algorithm, we elevate the ciphertext to a temporary rank $\hat{r}$, where a minimal gadget decomposition can be used when performing the key-switching procedure. We can port this idea to the rotation operation, especially when a series of rotations are applied to a single ciphertext consecutively.

Note that in the rotation procedure, key-switching with rank $\hat{r}$ is faster than that with r, since there is less ciphertext expansion. Our proposed *rank hoisting* method is outlined as follows: when consecutive rotations are to be performed, we can elevate the module rank of the ciphertext to a temporary $\hat{r}$ to perform these key-switching procedure at relatively low cost; after the last rotation is done, we apply the rank-down procedure to restore the initial rank. Note that unlike the *hoisting* algorithm proposed by [8, 26], which is used to accelerate multiple rotations on a single ciphertext $(\mathtt{rot}_{\gamma_1}(\mathrm{ct}), \cdots, \mathtt{rot}_{\gamma_m}(\mathrm{ct}))$, our algorithm aims to speed up consecutive rotations on the same ciphertext $(\mathtt{rot}_{\gamma_m} \circ \cdots \circ \mathtt{rot}_{\gamma_1}(\mathrm{ct}))$.

Our algorithm for improving the consecutive rotation is described in Algorithm 2.

Algorithm 2 Rank hoisted rotation DPad-HE.Rotate

Input: Ciphertext $\mathrm{ct} = (b, \boldsymbol{a}) \in \mathcal{R}_{Q_l}^{1+r}$, rotation keys $\mathrm{rotk}_i \leftarrow \mathsf{KSGen}((\boldsymbol{s}\|\boldsymbol{s}')^{5^i}, \boldsymbol{s}\|\boldsymbol{s}', \hat{\boldsymbol{g}}) \in \left(\mathcal{R}_{\hat{P}Q_L}^{(1+\hat{r})\times(1+\hat{r})}\right)^\mu$ where i is the power-of two, rank-down key $\mathrm{rdk} \in \left(\mathcal{R}_{PQ_L}^{(1+r)\times(\hat{r}-r)}\right)^d$, rotation step γ.

Output: Ciphertext ct'.

```
1: $\widehat{\mathrm{ct}} = (\hat{b}, \hat{\boldsymbol{a}}) \leftarrow (b, \boldsymbol{a}\|\boldsymbol{0})$        ▷ in $\mathcal{R}_{Q_\ell}^{1+\hat{r}}$
2: for $i \in \mathsf{NAF}(\gamma)$ do
3:     $\widehat{\mathrm{ct}} = (\hat{b}, \hat{\boldsymbol{a}}) \leftarrow (\phi_i(\hat{b}), \boldsymbol{0}) + \mathsf{KeySwitch}_\mu(\phi_i(\hat{\boldsymbol{a}}), \mathrm{rotk}_i)$
4: end for
5: $\mathrm{ct}' \leftarrow (\hat{b}, \hat{\boldsymbol{a}}[0:r]) + \mathsf{KeySwitch}_d(\hat{\boldsymbol{a}}[r:\hat{r}], \mathrm{rdk})$        ▷ in $\mathcal{R}_{Q_\ell}^{1+r}$
6: return $\mathrm{ct}'$
```

4 Lightweight Conversion between MLWE and RLWE schemes

As we mentioned in Section 3.2, the initial exploration of key-switching by converting MLWE ciphertexts into RLWE ciphertexts led to a side-product: through the mutual conversion of MLWE and RLWE, we can extend the advantages of DPad-HE to RLWE schemes. For instance, we can generalize rank reduction from DPad-HE to RLWE, thereby achieving ring switching [25] more simply. Alternatively, we can flexibly choose between MLWE and RLWE throughout the entire evaluation process to achieve better implementation efficiency.

Of course, all the above rely on sufficiently efficient conversion between MLWE and RLWE schemes. This section will introduce an efficient conversion scheme and its potential application in DPad-HE.

4.1 Fast Conversion in the NTT form

The conversion algorithm of coefficients-encoding ciphertext between MLWE and RLWE was first proposed by HERMES [4], which consists of several mappings that rearrange the polynomial coefficients. This operation naturally requires processing in the coefficient representation, thereby necessitating additional NTT/INTT transformations. This undoubtedly greatly increases the computational burden. In this section, we propose a method to perform this conversion in the NTT form.

Our algorithm is built on the concept of Embed and Extract algorithms in [4, §4.2]. We use the map $\iota : \mathcal{R}_{q,n} \to \mathcal{R}_{q,N}$ to embed a polynomial $a(X)$ into $a(X^r)$. We define the map $\epsilon : \mathcal{R}_{q,N} \to \mathcal{R}_{q,n}$ as $\epsilon(a(X)) = \sum_{i\in[n]} a_{ri} \cdot X^i$ to extract the coefficient of monomial X^i that r divides i. We use the map $\pi : \mathcal{R}_{q,N} \to \mathcal{R}^r_{q,n}$ to decompose a single polynomial $a(X) = \sum_{i\in[N]} a_i \cdot X^i$ into a rank-r module $(a'_0(X), \cdots, a'_{r-1}(X))$, where $a'_j(X) = \sum_{i\in[n]} a_{ri+j} \cdot X^i \in \mathcal{R}_{q,n}$. We use the map $\rho : \mathcal{R}^r_{q,n} \to \mathcal{R}^r_{q,n}$ to twist a rank-r module $(a_0(X), \cdots, a_{r-1}(X))$ into $(a_0(X), a_{r-1}(X) \cdot X^{-1}, a_{r-2}(X) \cdot X^{-1}, \cdots, a_1(X) \cdot X^{-1})$. Note that π and ρ are bijections.

4.1.1 Embed. The embedding map proposed by [4] converts a single MLWE ciphertext in $\mathcal{R}^{1+r}_{q,n}$ into an RLWE ciphertext $\mathcal{R}^2_{q,N}$ where $N = rn$. Here we made some modifications to adapt it to our subsequent acceleration algorithm. It is defined as

$$\texttt{Embed}(b, \boldsymbol{a}) = \left(\iota(b), \pi^{-1}(\boldsymbol{a})\right) \tag{1}$$

satisfying that

$$b + \langle \boldsymbol{a}, \boldsymbol{s}\rangle = \epsilon\left(\texttt{Embed}(b, \boldsymbol{a}) \cdot \left(1, \pi^{-1} \circ \rho(\boldsymbol{s})\right)\right)$$

Note that the ι and π^{-1}-mapping involve the reorganization of the coefficients in polynomials, which lead to additional conversions between the NTT representation and the CRT representation. Now we propose our algorithm to perform these mappings in the NTT form.

