Invited Paper: Efficient Design of FHEW/TFHE Bootstrapping Implementation with Scalable Parameters

Ming-Chien Ho[1,2], Yu-Te Ku[1,2,5], Yu Xiao[1,2], Feng-Hao Liu[3], Chih-Fan Hsu[1], Ming-Ching Chang[1,4], Shih-Hao Hung[2,6], Wei-Chao Chen[1]

[1] Inventec Corporation, Taipei City, 111059, Taiwan; [2] National Taiwan University, Taipei City, 10617, Taiwan; [3] Washington State University, Pullman, WA, 99164, USA; [4] State University of New York, University at Albany, Albany, NY 12222, USA; [5] Academia Sinica, Taipei City, 115201, Taiwan; [6] Mohamed bin Zayed University of Artificial Intelligence, Abu Dhabi

{r11944009,d08946006,r11922138}@ntu.edu.tw;hungsh@csie.ntu.edu.tw;feng-hao.liu@wsu.edu;{hsu.chih-fan,chen.wei-chao}@inventec.com;mchang2@albany.edu

ABSTRACT

Fully Homomorphic Encryption (FHE) is vital for computing over encrypted data, thereby enabling numerous privacy-preserving applications. This work focuses on the third generation FHE schemes (e.g., FHEW and TFHE), known for their fast bootstrapping, small FHE parameters, and robust security built on milder assumptions.

Our goal is to improve the efficiency of implementation for scalable parameters. Notably, scaling up FHE parameters moderately, such as the plaintext space, extends the applicability of third-generation FHEs to a broader range of practical scenarios. However, prevailing state-of-the-art libraries such as OpenFHE and TFHE either lack support for extensive FHE parameters beyond 64-bit integers or suffer significant performance slowdowns on widely used 64-bit architectures. To tackle this challenge, we propose a novel FFT-based multiplication implementation, which decomposes large numbers (e.g., 128-bit integers) into multiple doubles (64-bit floating points). To optimize the performance, we refine the error analysis in FFT-based FHEW/TFHE computation with the decomposition for the optimal balance between efficiency and decryption failure probability.

We evaluate our approach through comprehensive experiments, showing a 5-7x speedup using 64-bit architecture over 128-bit in core operations compared to existing libraries. Our implementation is particularly conducive to parallelization, making it well-suited for hardware acceleration, such as GPUs.

CCS CONCEPTS

• **Security and privacy → Cryptography**; • **Mathematics of computing → Numerical analysis**; • **Computing methodologies → Parallel computing methodologies**.

ICCAD '24, October 27–31, 2024, New York, NY, USA

ACM ISBN 979-8-4007-1077-3/24/10
https://doi.org/10.1145/3676536.3698873

KEYWORDS

Fully Homomorphic Encryption (FHE), FHEW, TFHE, Functional Bootstrapping, Neural Network, DNN, Decision Tree, Large Modulus Computation, 64-bit Architecture, FFT, Cryptographic Performance, Error Estimation, OpenFHE, Parallel Computing.

ACM Reference Format:
Ming-Chien Ho[1,2], Yu-Te Ku[1,2,5], Yu Xiao[1,2], Feng-Hao Liu[3], Chih-Fan Hsu[1], Ming-Ching Chang[1,4], Shih-Hao Hung[2,6], Wei-Chao Chen[1]. 2024. Invited Paper: Efficient Design of FHEW/TFHE Bootstrapping Implementation with Scalable Parameters. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '24), October 27–31, 2024, New York, NY, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3676536.3698873

1 INTRODUCTION

Fully Homomorphic Encryption (FHE) allows arbitrary computation on encrypted data without decryption, enabling numerous applications in secure cloud computing [23] and privacy-preserving data analytics. The third-generation FHE schemes, represented by FHEW [8] and TFHE [7], are robust against both classical and quantum attacks due to their plausible security foundation, namely *learning with errors* (LWE) assumptions that operate on a lower modulus-to-noise ratio[1]. However, applying FHEW and TFHE to high-precision or large-scale numerical data analysis, such as in statistics and machine learning, often requires a larger message space. This need for an increased modulus presents challenges for efficient implementation. Current state-of-the-art libraries face significant limitations. For instance, the fastest implementations, like TFHE-rs, limit the modulus to 2^{64}, while versatile libraries like OpenFHE encounter significant efficiency declines. For example, as the modulus increases from 2^{54} to 2^{66}, bootstrapping time slows down by about 14 times. Our work aims to enhance efficiency in this domain, focusing on the FHEW/TFHE implementations.

FHEW/TFHE bootstrapping is an important operation that reduces ciphertext noise, allowing continuous operations on the encrypted data. However, it is the most time-consuming procedure. To improve efficiency, researchers have studied to optimize the critical fundamental operation – polynomial ring multiplication. To achieve this, two methods have been studied: Number Theoretic Transform (NTT) and Fast Fourier Transform (FFT). NTT does not incur numerical errors during the multiplication procedure, but the computation is relatively slow compared to FFT, which however

[1]This is a weaker and thus more plausible LWE assumption.

incurs additional errors due to the precision limitation in floating point numbers. For larger modulus settings, the OpenFHE library (https://www.openfhe.org/) supports ciphertext modulus exceeding 2^{64} using NTT with 128-bit operations, but this significantly hampers speed on 64-bit computer architectures. This raises the question of whether a more efficient and 64-bit architecture-friendly method exists for bootstrapping with a large modulus. Doubling the precision for the FFT floating point computation could achieve efficient 128-bit polynomial ring multiplication. However, naively adopting FFT to the large-modulus bootstrapping introduces large numerical errors. The error inevitably impacts the bootstrapping result. For example, using FFT with double precision for modulus 2^{62} and message modulus 16 increases the bootstrapping failure probability to 0.93, given a gadget base of 2^{31}. Therefore, the modulus of the TFHE-rs (https://docs.zama.ai/tfhe-rs) can only support ciphertext modulus to 2^{64}.