Starting from the simple one. For the ι-mapping, we define $\tilde{b}_k = \texttt{NTT}_n(b)_k$ for $k = 0, \cdots, n-1$. We define $b'(X) = \iota(b)$ and $\tilde{b}'_K = \texttt{NTT}_N(b'(X))_K$ for $K = 0, \cdots, N-1$. It holds that

$$\tilde{b}'_K = \sum_{i=0}^{N-1} b'_i \psi_{2N}^{i(2K+1)} = \tilde{b}_{[K]_n}$$

Therefore, the ι-mapping in the NTT form can be performed by simply copying elements in $\tilde{b}$.

Theorem 4.1 (π^{-1}-mapping in the NTT form). *Let $a'(X) \in \mathcal{R}_{q,N}$ be $\pi^{-1}(\boldsymbol{a})$ where $\boldsymbol{a} = (a_0(X), \cdots, a_{r-1}(X)) \in \mathcal{R}^r_{q,n}$. We denote by $\tilde{a}_{j,k} = \texttt{NTT}_n(a_j(X))_k$ the k-th element in the NTT representation of $a_j(X)$ for $0 \le k < n$ and $0 \le j < r$. We denote by $\tilde{a}'_K = \texttt{NTT}_N(a'(X))_K$ the K-th element in the NTT representation of $a'(X)$. It holds that*

$$\tilde{a}'_{\alpha n+\beta} = \sum_{v=0}^{r-1} \left(\tilde{a}_{v,\beta} \psi_{2N}^{v(2\beta-n+1)}\right) \psi_{2r}^{v(2\alpha+1)} \tag{2}$$

where $K = \alpha n + \beta$ for $0 \le \alpha < r$ and $0 \le \beta < n$.

By applying Theorem 4.1, the π^{-1}-mapping in the NTT form can be implemented by performing Hadamard products with some precomputed scalars and then performing n independent r-point NTT. We present the detailed algorithm in Algorithm 3.

In summary, the complexity of our proposed Embed algorithm is only $O(rn \log r + rn) = O(N \log 2r)$, while the additional (I)NTT operations require a complexity of $O((1 + r)n \log n + 2N \log N) = O(N \log \frac{N^3}{r} + n \log n)$.

Algorithm 3 π^{-1}-mapping in the NTT form

```
Input: $\tilde{\boldsymbol{a}} = (\tilde{a}_0, \cdots, \tilde{a}_{r-1}) \in$ where $\tilde{a}_j = \texttt{NTT}_n(a_j(X))$ for $j = 0, \cdots, r-1$. $r$ is a power-of-two integer.
Output: $\tilde{a}' = \texttt{NTT}_N \circ \pi^{-1}(\boldsymbol{a})$
1: for $k = 0$ to $n-1$ do
2:     temp $\leftarrow (\tilde{a}_{0,k}, \tilde{a}_{1,k}, \cdots, \tilde{a}_{r-1,k})$
3:     temp $\leftarrow$ temp $* \left(\psi_{2N}^{0\times(2k-n+1)}, \cdots, \psi_{2N}^{(r-1)\times(2k-n+1)}\right)$
4:     $(\tilde{a}'_{0\times n+k}, \cdots, \tilde{a}'_{(r-1)\times n+k}) \leftarrow \texttt{NTT}_r(\text{temp})$
5: end for
6: return $\tilde{a}'$
```

4.1.2 Extract. The extracting map performs the inverse operation of the embedding map, which is defined as

$$\texttt{Extract}(b', a') = \left(\epsilon(b'), \pi(a')\right) \tag{3}$$

satisfying that

$$\epsilon(b' + a's) = \texttt{Extract}(b', a') \cdot \left(1, \rho^{-1} \circ \pi(s)\right)$$

We can perform a similar technique to accelerate the π-mapping since it's the inverse of π^{-1}-mapping.

Theorem 4.2 (π-mapping in the NTT form). *Let $\boldsymbol{a} = (a_0(X), \cdots, a_{r-1}(X)) \in \mathcal{R}^r_{q,n}$ be $\pi(a'(X))$ where $a'(X) \in \mathcal{R}_{q,N}$. We denote by $\tilde{a}'_K = \texttt{NTT}_N(a'(X))_K$ the K-th element in the NTT representation of $a'(X)$. We denote by $\tilde{a}_{j,k} = \texttt{NTT}_n(a_j(X))_k$ the k-th element in the NTT representation of $a_j(X)$ for $0 \le k < N$ and $0 \le j < r$. It holds that*

$$\tilde{a}_{j,k} = \psi_{2N}^{-j(2k-n+1)} \cdot r^{-1} \sum_{v=0}^{r-1} \tilde{a}'_{vn+k} \psi_{2r}^{-j(2v+1)}. \tag{4}$$

By applying Theorem 4.2, the π-mapping in the NTT form can be implemented by performing n independent r-point NTT and then performing Hadamard products with some precomputed scalars. We present the detailed algorithm in Algorithm 4.

Algorithm 4 π-mapping in the NTT form

```
Input: $\tilde{a}' = \texttt{NTT}_N(a'(X))$.
Output: $\tilde{\boldsymbol{a}} = (\tilde{a}_0, \cdots, \tilde{a}_{r-1}) = \texttt{NTT}_n \circ \pi(a'(X)) \in$ where $\tilde{a}_j = \texttt{NTT}_n(a_j(X))$ for $j = 0, \cdots, r-1$. $r$ is a power-of-two integer.
1: for $k = 0$ to $n-1$ do
2:     temp $\leftarrow \texttt{INTT}_r\left(\tilde{a}'_{0\times n+k}, \tilde{a}'_{1\times n+k}, \cdots, \tilde{a}'_{(r-1)\times n+k}\right)$
3:     temp $\leftarrow$ temp $* \left(\psi_{2N}^{-0\times(2k-n+1)}, \cdots, \psi_{2N}^{-(r-1)\times(2k-n+1)}\right)$
4:     $(\tilde{a}_{0,k}, \tilde{a}_{1,k}, \cdots, \tilde{a}_{r-1,k}) \leftarrow$ temp
5: end for
6: return $\tilde{\boldsymbol{a}}$
```

Unfortunately, for the ϵ-mapping, we have not figured out an algorithm that can perform the conversion in the NTT form, as the mapping is irreversible.