In this paper, we introduce a novel implementation of FFT-based large-modulus bootstrapping for FHEW/TFHE, incorporating a specially designed large-number decomposition. Our implementation is suitable for running on 64-bit architectures and offers faster execution. Additionally, we examine the trade-offs between computational efficiency and bootstrapping error and refine an existing error function [26]. By the refined error function, users can adjust the bootstrapping parameters to balance the trade-off between efficiency and error based on the specific requirements of their target problem.

1.1 Our Contributions

We summarize our contributions as follows:

- *Efficient Polynomial Ring Multiplication:* We propose a method tailored for FHEW/TFHE that efficiently handles large modulus by decomposing numbers into the minimum doubles necessary to suppress error growth.
- *Refined Bootstrapping Error Function:* We refine the error function for large modulus to accommodate numerical errors from FFT with double precision. This improves the bootstrapping error estimation, and balances efficiency and decryption failure rates, thereby assisting in the optimal bootstrapping parameters selection.
- *Method Realization in OpenFHE v1.1.0:* Our method is built upon OpenFHE v1.1.0, which provides flexible parameter selection and broad application scenarios. Compared to ordinary OpenFHE v1.1.0, our bootstrapping implementation is 7-8x faster in single-threaded mode and 5-7x faster in multi-threaded mode with modulus ranging from 2^{62} to 2^{76}; compared to TFHE-rs v0.6.1, our method supports a larger ciphertext modulus and effectively manages numerical errors. We have evaluated our bootstrapping method on OpenFHE v1.1.0. It can also run on the latest OpenFHE v2.1.0 but the performance boost should be carefully investigated.
- *Better Efficiency and Performance:* Our method outperforms existing approaches, including an implementation using a single long double for computations and another implementation using a different large number decomposition method (for ciphertext modulus below 2^{76}), as detailed in § 5.1, demonstrating superior efficiency in real-world scenarios.

Our implementation and the refined error function improve the efficiency and practicalness of FHEW/TFHE bootstrapping for large modulus support, which could widely be adapted to different cryptographic applications.

1.2 Related Work

Fully Homomorphic Encryption (FHE) has evolved through several generations since Gentry's pioneering work [14]. First-generation schemes introduced bootstrapping based on ideal lattices. Second-generation schemes including BGV [4] and BFV [9] focused on improving computational efficiency with leveled encryption. Third-generation schemes including FHEW [8] and TFHE [7] used fast bootstrapping techniques for effective evaluation of non-linear functions. Fourth-generation schemes like CKKS [5] support encrypted operations on real numbers but provide only approximate results upon decryption. More details on each generation of FHE schemes are in [25].

Second-generation and fourth-generation FHE schemes prioritize computational efficiency by using a higher modulus-to-noise ratio. However, they depend on aggressive security assumptions, making them vulnerable to attacks from traditional methods and quantum computers. In contrast, third-generation schemes use smaller FHE parameters, leading to lightweight ciphertexts and smaller secret keys. They also rely on lattice problems with fewer security assumptions, offering stronger confidence in security applications.

Bootstrapping for extended message spaces: Lu *et al.* [15] introduced the PEGASUS framework to transition between CKKS and FHEW. Since CKKS has a larger message space than FHEW, PEGASUS uses *large-domain bootstrapping* to scale down CKKS ciphertexts to fit FHEW's smaller message space. This extends bootstrapping to support a larger message domain but decreases precision. In contrast, Liu *et al.* [22] introduced a digit decomposition method that divides a large message into smaller chunks and performs individual bootstrapping on each chunk. This approach preserves the input message's precision but requires multiple rounds of bootstrapping.

Given that our approach efficiently supports bootstrapping under a large modulus, we can extend both of these prior works to accommodate an even larger message modulus.

2 PRELIMINARIES

In this section, we overview the basic schemes of FHEW/TFHE bootstrapping [6–8, 26]. § 2.1 and 2.2 introduce the encryption schemes used in bootstrapping. § 2.3 discusses specific operations integral to bootstrapping, based on the work of Micciancio and Polyakov [26]. Finally, § 2.4 presents methods for accelerating polynomial ring multiplications.

Throughout this paper, logarithms are expressed in base two unless stated otherwise. The integer field modulo q is denoted as $\mathbb{Z}_q$. We define the polynomial ring $R = \mathbb{Z}[X]/(X^N + 1)$, where N is a power of two, making it a cyclotomic ring. $R_Q = \mathbb{Z}_Q[X]/(X^N+1)$ is a polynomial ring with coefficients modulo Q and dimension N; we refer to the ring as R_Q throughout the paper. A vector is represented with an arrow above the letter, *e.g.*, $\vec{a}$, with $a[i]$ indicating the i-th element of $\vec{a}$. For a ring $\mathbf{d}(X) \in R_Q$, its coefficient vector is $\vec{\mathbf{d}}$, and it

can be expressed as $\mathbf{d}(X) = \mathbf{d}[0]X^0 + \mathbf{d}[1]X^1 + \cdots + \mathbf{d}[N-1]X^{N-1}$. For simplicity, we denote a ring by lowercase bold letters, *e.g.*, $\mathbf{d}$. A ciphertext is denoted by uppercase bold letters, such as $\mathbf{C}$ or $\mathbf{ACC}$. The notation $\mathbf{d}_i^j$ and $\mathbf{C}_i^j$ refer to the j-th component after decomposing the ring $\mathbf{d}_i$ and the ciphertext $\mathbf{C}_i$, respectively.

2.1 LWE Encryption Scheme

Let q represent the ciphertext modulus and p represent the message modulus. In the LWE scheme, a message $m \in \mathbb{Z}_p$ is encrypted using a secret key $\vec{s} \in \mathbb{Z}_q^n$. The encryption is represented as $\mathsf{LWE}_{\vec{s}}(\frac{q}{p}\ m) = (\vec{a}, \langle \vec{a}, \vec{s} \rangle + \frac{q}{p}\ m + e) = (\vec{a}, b)$, where $\vec{a} \in \mathbb{Z}_q^n$, $b \in \mathbb{Z}_q$, and $e \in \mathbb{Z}_q$ is a small Gaussian noise. The decrypted message g of a ciphertext $(\vec{a}, b)$ in LWE scheme is calculated as $(\vec{a}, b)$ is $g = \lceil \frac{p}{q}\ (b - \langle \vec{a} \cdot \vec{s} \rangle) \rfloor$, where $g \in \mathbb{Z}_p$ and $\lceil \cdot \rfloor$ is rounding function. During a sequence of homomorphic operations, the error in the ciphertext accumulates. For successful decryption, the error in the ciphertext must remain below $\frac{q}{2p}$.

2.2 Ring LWE Encryption Scheme

In the Ring LWE (RLWE) scheme, a message $\mathbf{m} \in R_Q$ is encrypted with secrete key $\mathbf{s} \in R$ as: $\mathsf{RLWE}_{\mathbf{s}}(\mathbf{m}) = (\mathbf{a}, \mathbf{a} \cdot \mathbf{s} + \mathbf{e} + \mathbf{m})$, where $\mathbf{a} \leftarrow R_Q$ is chosen uniformly at random, and the error $\mathbf{e} \leftarrow \mathcal{N}_{(0,\sigma^2)}^N$ is sampled from a discrete Gaussian distribution $\mathcal{N}$ with zero mean and variance σ. To support multiplication of $\mathsf{RLWE}_{\mathbf{s}}(\mathbf{m})$ with an arbitrary constant ring with small error, RLWE' scheme is introduced and use the base B to balance efficiency and noise growth: $\mathsf{RLWE}'_{\mathbf{s}}(\mathbf{m}) = \left(\mathsf{RLWE}_{\mathbf{s}}(\mathbf{m}), \mathsf{RLWE}_{\mathbf{s}}(B\mathbf{m}), \cdots, \mathsf{RLWE}_{\mathbf{s}}(B^{k-1}\mathbf{m})\right)$, with $k = \log_B Q$. This scheme avoids direct multiplication of the error term with the constant ring. Since RLWE and RLWE' schemes do not support ciphertext multiplications inherently, RGSW scheme built upon RLWE' is introduced: $\mathsf{RGSW}_{\mathbf{s}}(\mathbf{m}) = (\mathsf{RLWE}'_{\mathbf{s}}(-\mathbf{s} \cdot \mathbf{m}), \mathsf{RLWE}'_{\mathbf{s}}(\mathbf{m}))$. RGSW enables multiplication of an RLWE ciphertext $\mathsf{RLWE}_{\mathbf{s}}(\mathbf{m}) = (\mathbf{a}, \mathbf{b})$ by using gadget decomposition $\mathcal{G}(\cdot)$:

$$\mathcal{G}\left(\mathsf{RLWE}_{\mathbf{s}}(\mathbf{m})\right) = \left(\mathbf{a}^0, \mathbf{b}^0, \mathbf{a}^1, \mathbf{b}^1, \cdots, \mathbf{a}^{d_{\mathrm{g}}-1}, \mathbf{b}^{d_{\mathrm{g}}-1}\right),$$

where $\mathbf{a} = \sum_{i=0}^{d_{\mathrm{g}}-1} \mathbf{a}^i \cdot B^i$, $\mathbf{b} = \sum_{i=0}^{d_{\mathrm{g}}-1} \mathbf{b}^i \cdot B^i$ and $d_{\mathrm{g}} = \log_B Q$. The rings $\mathbf{a}^i$ and $\mathbf{b}^i$ are bounded by B. The multiplication of an RGSW ciphertext encrypting the message $\mathbf{m}_0$ and the RLWE ciphertext encrypted the message ring $\mathbf{m}_1$ is termed to as the external product in [6]. The expansion of the external product is represented as:

$$\mathcal{G}\left(\mathsf{RLWE}_{\mathbf{s}}(\mathbf{m}_0)\right) \cdot \mathsf{RGSW}_{\mathbf{s}}(\mathbf{m}_1) = \mathsf{RLWE}_{\mathbf{s}}(\mathbf{m}_0 \cdot \mathbf{m}_1).$$

For a detailed overview of these encryption schemes, refer to [26].