In summary, the complexity of our proposed Extract algorithm is $O(rn \log r + rn + n \log n + N \log N) = O(N \log 2Nr + n \log n)$, while the additional (I)NTT operations require a complexity of $O(N \log \frac{N^3}{r} + n \log n)$.

4.2 Coefficients-encoding to Slots-encoding

It should be noted that the RLWE ciphertext output by Embed is coefficients-encoding, which requires an additional $\iota \circ \epsilon$ mapping to attain the correct encryption of the original sparsely-encoded message, therefor preventing the ciphertext from the subsequent homomorphic evaluations. In order to preserve the homomorphic capacity of the RLWE ciphertext, we need to transform it into the slots-encoding form. Different from packing several MLWE ciphertexts into one RLWE ciphertext, our task only involves the conversion between single ciphertexts, so we only need to homomorphically evaluate the $\iota \circ \epsilon$ mapping.

For the evaluation of $\iota \circ \epsilon$ mapping which zeroizes certain coefficients in the underlying RLWE plaintext, we take a similar approach used in [10, Alg. 2]. With all the building blocks, we derive the MLWE-to-RLWE conversion procedure defined as

$$\mathtt{Eval}_{\iota\circ\epsilon} \circ \mathtt{Embed},$$

which converts an MLWE ciphertext encrypting $m(X)$ into an RLWE ciphertext encrypting $m(X^r)$.

Generally speaking, we achieve the MLWE-to-RLWE conversion by applying one fast Embed and $\log r$ RLWE rotations, which is relatively lightweight. For the inverse conversion, we just need to employ the fast Extract mapping on the sparsely-packed RLWE ciphertext.

Algorithm 5 Homomorphic Evaluation of $\iota \circ \epsilon$ mapping ($\mathtt{Eval}_{\iota\circ\epsilon}$)

```
Input: ct ∈ $\mathcal{R}^2_{q,N}$ encrypting $m(X)$.
Output: ct′ ∈ $\mathcal{R}^2_{q,N}$ encrypting $\iota \circ \epsilon(m(X))$.
1: ct′ ← ct
2: for $k = 0$ to $\log r - 1$ do
3:     ct′ ← ct′ + RLWE.Rotate(ct, $2^{\log n + k - 1}$)
4: end for
5: return ct′
```

4.3 Potential Application in DPad-HE

The conversion algorithm can directly benefit DPad-HE, and we briefly introduce two potential applications.

Scheme flexibility. As described earlier and in subsequent evaluations, there is no absolute superiority between MLWE and RLWE schemes. They each have their own advantages and disadvantages depending on different parameter settings and evaluation stages. Moreover, their performance can vary on different hardware platforms (since DPad-HE is more hardware-friendly). With the lightweight conversion, we can pre-evaluate the performance profiles of RLWE and MLWE under various conditions. Then, we can dynamically choose the optimal software/hardware implementation within each layer range, facilitated by the conversion algorithm.

A novel ring-switching method for RLWE. Ring switching was first introduced in [9] and further studied in [25], which allows conversions between rings of different degrees to speed up the homomorphic operations for the lower levels of the circuit. Note that the RankReduce operation in MLWE schemes serves a similar purpose. Therefore, we can use the following procedure

$$\mathtt{Eval}_{\iota\circ\epsilon} \circ \mathtt{Embed} \circ \mathtt{RankReduce} \circ \mathtt{Extract}$$

to perform such a task, by combining DPad-HE and the conversion algorithm.

5 Evaluation

We implemented DPad-HE on both CPU and GPU platforms and rewrote the scheme in ModHE for comparison, which was originally released as a proof-of-concept implementation in SageMath. Our code is developed upon the C++ SEAL [42] library. For a fair comparison, we used the same API for RLWE implementation as for MLWE (RLWE being a special case of MLWE when $r = 1$), and did not use the highly optimized version from the latest SEAL library.

In the GPU implementation, we developed upon 100x [29] and utilized the same (I)NTT implementations and kernel fuse technology.

All the experiments are conducted on an NVIDIA RTX 4090 GPU with CUDA 12.1 and an Intel Xeon Silver 4410Y @2.0GHz CPU with 128GB of RAM running Ubuntu 20.04 LTS. The experiments implemented on the CPU are conducted single-threaded. All the security estimates derive from [2, 3].

Although our scheme and the corresponding complexity analysis apply to an arbitrary number of special moduli k, we fix $k = 1$ for all the consequent experiments for ease of comparison with ModHE.

Our evaluation answers the following research questions.

RQ1: How much does the DPad-HE outperform the SOTA MLWE-based FHE with different rank r in around 128-bit security?

This experiment is designed to demonstrate the performance pattern of choosing a fixed base ring and then building upon the associated module rank depending on the desired circuit depth. In this experiment, we fix the polynomial degree n, and incrementally increase r from 1 to 8. The detailed parameter sets for MLWE-based schemes are provided in Table 3.

Table 3: Parameters with the same polynomial degree for RQ1. Here $\hat{r}$ refers to the rank of our cross-relinearization key and $\hat{\lambda}$ refers to the security of the context after rank-up. $\log_2 q$ denotes the bit-sizes of prime factors in the modulus chains, $\log_2 P$ denotes the size of the special modulus for $\mathtt{KeySwitch}_d$, and $\log_2 \hat{P}$ denotes the size of the special modulus for $\mathtt{KeySwitch}_\mu$.

n	r	L	$\hat{r}$	$\log_2 q$	$\log_2 P$	$\log_2 \hat{P}$	λ	$\hat{\lambda}$
2^{13}	2	8	3	$60 + 40 \times 8$	60	$60 + 55 \times 4$	125.4	126.0
	3	13	4	$60 + 40 \times 13$		$60 + 55 \times 4$	130.1	129.2
	4	19	5	$60 + 40 \times 19$		$60 + 55 \times 4$	126.2	126.5
	5	25	7	$60 + 40 \times 25$		$60 + 55 \times 8$	124.2	125.3
	6	30	8	$60 + 40 \times 30$		$60 + 55 \times 7$	126.7	131.3
	7	35	9	$60 + 40 \times 35$		$60 + 55 \times 8$	128.6	128.7
	8	41	10	$60 + 40 \times 41$		$60 + 55 \times 8$	127.1	127.5

Table 4 presents the performance comparison between ModHE and DPad-HE. Compared to ModHE, DPad-HE reaches a speedup of 5.7× with $r = 8, L = 41$, and $\log_2 QP = 1760$. For a modest setting with $r = 4$, DPad-HE demonstrates a 2.4× speedup. Overall, as r increases, our scheme remains more controllable than ModHE. Additionally, DPad-HE features significantly smaller key sizes compared to ModHE, along with slower key expansion with r.