2.3 FHEW/TFHE Bootstrapping

Bootstrapping performs decryption operations homomorphically. Given an encrypted secret key with the LWE secret key $\vec{s}$, bootstrapping produces a new LWE ciphertext of the same message with reduced noise. The noise after bootstrapping depends only on the noise of the encrypted secret key and not on the original LWE ciphertext. Here, the encrypted secret key is encrypted using a different scheme from LWE, such as RGSW. The bootstrapping key is: $\mathbf{BTK} = (\mathsf{RGSW}(s[1]), \mathsf{RGSW}(s[2]), \cdots, \mathsf{RGSW}(s[n]))$. Table 1 lists the FHEW parameters used in bootstrapping.

Table 1: FHEW parameters used in bootstrapping.

Symbol	Meaning
p	the modulus for the message.
n	the LWE dimension.
q	small modulus for LWE.
N	ring dimension for RLWE and RGSW.
Q	large modulus for RLWE and RGSW.
B	gadget digit decomposition base dividing integers mod Q into d_{g} digits.

Algorithm 1: Blind Rotation

input : A ciphertext $\mathbf{C} \in$ RLWE; a vector $\vec{a} = (a[1], a[2], \cdots, a[n]), a[i] \in \mathbb{Z}_q$; the bootstrapping key $\{\mathbf{BTK}_{0i}, \mathbf{BTK}_{1i}\}_{i\in[1,n]}, \mathbf{BTK}_{ji} \in$ RGSW;
output : A refreshed ciphertext $\mathbf{ACC} \in$ RLWE

$\mathbf{ACC} \leftarrow \mathbf{C}$
for $i \leftarrow 1$ **to** n **do**
  $\mathbf{ACC}_{\mathrm{d}} \leftarrow \mathcal{G}(\mathbf{ACC})$
  $\mathbf{ACC}_{\mathrm{t}} \leftarrow (X^{a[i]} - 1) \cdot (\mathbf{ACC}_{\mathrm{d}} \cdot \mathbf{BTK}_{0i}) + (X^{-a[i]} - 1) \cdot (\mathbf{ACC}_{\mathrm{d}} \cdot \mathbf{BTK}_{1i})$
  $\mathbf{ACC} \leftarrow \mathbf{ACC} + \mathbf{ACC}_{\mathrm{t}}$
return $\mathbf{ACC}$

In this work, we leverage bootstrapping $\mathsf{Boots}[f]$ for a given function f through an RLWE accumulator from [26], facilitating the following operations:

- **Initialization**: Encrypt the $\mathsf{LWE}^{n,q}$ ciphertext $(\vec{a}, b)$ into an RLWE accumulator $\mathsf{RLWE}^{N,Q}$ without adding noise.
- **Update**: Use the ring structure to homomorphically compute $z = b - \langle \vec{a}, \vec{s} \rangle$ over the exponent, resulting in a ciphertext of $\mathsf{RLWE}^{N,Q}(X^z)$. This step, known as blind rotation [6], is detailed in Algorithm 1.
- **Extraction**: Extract the updated $\mathsf{RLWE}^{N,Q}$ ciphertext back to an $\mathsf{LWE}^{N,Q}$ ciphertext.

For detailed bootstrapping procedures, please refer to [26] and [2].

2.4 Polynomial Multiplication Acceleration

Two main methods for accelerating polynomial ring multiplication are the Number Theoretic Transform (NTT) and the Fast Fourier Transform (FFT). Both transform polynomials from their coefficient representation to NTT/FFT format. The transformed polynomials then undergo elementwise multiplication. Finally, the result is transformed back to coefficient representation. NTT operates within the integer domain using modular arithmetic in finite fields, requiring a specific prime modulus. FFT operates within the complex domain and uses floating-point operations in infinite field, introducing numerical errors but generally being faster by working with half the polynomial dimensions [16]. FFT applies modular arithmetic only after the inverse FFT (iFFT) and typically uses a power-of-two modulus. These methods reduce the complexity to $O(N \log N)$. In short, NTT ensures exact results, while FFT offers faster processing with approximate results.

3 EFFICIENT POLYNOMIAL MULTIPLICATION FOR SCALABLE FHEW/TFHE CRYPTOSYSTEM

Bootstrapping primarily involves time-consuming polynomial multiplications. To improve efficiency, we use FFT for polynomial ring multiplication. However, FFT operations with floating-point numbers in infinite fields cause numerical errors and require modulo operations after the iFFT transformation. Supporting large modulus exceeding 2^{64} with 64-bit floating-point numbers (double) exacerbates these errors. A naive approach is to use 128-bit floating-point numbers (long double) to reduce errors, but this is inefficient on current computer architectures and provides limited speed improvements. Our goal is to maintain efficiency and minimize numerical errors by using 64-bit floating-point numbers.

§ 3.1 explains our integration of the FHEW bootstrapping feature and the decomposition method to mitigate numerical errors effectively. In § 3.2, we demonstrate the extension of our method to support polynomial multiplication with a larger modulus.