Table 4: Execution time of the `HMult+Rescale` operation and relinearization key size for parameters in Table 3, on the CPU platform.

	r	2	3	4	5	6	7	8
Time (sec)	ModHE [36]	0.049	0.242	0.772	2.049	5.676	8.173	19.06
	DPad-HE	0.039	0.124	0.325	0.737	1.328	2.066	3.337
	Speedup	1.26×	1.95×	2.37×	2.78×	4.27×	3.95×	5.71×
Size (MB)	ModHE [36]	35	210	859	2,559	5,874	11,970	23,388
	DPad-HE	24	106	344	793	1,696	2,758	5,234

RQ2: How does the proposed scheme compare in performance and key size to traditional RLWE schemes?

This experiment aims to demonstrate the performance comparison between our scheme and existing schemes given the same multiplicative depth. To facilitate comparison with RLWE-based schemes at the same multiplicative depth, we only select r as a power of 2, while keeping the same $N = r \cdot n$ to maintain the same level of security with the RLWE-based schemes.

We conducted experiments under two settings, $N = 2^{15}$ and 2^{16}. And the MLWE-based scheme is evaluated with the representative parameters of small ranks $r = 2$ and $r = 4$. The detailed parameter sets are provided in Table 5.

To ensure a fair comparison, all schemes are bound to similar multiplication precision, around 33.9 for $N = 2^{15}$ and 33.4 for $N = 2^{16}$, measured as the average of the negative log of the ℓ_2-norm of the difference between the expected and the decrypted messages.

Table 5: Parameters with the same multiplicative depth for RQ2-6.

Set	N	n	r	$\hat{r}$	$\log_2 q$	$\log_2 P$	$\log_2 \hat{P}$	λ	$\hat{\lambda}$
I	2^{15}	2^{15}	1	-	$60 + 40 \times 19$	60	-	126.2	-
II		2^{14}	2	3			$60 + 55 \times 6$		138.9
III		2^{13}	4	5			$60 + 55 \times 4$		126.5
IV	2^{16}	2^{16}	1	-	$60 + 40 \times 41$		-	127.1	-
V		2^{15}	2	3			$60 + 55 \times 13$		136.6
VI		2^{14}	4	5			$60 + 55 \times 9$		133

For the $N = 2^{15}$ setting, our algorithm achieved speedups of 1.35× and 2.37× compared to ModHE, respectively, although we still experienced approximately a 10% performance loss compared to RNS-CKKS. For the $N = 2^{16}$ setting, our algorithm achieved speedups of 1.87× and 3.17× compared to ModHE respectively. Furthermore, compared to RNS-CKKS, we observed approximately a 20% performance improvement in latency with this setting.

In terms of the relinearization key size, as is demonstrated in Table 7, our scheme still witnesses larger keys despite the significant improvement upon ModHE. However, this issue is relatively minor compared to the ciphertext size because we typically store only one copy of the relinearization key. Note that the MLWE-based ciphertext size is only $\frac{1+r^{-1}}{2}$ of the RLWE-based ciphertext size with fixed N, while ciphertext expansion is a more critical barrier in practical FHE applications. Therefore, the enlarged key size is somewhat compensated by the reduced size of the ciphertext.

Table 6: Execution time (sec) of the `HMult+Rescale` operation for parameters in Table 5, on the CPU platform. The ratio indicates the performance ratio of DPad-HE vs. RNS-CKKS

N	$r = 1, n = N$	$r = 2, n = N/2$			$r = 4, n = N/4$		
	RNS-CKKS [11, 27]	ModHE [36]	DPad-HE		ModHE [36]	DPad-HE	
2^{15}	0.301	0.429	0.317	0.95×	0.772	0.325	0.93×
2^{16}	2.482	3.718	1.99	1.25×	6.645	2.03	1.22×

Table 7: Estimated size (in MB) of the relinearization keys for parameters in Table 5. The ratio indicates the key size ratio of DPad-HE vs. RNS-CKKS.

N	$r = 1, n = N$	$r = 2, n = N/2$			$r = 4, n = N/4$		
	RNS-CKKS [11, 27]	ModHE [36]	DPad-HE		ModHE [36]	DPad-HE	
2^{15}	137.5	309.4	166.9	121%	859.4	343.8	250%
2^{16}	1155.0	2598.75	1214.3	105%	7218.7	2010.9	174%

RQ3: How hardware-friendly does the DPad-HE compared with existing schemes?

Evaluating hardware-friendliness is indeed challenging. Here, we roughly compare the performance improvement of GPU over CPU. To achieve this, we implemented ModHE and DPad-HE on GPU. For comparison with RLWE schemes, we conducted experiments using the source code from 100x [29]. We conducted the same experiments with the parameters listed in Table 5, on the GPU platform. The results of the execution time comparison are shown in Table 8.

Table 8: Execution time (microsec) of the `HMult+Rescale` operation for parameters in Table 5, on the GPU platform. Without parentheses, the ratio indicates the performance ratio of DPad-HE vs. 100x. With parentheses, the ratio refers to the performance improvement of the GPU implementation compared to the CPU implementation (in Table 6).

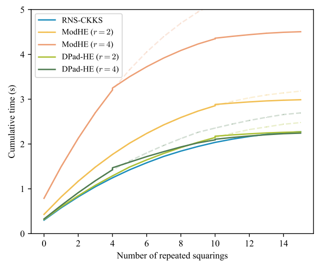
N	$r = 1, n = N$	$r = 2, n = N/2$			$r = 4, n = N/4$		
	100x [29]	ModHE [36]	DPad-HE		ModHE [36]	DPad-HE	
2^{15}	1.186 (254×)	1.869 (230×)	1.035 (306×)	1.15×	3.661 (210×)	1.455 (223×)	0.82×
2^{16}	12.37 (201×)	18.84 (197×)	8.778 (227×)	1.41×	31.36 (212×)	8.686 (234×)	1.42×

Compared to the GPU version of ModHE, DPad-HE achieves speedups ranging from 1.81× to 3.61× varying different parameter sets, with a modest module rank. For the $N = 2^{16}$ setting, we also reach a speedup of 1.41× over 100x [29]. In terms of the acceleration ratio between GPU and single-threaded CPU implementations, the GPU implementation beats our own CPU implementation by a maximum of 306×, while in 100x the corresponding value is 254×. This indicates that MLWE-based schemes are more suitable for parallel platforms.

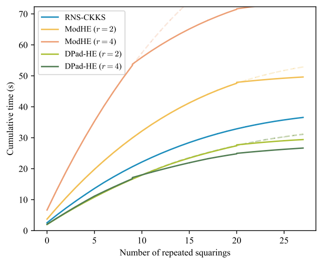
RQ4: In an entire homomorphic evaluation process, how much performance advantage does the rank reduction feature bring?

In this experiment, we will go through all the multiplicative levels on the CPU platform. For MLWE-based schemes, we will carry out the evaluation with and without rank reduction at the intermediate levels. Our evaluation employs a function of repeated squaring and uses the same parameters as specified in Table 5 to assess the execution time of `HMult+Rescale`. We reduce one module rank whenever the current modulus is small enough to maintain the security of the rank reduction key[4].

Fig. 3 illustrates the total time consumed by the function of repeated squaring, highlighting the performance characteristics of the MLWE-based scheme when rank reduction is used to lower the rank of ciphertext in the intermediate layers. It can be observed that each rank reduction operation effectively shortens the evaluation time for subsequent layers. Overall, rank reduction brings more benefits than drawbacks to the performance of MLWE-based schemes.



(a) Set I, II, III ($N = 2^{15}$, $L = 19$)



(b) Set IV, V, VI ($N = 2^{16}$, $L = 41$)

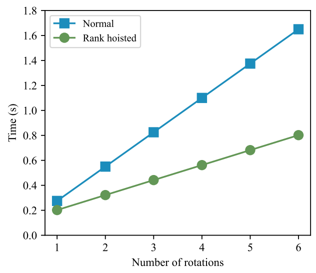
Figure 3: Accumulative time of evaluating repeated squarings on the CPU platform. The dashed lines represent the performance curve of the MLWE-based scheme without rank reduction, while minor folds in the solid lines represent the additional time consumption introduced by the rank reduction operation itself.

When utilizing the rank reduction operation, DPad-HE takes 60% and 36% of the total time compared to ModHE for $N = 2^{16}$, with $r = 2$ and $r = 4$, respectively. Especially, DPad-HE's time consumption with $r = 4$ is 71% of the RNS-CKKS scheme. For $N = 2^{15}$, DPad-HE consumes 76% and 49% of the total time compared to ModHE, with total time consumption at this parameter setting being close to that of RNS-CKKS.

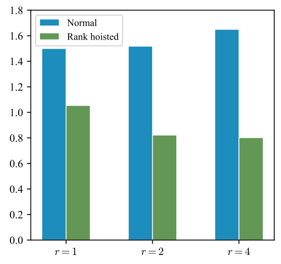
[4]For $r \geq 3$, many successive rank reduction for $r \longrightarrow (r-1)$ might lead to a possible cryptanalysis [36], while $r \longrightarrow r/2$ not. Since r here is at most 4, the performance gap between the two strategies is negligible. Thus we omit the experiment for the latter.

RQ5: How does the rank-hoisted rotation perform compared with the normal method?

With regard to our proposed rank-hoisting rotation algorithm, we demonstrate that the advantage brought by rank-hoisting grows with the increase of rotation numbers. It's worth noting that in the RLWE-based scheme ($r = 1$), we also can lift its module rank to a temporary 2 to accelerate consecutive rotations. This result reveals that the MLWE-based scheme holds an edge in handling consecutive rotations, allowing for reducing the overall key size with less performance loss.



(a) Varying numbers of rotations given $N = 2^{15}$ and $r = 4$.



(b) Varying module rank for 6 rotations given $N = 2^{15}$.

Figure 4: Performance of rank-hoisted rotation on different set of parameters.

RQ6: How lightweight is the proposed method for conversion between MLWE and RLWE schemes?

In this experiment, we conduct a micro-benchmark to assess the performance of our MLWE-RLWE conversion method. We reuse the parameters in Table 5, to perform the single-ciphertext transformation between $\text{RLWE}_{Q,N}$ and $\text{MLWE}^{r}_{Q,n}$. The result is shown in Table 9. Our method consists of an `Embed` mapping and $\log r$ of RLWE rotations. With our optimizations, the time required for `Embed` is negligible, making the conversion the same order of magnitude with RLWE key-switching. This efficiency allows us to incorporate such switching into any evaluation, thereby enabling more possibilities.

6 Discussion

From the above evaluation, we can see that we take a giant step towards an efficient MLWE-based FHE, making its performance (more precisely, execution latency) comparable to RLWE-based ones. Particularly, DPad-HE's inherent advantages such as greater flexibility, hardware reusability, parallelism, and better security assumptions endow it with broader application potential. Of course, we still leave something unsolved. This section discusses the limitations and scalability of DPad-HE.

Table 9: Execution time (sec) of the MLWE-RLWE conversion operations on the CPU platform. "M" and "R" are short for MLWE and RLWE ciphertexts. The conversion is performed across slot-encoding ciphertexts.

N	$r = 2, n = N/2$		$r = 4, n = N/4$		RLWE
	M→R	R→M	M→R	R→M	Key-switching
2^{15}	0.253	0.015	0.517	0.014	0.245
2^{16}	2.285	0.062	4.439	0.051	2.234

6.1 Decreased Packing Message

As is mentioned in [36], regardless of the rank, the amount of messages that can be packed is at most $n/2$ for MLWE-based schemes, while for RLWE schemes with the same configuration, it is $N/2 = r \cdot n/2$. This means that although DPad-HE can gain some advantages in terms of execution latency, it still has certain disadvantages from a throughput perspective. However, the main challenge for current FHE-based applications is high latency and excessive memory occupation, while SIMD capability can be slightly sacrificed, especially when evaluating very deep circuits with a tremendous dimension. Real-world examples like [33] also reveal that the large packing sizes often exceed practical needs, since the actual number of messages to be evaluated simultaneously is binding with the specific task, instead of the circuit depth as the HE-cryptosystem is.

6.2 Support for Boostrapping

Like in the RNS-CKKS scheme, a straightforward method of approximate bootstrapping in DPad-HE can be performed via: `ModRaise`, `CoeffsToSlots`, `EvalMod` and `SlotsToCoeffs`, of which the building blocks are basic homomorphic operations[5]. Yet another potential solution is to combine the conversion algorithm proposed in Section 4: converting the ciphertext to the RLWE format, performing bootstrapping using existing RLWE-based methods, and then converting it back to an MLWE ciphertext. We have listed a more MLWE-friendly bootstrapping method as a future work.