3.1 Optimized Method of Polynomial Multiplication with Small Numerical Error

Using the coefficient-limiting properties of gadget decomposition, we decompose a large-coefficient polynomial into a minimal number of smaller-coefficient polynomials, effectively reducing numerical errors in polynomial multiplication. Polynomial multiplication during bootstrapping involves the ring $\mathbf{c}$ bounded by B and the ring $\mathbf{d}$ with modulus Q because $\mathbf{c}$ is the output of gadget decomposition. When the base $\bar{B}$ exceeds B, we only need to decompose $\mathbf{d}$ into multiple components $\mathbf{d}^i$. By carefully selecting $\bar{B}$, we ensure that the numerical error of $\mathbf{c}\cdot\mathbf{d}$ is predominantly influenced by the error from $\mathbf{c}\cdot\mathbf{d}^0$. The following example illustrates decomposing the ring $\mathbf{d}$ into two components, $\mathbf{d}^0$ and $\mathbf{d}^1$, and how to select $\bar{B}$ to control the numerical error of $\mathbf{c}\cdot\mathbf{d}$.

The Procedure of Decomposition. In this section, we delve into the methodology of ring decomposition. Given a ring $\mathbf{d}$, we decompose each coefficient using the base $\bar{B}=2^k$ with $k<64$, so that $\mathbf{d}[i]=\mathbf{d}^0[i]+\mathbf{d}^1[i]\bar{B}$. The decomposed form of $\mathbf{d}$ is $\mathbf{d}=\mathbf{d}^0+\mathbf{d}^1\bar{B}$, where $\mathbf{d}^0=\mathbf{d}^0[0]+\mathbf{d}^0[1]X+\cdots+\mathbf{d}^0[N-1]X^{N-1}$ and $\mathbf{d}^1=\mathbf{d}^1[0]+\mathbf{d}^1[1]X+\cdots+\mathbf{d}^1[N-1]X^{N-1}$. We denote this decomposition as: $\mathcal{D}(\mathbf{d})=(\mathbf{d}^0,\mathbf{d}^1)$, such that $\mathbf{d}=\mathbf{d}^0+\mathbf{d}^1\bar{B}\in R_Q$. Here, $\mathbf{d}^0$ and $\mathbf{d}^1$ are bounded by $\bar{B}$ and $\frac{Q}{\bar{B}}$, respectively. Because we use FFT with double precision, larger coefficients resulting from polynomial multiplication lead to greater numerical errors. To mitigate this, we use signed digit decomposition as described in [1], ensuring each coefficient of $\mathbf{d}^0$ and $\mathbf{d}^1$ is a signed number, with $\left|\mathbf{d}^0[i]\right|<\bar{B}/2$ and $\left|\mathbf{d}^1[i]\right|<\frac{Q}{2\bar{B}}$. This method effectively limits coefficient growth in the resulting polynomial multiplication. The composition of the decomposed ring is given by: $C(\mathbf{d}^0,\mathbf{d}^1)=\mathbf{d}^0+\mathbf{d}^1\bar{B}=\mathbf{d}$, where $\mathbf{d}^0$ and $\mathbf{d}^1$ are bounded by $\bar{B}$ and $\frac{Q}{\bar{B}}$, respectively, and $\mathbf{d}\in R_Q$. This decomposition method extends to various ciphertexts. For an RLWE ciphertext $\mathbf{C}=(\mathbf{a},\mathbf{b})$, its decomposition is:

$$\mathcal{D}_{rlwe}(\mathbf{C})=(\mathcal{D}(\mathbf{a}),\mathcal{D}(\mathbf{b}))=(\mathbf{C}^0,\mathbf{C}^1),$$

where $\mathbf{C}^0=(\mathbf{a}^0,\mathbf{b}^0)$ and $\mathbf{C}^1=(\mathbf{a}^1,\mathbf{b}^1)$. For a RLWE′ ciphertext $\dot{\mathbf{C}}=(\mathbf{C}_0,\mathbf{C}_1,\cdots,\mathbf{C}_{k-1})$, its decomposition is $\mathcal{D}_{rlwe'}(\dot{\mathbf{C}})=(\mathcal{D}_{rlwe}(\mathbf{C}_0),\mathcal{D}_{rlwe}(\mathbf{C}_1),\cdots,\mathcal{D}_{rlwe}(\mathbf{C}_{k-1}))=(\dot{\mathbf{C}}^0,\dot{\mathbf{C}}^1)$, where $\dot{\mathbf{C}}^0=(\mathbf{C}_0^0,\mathbf{C}_1^0,\cdots,\mathbf{C}_{k-1}^0)$ and $\dot{\mathbf{C}}^1=(\mathbf{C}_0^1,\mathbf{C}_1^1,\cdots,\mathbf{C}_{k-1}^1)$. For an RGSW ciphertext $\ddot{\mathbf{C}}=(\dot{\mathbf{C}}_0,\dot{\mathbf{C}}_1)$, its decomposition is:

$$\mathcal{D}_{rgsw}(\ddot{\mathbf{C}})=(\mathcal{D}_{rlwe'}(\dot{\mathbf{C}}_0),\mathcal{D}_{rlwe'}(\dot{\mathbf{C}}_1))=(\ddot{\mathbf{C}}^0,\ddot{\mathbf{C}}^1),$$

where $\ddot{\mathbf{C}}_0=(\dot{\mathbf{C}}_0^0,\dot{\mathbf{C}}_0^1)$ and $\ddot{\mathbf{C}}_1=(\dot{\mathbf{C}}_1^0,\dot{\mathbf{C}}_1^1)$. The composition of the decomposed RLWE ciphertext is:

$$\mathcal{C}_{rlwe}(\mathbf{C}^0,\mathbf{C}^1)=\mathbf{C}^0+\mathbf{C}^1\bar{B}=\mathbf{C},$$

where $\mathbf{C}^0$ and $\mathbf{C}^1$ are RLWE ciphertext bounded by $\bar{B}$ and $\frac{Q}{\bar{B}}$, respectively.