6.3 Incorporating with Key Decomposition

Key decomposition is an emerging technique that accelerates the external product operation introduced by the gadget decomposition. It was first proposed by Kim et al. [31] for the RNS-based decomposition, and recently developed by Belorgey et al. [5] to adapt it to the natural 2^K-base integer representation, offering better generalization and performance. As an algorithmic optimization for RNS-based gadget decomposition, Kim et al.'s algorithm can be migrated to our proposed generalized key-switching method in Section 3.1.1, thereby further enhancing the performance of DPad-HE. However, it should be noted that such a technique will exacerbate the evaluation key size. We recommend applying this technique when the key size is not a critical concern.

[5]For ciphertexts that have undergone a RankReduce procedure, a pre-processing step to restore module rank by appending zero polynomials may be essential.

7 Conclusion

In this work, we have introduced a novel MLWE-based FHE scheme called DPad-HE, with better performance, less memory consumption, and better noise control. Inspired by our insightful finding for the reason of the unsatisfactory performance of prior schemes, DPad-HE introduces a novel design that manipulates homomorphic evaluation in the module rank dimension.

We have implemented most of our contributions on the CPU and GPU platforms. We observe a maximum speedup of 5.7× in MLWE-based homomorphic multiplication over the SOTA work [36]. Moreover, to the best of our knowledge, this is the first reported instance that MLWE-based schemes are on par with traditional RLWE ones, achieving up to 1.25× and 1.41× speedup on CPU and GPU, respectively, in certain scenarios. Additionally, we also provide a lightweight conversion method between RLWE and MLWE, which opens up new possibilities for both RLWE-based and MLWE-based FHEs.

An immense unexplored field of MLWE-based HE cryptosystems still exists. In the future, we will focus on efficient bootstrapping and GPU-based acceleration (e.g., using tensor cores in H100 [38]) for the MLWE-based FHE.

Acknowledgements

This work is supported in part by the National Natural Science Foundation of China No. 61902392, CCF-AFSG Research Fund under Award No. CCF-AFSG RF20230206. Fangyu Zheng is the corresponding author (*E-mail: zhengfangyu@ucas.ac.cn*).

References

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

[2] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, et al. 2021. Homomorphic encryption standard. *Protecting privacy through homomorphic encryption* (2021), 31–62.

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

[4] Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, Jai Hyun Park, and Damien Stehlé. 2023. HERMES: Efficient Ring Packing Using MLWE Ciphertexts and Application to Transciphering. In *Annual International Cryptology Conference*. Springer, 37–69.

[5] Mariya Georgieva Belorgey, Sergiu Carpov, Nicolas Gama, Sandra Guasch, and Dimitar Jetchev. 2023. Revisiting Key Decomposition Techniques for FHE: Simpler, Faster and More Generic. Cryptology ePrint Archive, Paper 2023/771. https://eprint.iacr.org/2023/771 https://eprint.iacr.org/2023/771.

[6] Song Bian, Zhou Zhang, Haowen Pan, Ran Mao, Zian Zhao, Yier Jin, and Zhenyu Guan. 2023. HE3DB: An Efficient and Elastic Encrypted Database Via Arithmetic-And-Logic Fully Homomorphic Encryption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2930–2944.

[7] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 353–367.

[8] Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2021. Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys. In *Advances in Cryptology – EUROCRYPT 2021*, Anne Canteaut and François-Xavier Standaert (Eds.). Springer International Publishing, Cham, 587–617.

[9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 1–36.

[10] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. Bootstrapping for Approximate Homomorphic Encryption. In *Advances in Cryptology – EUROCRYPT 2018*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer International Publishing, Cham, 360–384.
[11] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. A full RNS variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*. Springer, 347–368.
[12] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 409–437.
[13] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer, 409–437.
[14] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology* 33, 1 (2020), 34–91.
[15] Ana Costache and Nigel P Smart. 2016. Which ring based somewhat homomorphic encryption scheme is best?. In *Topics in Cryptology-CT-RSA 2016: The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29-March 4, 2016, Proceedings*. Springer, 325–340.
[16] Cryptolab. 2024. HEaaN Private AI Homomorphic Encryption Library. https://heaan.it.
[17] Wei Dai and Berk Sunar. 2015. cuHE: A homomorphic encryption accelerator library. In *International Conference on Cryptography and Information Security in the Balkans*. Springer, 169–186.
[18] Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), 238–268.
[19] Léo Ducas and Daniele Micciancio. 2015. FHEW: bootstrapping homomorphic encryption in less than a second. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 617–640.
[20] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.* 2012 (2012), 144.
[21] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. 2023. Tensorfhe: Achieving practical computation on encrypted data using gpgpu. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 922–934.
[22] Lili Gao, Fangyu Zheng, Niall Emmart, Jiankuo Dong, Jingqiang Lin, and Charles Weems. 2020. DPF-ECC: Accelerating Elliptic Curve Cryptography with Floating-Point Computing Power of GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 494–504.
[23] Lili Gao, Fangyu Zheng, Rong Wei, Jiankuo Dong, Niall Emmart, Yuan Ma, Jingqiang Lin, and Charles Weems. 2021. DPF-ECC: A Framework for Efficient ECC With Double Precision Floating-Point Computing Power. *IEEE Transactions on Information Forensics and Security* 16 (2021), 3988–4002.
[24] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 169–178.
[25] Craig Gentry, Shai Halevi, Chris Peikert, and Nigel P Smart. 2013. Field switching in BGV-style homomorphic encryption. *Journal of Computer Security* 21, 5 (2013), 663–684.
[26] Shai Halevi and Victor Shoup. 2018. Faster homomorphic linear transformations in HElib. In *Annual International Cryptology Conference*. Springer, 93–120.
[27] Kyoohyung Han and Dohyeong Ki. 2020. Better bootstrapping for approximate homomorphic encryption. In *Cryptographers' Track at the RSA Conference*. Springer, 364–390.
[28] Lei Jiang, Qian Lou, and Nrushad Joshi. 2022. Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 235–240.
[29] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 2021. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 114–148.
[30] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. 2022. Approximate homomorphic encryption with reduced approximation error. In *Cryptographers' Track at the RSA Conference*. Springer, 120–144.
[31] Miran Kim, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. 2023. Accelerating HE Operations from Key Decomposition Technique. In *Advances in Cryptology – CRYPTO 2023*, Helena Handschuh and Anna Lysyanskaya (Eds.). Springer Nature Switzerland, Cham, 70–92.
[32] Adeline Langlois and Damien Stehlé. 2015. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography* 75, 3 (2015), 565–599.
[33] Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. 2022. Low-Complexity Deep Convolutional Neural Networks on Fully Homomorphic Encryption Using Multiplexed Parallel Convolutions. In *Proceedings of the 39th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 162)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (Eds.). PMLR, 12403–12422. https://proceedings.mlr.press/v162/lee22e.html
[34] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1–23.
[35] Souhail Meftah, Benjamin Hong Meng Tan, Chan Fook Mun, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Vijay Chandrasekhar. 2021. Doren: toward efficient deep convolutional neural networks with fully homomorphic encryption. *IEEE Transactions on Information Forensics and Security* 16 (2021), 3740–3752.
[36] Anisha Mukherjee, Aikata Aikata, Ahmet Can Mert, Yongwoo Lee, Sunmin Kwon, Maxim Deryabin, and Sujoy Sinha Roy. 2024. ModHE: Modular Homomorphic Encryption Using Module Lattices: Potentials and Limitations. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024, 1 (2024), 527–562.
[37] NVIDIA. 2024. CUDA C++ Programming Guide v12.4. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#.
[38] NVIDIA. 2024. NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-tensor-core.
[39] Oded Regev. 2005. On lattices, learning with errors, random linear codes, and cryptography. 84–93. https://doi.org/10.1145/1060590.1060603
[40] George W Reitwiesner. 1960. Binary arithmetic. In *Advances in computers*. Vol. 1. Elsevier, 231–308.
[41] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. 1978. On data banks and privacy homomorphisms. *Foundations of secure computation* 4, 11 (1978), 169–180.
[42] SEAL 2022. Microsoft SEAL (release 4.0). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA..
[43] Tian Zhou, Fangyu Zheng, Guang Fan, Lipeng Wan, Wenxu Tang, Yixuan Song, Yi Bian, and Jingqiang Lin. 2024. ConvKyber: Unleashing the Power of AI Accelerators for Faster Kyber with Novel Iteration-based Approaches. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024, 2 (2024), 25–63.