Error Analysis. This section elaborates on the decomposition method discussed earlier, focusing on numerical errors from using FFT with double precision for polynomial multiplication in bootstrapping and strategies to mitigate these errors. In bootstrapping, with $B<\bar{B}$, the multiplication of the rings $\mathbf{c}$ bounded by B and $\mathbf{d}$ with modulus Q is represented as $\mathbf{c}\cdot\mathbf{d}=\mathbf{c}\cdot\mathbf{d}^0+\mathbf{c}\cdot\mathbf{d}^1\bar{B}$, as shown in Algorithm 2. According to [8], the magnitude of coefficients in $\mathbf{c}\cdot\mathbf{d}^0$ is $t_0=\frac{B\cdot\bar{B}\sqrt{N}}{4}$, while that of $\mathbf{c}\cdot\mathbf{d}^1$ is represented as $t_1=\frac{B\cdot Q\sqrt{N}}{4\bar{B}}$. The induced FFT numerical error for t_i is $\lceil t_i\cdot\epsilon\rfloor$, where ϵ is the relative error depending on N and the FFT library. The overall FFT numerical error, denoted as e_{fft}, is given by:

$$e_{\text{fft}}=\lceil t_0\cdot\epsilon\rfloor+\lceil t_1\cdot\epsilon\rfloor\bar{B}.$$

To minimize e_{fft}, the criterion $t_1\epsilon<0.5$ must be met, indicating that the choice of $\bar{B}$ is crucial. When this criterion is satisfied, the numerical error simplifies to $e_{\text{fft}}=\lceil t_0\cdot\epsilon\rfloor$.

Algorithm 2: Ring Multiplication with Decomposition

input : Two rings $\mathbf{c},\mathbf{d}\in R_Q$, $\mathbf{c}$ bounded by B;
output: A ring $\mathbf{h}\in R_Q$

$(\mathbf{d}^0,\mathbf{d}^1)\leftarrow\mathcal{D}(\mathbf{d})$
$\widetilde{\mathbf{c}}\leftarrow\mathsf{FFT}(\mathbf{c}),\widetilde{\mathbf{d}^0}\leftarrow\mathsf{FFT}(\mathbf{d}^0),\widetilde{\mathbf{d}^1}\leftarrow\mathsf{FFT}(\mathbf{d}^1)$
for $j\leftarrow 0$ **to** 1 **do**
  for $i\leftarrow 0$ **to** $N/2-1$ **do**
    $\widetilde{\mathbf{h}^j}[i]\leftarrow\widetilde{\mathbf{c}}[i]\,\widetilde{\mathbf{d}^j}[i]$ //complex multiplication
  $\mathbf{h}^j\leftarrow\mathsf{FFT}^{-1}(\widetilde{\mathbf{h}^j})$
$\mathbf{h}\leftarrow\mathbf{h}^0+\mathbf{h}^1\bar{B},\mathbf{h}\leftarrow\mathbf{h}\mod Q$, **return** $\mathbf{h}$

3.2 Extension Method for Larger Modulus

The presented decomposition method facilitates support for a larger modulus by decomposing multiple components. This section delves into the extension of this method to accommodate a larger modulus. Given a ring $\mathbf{d}$, we decompose each coefficient $\mathbf{d}[i]$ using the bases $\{\bar{B}_0,\bar{B}_1,\cdots,\bar{B}_{k-1}\}$, such that: $\mathbf{d}[i]=\mathbf{d}^0[i]+\sum_{j=1}^{k-1}\left(\mathbf{d}^j[i]\cdot\prod_{t=0}^{j-1}\bar{B}_t\right)$. The multiplication of $\mathbf{c}$ by $\mathbf{d}$ is $\mathbf{c}\cdot\mathbf{d}=\mathbf{c}\cdot\mathbf{d}^0+\sum_{i=1}^{k-1}\left(\mathbf{c}\cdot\mathbf{d}^i\cdot\prod_{j=0}^{i-1}\bar{B}_j\right)$, where $\mathbf{d}^i$ is the ring bounded by $\bar{B}_i$. The magnitude of each coefficient t_i for $\mathbf{c}\cdot\mathbf{d}^i$ is $\frac{B\cdot\bar{B}_i\sqrt{N}}{4}$. The numerical error resulting from FFT-based polynomial multiplication is:

$$e=\lceil t_0\cdot\epsilon\rfloor+\sum_{i=1}^{k-1}\left(\lceil t_i\cdot\epsilon\rfloor\cdot\prod_{j=0}^{i-1}\bar{B}_j\right),$$

Algorithm 3: Blind Rotation for Decomposed ACC Using FFT

input : A ciphertext $\mathsf{C} \in$ RLWE; a vector $\vec{a} = (a[1], a[2], \cdots, a[n]), a[i] \in \mathbb{Z}_q$; the bootstrapping key $\{\mathbf{BTK}_{0i}, \mathbf{BTK}_{1i}\}_{i\in[1,n]}, \mathbf{BTK}_{ji} \in$ RGSW;
output : A refreshed ciphertext ACC $\in$ RLWE

1 $\mathrm{ACC} \leftarrow \mathsf{C}$, $(\mathrm{ACC}^0, \mathrm{ACC}^1) \leftarrow \mathcal{D}_{rlwe}(\mathrm{ACC})$
2 **for** $i \leftarrow 1$ **to** n **do**
3 $\quad (\mathbf{BTK}_{0i}^0, \mathbf{BTK}_{0i}^1) \leftarrow \mathcal{D}_{rgsw}(\mathbf{BTK}_{0i}$
4 $\quad (\mathbf{BTK}_{1i}^0, \mathbf{BTK}_{1i}^1) \leftarrow \mathcal{D}_{rgsw}(\mathbf{BTK}_{1i})$
5 $\quad \widetilde{\mathbf{BTK}_{0i}^0} \leftarrow \mathsf{FFT}(\mathbf{BTK}_{0i}^0)$, $\widetilde{\mathbf{BTK}_{0i}^1} \leftarrow \mathsf{FFT}(\mathbf{BTK}_{0i}^1)$
6 $\quad \widetilde{\mathbf{BTK}_{1i}^0} \leftarrow \mathsf{FFT}(\mathbf{BTK}_{1i}^0)$, $\widetilde{\mathbf{BTK}_{1i}^1} \leftarrow \mathsf{FFT}(\mathbf{BTK}_{1i}^1)$
7 $\quad \mathrm{ACC_d} \leftarrow \mathcal{G}_d(\mathrm{ACC}_0, \mathrm{ACC}_1)$,
8 $\quad \widetilde{\mathrm{ACC_d}} \leftarrow \mathsf{FFT}(\mathrm{ACC_d})$
9 $\quad$ **for** $j \leftarrow 0$ **to** 1 **do**
10 $\quad\quad \widetilde{\mathrm{ACC_t}^j} \leftarrow (X^{\widetilde{a[i]}} - 1)(\widetilde{\mathrm{ACC_d}} \cdot \widetilde{\mathbf{BTK}_{0i}^j}) + (X^{\widetilde{-a[i]}} - 1)(\widetilde{\mathrm{ACC_d}} \cdot \widetilde{\mathbf{BTK}_{1i}^j})$
11 $\quad \mathrm{ACC_t}^0 \leftarrow \mathsf{FFT}^{-1}(\widetilde{\mathrm{ACC_t}^0})$, $\mathrm{ACC_t}^1 \leftarrow \mathsf{FFT}^{-1}(\widetilde{\mathrm{ACC_t}^1})$
12 $\quad (\mathrm{ACC}^0, \mathrm{ACC}^1) \leftarrow \mathcal{D}_{add}((\mathrm{ACC}^0, \mathrm{ACC}^1), (\mathrm{ACC_t}^0, \mathrm{ACC_t}^1))$
13 $\mathrm{ACC} \leftarrow \mathcal{C}_{rlwe}(\mathrm{ACC}^0, \mathrm{ACC}^1)$, **return** ACC

where $\lceil \cdot \rfloor$ denotes rounding. It is imperative to ensure $t_i \cdot \epsilon < 0.5$ for $i \in [1, k-1]$. Upon satisfying this criterion, the precision loss is calculated as $e = \lceil t_0 \cdot \epsilon \rfloor$.

4 FFT-BASED FHEW/TFHE BOOTSTRAPPING WITH SMALL NUMERICAL ERROR

While the decomposition method discussed in § 3 can be seamlessly integrated into the bootstrapping operation, it might not offer the most optimal optimization for the bootstrapping process. We denote the modified bootstrapping as $\mathrm{Boots_d}[f]$.

In this paper, we detail how to integrate polynomial multiplication into FHEW/TFHE bootstrapping with the small numerical error optimally. In § 4.1, we detail how to modify bootstrapping when our method is introduced and optimize the modified parts. In § 4.2, we detail the error of input ciphertext for our proposed bootstrapping and that of refresh ciphertext after new bootstrapping.

4.1 Integrational Optimization

Incorporating the decomposition method has led to modifications in the blind rotation procedure, as outlined in Algorithm 4. There are three main adjustments:

(1) **Decomposition**: In the modified bootstrapping process, four main components require decomposition.

Initial RLWE Accumulator. The initial RLWE accumulator is decomposed into two RLWE accumulators based on $\bar{B}$ using the function $\mathcal{D}_{rlwe}(\cdot)$, as shown in line 1 of Algorithm 3.

Bootstrapping Key. The bootstrapping key is decomposed into two bootstrapping keys based on $\bar{B}$ using the function $\mathcal{D}_{rgsw}(\cdot)$, as shown in lines 3 to 6 of Algorithm 3. Notably, the decomposition of the bootstrapping key can be preprocessed for optimization.

Gadget Decomposition. Traditionally, gadget decomposition pertains to the decomposition of the RLWE accumulator based on B. In the context of modified bootstrapping, it involves the decomposed RLWE accumulator, represented as:

$$\mathcal{G}_d(\mathrm{ACC}^0, \mathrm{ACC}^1) = \mathrm{ACC_d}.$$

Here, $(\mathrm{ACC}^0, \mathrm{ACC}^1)$ is the decomposed RLWE accumulator based on $\bar{B}$, and $\mathrm{ACC_d} \in R_Q^{2d_g}$ is bounded by B.