A Proofs of Lemmas and Theorems

A.1 Noise Estimations

The canonical embedding $\mathsf{can} : \mathbb{R}[X]/(X^N + 1) \to \mathbb{C}^N$ is defined as $\mathsf{can}(a(X)) = (a(\zeta^j))_{j \in \mathbb{Z}^*_{N/2}}$ for $\zeta = \exp(\pi i/N)$. It's ℓ_∞-norm is called the *canonical embedding norm* and is denoted as $\|a(X)\|^{\mathsf{can}}_\infty = \|\mathsf{can}(a(X))\|_\infty$.

We follow the heuristic approach in [13, 15, 30]. Assume that the coefficients of a sampled polynomial $a(X)$ are independent and identically distributed, where σ^2 is the variance of each such distribution. Since the vector $(1, \zeta, \cdots, \zeta^{N/2})$ has an Euclidean norm of $\sqrt{N}$, the random variable $a(\zeta)$ has variance $\sigma^2 N$. Then, a polynomial sampled from a uniform distribution over $\mathcal{R}_{q,N}$ has a variance of $q^2N/12$. We use the bound $\|a(X)\|^{\mathsf{can}}_\infty$ for the canonical embedding norm of $a(X)$. For a multiplication of two independent polynomials $a(X), b(X)$ of which the coefficients are sampled from Gaussian distributions with variables σ_1^2 and σ_2^2, we use a high-probability bound $\|a(X)b(X)\|^{\mathsf{can}}_\infty \le 16\sigma_1\sigma_2 N^2$.

A.1.1 Proof of Lemma 3.1.

Proof. **(Lemma 3.1)** The output ciphertext ct_{up} of the second step "inner product" satisfies that

$$\begin{aligned}\mathsf{ct}_{up} \cdot (1, s_2) &= (1, s_2^\top) \cdot \sum_{0 \le i < d} \mathsf{swk}_i \cdot \boldsymbol{c}'_i \\ &= \sum_{0 \le i < d} (1, s_2^\top) \cdot (\boldsymbol{b}_{\mathsf{swk}_i}, \boldsymbol{A}_{\mathsf{swk}_i})^\top \cdot \boldsymbol{c}'_i \\ &= \sum_{0 \le i < d} (\boldsymbol{b}_{\mathsf{swk}_i} + \boldsymbol{A}_{\mathsf{swk}_i} s_2)^\top \cdot \boldsymbol{c}'_i\end{aligned}$$

$$
\begin{aligned}
&= \sum_{0\le i<d} (Pg_i s_1 + \boldsymbol{e}_{\mathsf{swk}_i})^\top \cdot \boldsymbol{c}'_i \\
&= P\left\langle \sum_{0\le i<d} g_i \boldsymbol{c}'_i, \boldsymbol{s}_1 \right\rangle + \sum_{0\le i<d} \langle \boldsymbol{c}'_i, \boldsymbol{e}_{\mathsf{swk}_i}\rangle \\
&= P\langle \boldsymbol{c}, \boldsymbol{s}_1\rangle + e'
\end{aligned}
$$

where

$$e' = \sum_{0\le i<d} \langle \boldsymbol{c}'_i, \boldsymbol{e}_{\mathsf{swk}_i}\rangle = \sum_{0\le j<r_1}\sum_{0\le i<d} \boldsymbol{c}'_{i,j}(X)\cdot \boldsymbol{e}_{\mathsf{swk}_{i,j}}(X).$$

Therefore, e' is bounded by

$$\|e'\|_\infty^{\mathsf{can}} \le r_1 \cdot \frac{16\cdot d\cdot \sigma n}{\sqrt{12}} \max_{0\le i<d} Q'_i.$$

The last step "modulus down" outputs a ciphertext $\mathsf{ct} = \lfloor P^{-1}\cdot \mathsf{ct}_{up}\rceil$, therefore the overall noise consists of two parts: the noise $P^{-1}e'$ which comes from ct_{up} and the rounding noise $\epsilon_0 + \langle \boldsymbol{\epsilon}_1, \boldsymbol{s}_2\rangle$ where polynomials in $(\epsilon_0, \boldsymbol{\epsilon}_1)$ have coefficients smaller than 1/2. The bound of e_{KS} is computed by the following inequality

$$
\begin{aligned}
\|e_{KS}\|_\infty^{\mathsf{can}} &\le P^{-1}\|e'\|_\infty^{\mathsf{can}} + \|\epsilon_0\|_\infty^{\mathsf{can}} + \|\langle \boldsymbol{\epsilon}_1, \boldsymbol{s}_2\rangle\|_\infty^{\mathsf{can}} \\
&\le \frac{16\cdot d\cdot \sigma r_1 n}{\sqrt{12}P} \max_{0\le i<d} Q'_i + 6\sqrt{\frac{n}{12}} + r_2\cdot 16\cdot \sqrt{\frac{1}{12}}\sqrt{\frac{2}{3}n} \\
&= \frac{8\cdot d\cdot \sigma r_1 n}{\sqrt{3}P} \max_{0\le i<d} Q'_i + \sqrt{3n} + \frac{8\sqrt{2}r_2 n}{3}
\end{aligned}
$$

□

A.1.2 *Proof of Lemma 3.2.*

Proof. **(Lemma 3.2)** The output ciphertext ct' satisfies that

$$
\begin{aligned}
\mathsf{ct}'\cdot(1, \boldsymbol{s}) &= c_0 + \hat{c}_0 + \check{c}_0 + \langle \boldsymbol{c}_1, \boldsymbol{s}\rangle + \langle \hat{\boldsymbol{c}}_1[:r], \boldsymbol{s}\rangle + \langle \check{\boldsymbol{c}}_1, \boldsymbol{s}\rangle \\
&= c_0 + \hat{c}_0 + \langle \boldsymbol{c}_1, \boldsymbol{s}\rangle + \langle \hat{\boldsymbol{c}}_1[:r], \boldsymbol{s}\rangle + \langle \hat{\boldsymbol{c}}_1[r:], \boldsymbol{s}'\rangle + e_{\mathsf{RkDn}} \\
&= c_0 + \hat{c}_0 + \langle \boldsymbol{c}_1, \boldsymbol{s}\rangle + \langle \hat{\boldsymbol{c}}_1, (\boldsymbol{s}\|\boldsymbol{s}')\rangle + e_{\mathsf{RkDn}} \\
&= c_0 + \langle \boldsymbol{c}_1, \boldsymbol{s}\rangle + \langle \boldsymbol{c}_2, \boldsymbol{s}\otimes \boldsymbol{s}\rangle + e_{\mathsf{cRelin}} + e_{\mathsf{RkDn}}
\end{aligned}
$$

Therefore, the bound of $e_{\mathsf{DPad.Relin}}$ is computed by the following inequality

$$\|e_{\mathsf{DPad.Relin}}\|_\infty^{\mathsf{can}} \le \|e_{\mathsf{cRelin}}\|_\infty^{\mathsf{can}} + \|e_{\mathsf{RkDn}}\|_\infty^{\mathsf{can}}$$

By applying Lemma 3.1, this leads to the claimed bound. □

A.2 Embedding and Extracting in the NTT form

A.2.1 *Proof of Theorem 4.1.*

Proof. According to the definition of the negacyclic NTT, it holds that

$$\tilde{a}_{j,k} = \sum_{i=0}^{n-1} a_{j,i}\psi_{2n}^{i(2k+1)} \tag{5}$$

$$\tilde{a}'_K = \sum_{i=0}^{N-1} a'_i \psi_{2N}^{i(2K+1)} \tag{6}$$

Now we decompose the loop variable i of Equation (6) into $ur+v$ for $0\le u<n$ and $0\le v<r$. Note that $a'_{ur+v} = a_{v,u}$ and $\psi_{2N}^r = \psi_{2n}$, we have

$$
\begin{aligned}
\tilde{a}'_K &= \sum_{v=0}^{r-1}\sum_{u=0}^{n-1} a'_{ur+v}\psi_{2N}^{(ur+v)(2K+1)} \\
&= \sum_{v=0}^{r-1}\sum_{u=0}^{n-1} a_{v,u}\psi_{2N}^{ur(2K+1)}\psi_{2N}^{v(2K+1)} \\
&= \sum_{v=0}^{r-1}\psi_{2N}^{v(2K+1)}\sum_{u=0}^{n-1} a_{v,u}\psi_{2n}^{u(2K+1)}
\end{aligned}
$$

We decompose the entry K into $\alpha n+\beta$ for $0\le\alpha<r$ and $0\le\beta<n$, we have

$$
\begin{aligned}
\tilde{a}'_{\alpha n+\beta} &= \sum_{v=0}^{r-1}\psi_{2N}^{v(2\alpha n+2\beta+1)}\sum_{u=0}^{n-1} a_{v,u}\psi_{2n}^{u(2\alpha n+2\beta+1)} \\
&= \sum_{v=0}^{r-1}\psi_{2N}^{v(2\alpha n+2\beta+1)}\sum_{u=0}^{n-1} a_{v,u}\psi_{2n}^{u(2\beta+1)}
\end{aligned}
$$

Note that the inner loop vanished by applying Equation (5), therefore

$$
\begin{aligned}
\tilde{a}'_{\alpha n+\beta} &= \sum_{v=0}^{r-1}\tilde{a}_{v,\beta}\psi_{2N}^{v(2\alpha n+2\beta+1)} \\
&= \sum_{v=0}^{r-1}\left(\tilde{a}_{v,\beta}\psi_{2N}^{v(2\beta-n+1)}\right)\psi_{2r}^{v(2\alpha+1)}
\end{aligned}
$$

□

A.2.2 *Proof of Theorem 4.2.*

Proof. Note that in Equation (2), if we view the expression within the parenthesis as a whole, the equation is performing the negacyclic NTT operation. According to the definition of the inverse negacyclic NTT, we have

$$\tilde{a}_{v,\beta}\cdot\psi_{2N}^{v(2\beta-n+1)} = r^{-1}\sum_{\alpha=0}^{r-1} a'_{\alpha n+\beta}\psi_{2r}^{-v(2\alpha+1)}$$

for $0\le v<r$ and $0\le\beta<n$. If we replace the symbol v by j and replace the symbol β by k, there is

$$\tilde{a}_{j,k} = \psi_{2N}^{-j(2k-n+1)}\cdot r^{-1}\sum_{v=0}^{r-1}\tilde{a}'_{vn+k}\psi_{2r}^{-j(2v+1)}.$$

□