Re-decomposition for Addition. During the addition of $(\mathrm{ACC}^0, \mathrm{ACC}^1)$ and $(\mathrm{ACC_t}^0, \mathrm{ACC_t}^1)$, re-decomposition based on $\bar{B}$ is necessary due to the lack of modular reduction in FFT operations. The series of polynomial multiplications and additions may cause the coefficients of $\mathrm{ACC_t}^0$ to exceed $\bar{B}$, and the coefficients of $\mathrm{ACC_t}^1$ to surpass $\frac{Q}{\bar{B}}$. The decomposition for this addition is denoted as:

$$\mathcal{D}_{add}((\mathrm{ACC}^0, \mathrm{ACC}^1), (\mathrm{ACC_t}^0, \mathrm{ACC_t}^1)) = (\hat{\mathrm{ACC}}^0, \hat{\mathrm{ACC}}^1).$$

Here, ACC^i and $\hat{\mathrm{ACC}}^i$ represent the RLWE ciphertexts bounded by $\bar{B}$, while $\mathrm{ACC_t}^i$ denotes the RLWE ciphertext.

(2) **ACC Update**: In lines 2 to 10 of Algorithm 3, the external product of the decomposed bootstrapping key and the decomposed RLWE ciphertext results in $\lceil \log_{\bar{B}} Q \rceil$ times more polynomial multiplications and additions compared to the original bootstrapping procedure.

(3) **Composition**: After n iterations of ACC updation, the decomposed RLWE ciphertext is reassembled into a single RLWE ciphertext using $\mathcal{C}_{rlwe}(\cdot)$, as shown in line 12 of Algorithm 3.

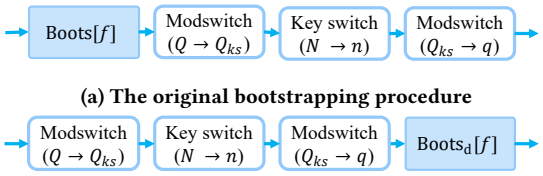
In our approach, the primary optimal functions are the gadget decomposition for the decomposed RLWE accumulator $\mathcal{G}_d(\cdot)$ and the re-decomposition for addition $\mathcal{D}_{add}(\cdot)$. Next, we will discuss the optimization strategies employed.

Optimization of Gadget Decomposition. The straightforward implementation of $\mathcal{G}_d(\cdot)$ reassembles the decomposed RLWE ciphertext before decomposing using B as the base. We propose a method that directly decomposes with B without reassembling. For $m \in \mathbb{Z}_Q$ decomposed into m_0 and m_1 with $\bar{B}$, we have $m = m_0 + m_1 \cdot \bar{B}$. Here, m_0 and m_1 correspond to coefficients from ACC^0 and ACC^1. Initially, m_0 is segmented into $(\bar{m}_0, \bar{m}_1, \cdots, \bar{m}_k)$ using B. If $\bar{m}_k < B$, m can be represented as: $m = \bar{m}_0 + \sum_{i=1}^{k-1} (\bar{m}_i \cdot B^i) + B^k \left(\bar{m}_k + \frac{m_1 \cdot \bar{B}}{B^k}\right)$. The term $(\bar{m}_k + \frac{m_1 \cdot \bar{B}}{B^k})$ undergoes further decomposition with B. We implement this decomposition using signed digit decomposition [1]. For $Q > 2^{64}$ and $\bar{B} < 2^{64}$, if $\frac{m_1 \cdot \bar{B}}{B^k} < 2^{64}$, values can be stored using 64-bit integers, making the method more efficient than using 128-bit integers.

Optimization of Re-decomposition for Addition. The function $\mathcal{D}_{add}(\cdot)$ adds two decomposed RLWE ciphertexts and re-decomposes the result using $\bar{B}$. This ensures ACC^0 coefficients are bounded by $\bar{B}$ and ACC^1 coefficients by $\frac{Q}{\bar{B}}$. Since polynomial multiplication in FFT format does not operate modulo Q, coefficients may exceed Q, requiring a modulo operation. Given that Q is a power of two, re-decomposition effectively performs a mod Q operation.

A naive approach adds $\mathrm{ACC_t}^0$ to ACC^0 and $\mathrm{ACC_t}^1$ to ACC^1, then re-decomposes $\mathrm{ACC}^0 + \mathrm{ACC}^1 \bar{B}$ using $\mathcal{D}_{rlwe}(\cdot)$. This is inefficient on 64-bit architectures due to the long integer operations required to mitigate numerical errors resulting from adding large numbers to small ones with double precision.

When Q is a power of two, we optimize by directly using decomposition without re-composition. For coefficients m_i from ACC^i and $\hat{m}_i$ from $\mathrm{ACC_t}^i$, after polynomial operations, $\hat{m}_0$ and $\hat{m}_1$ may



(a) The original bootstrapping procedure

(b) The modified bootstrapping procedure

Figure 1: Comparison of Bootstrapping Procedures.

exceed their bounds. We add $\hat{m}_0$ to m_0, divide by $\bar{B}$ to get the new m_0 (the remainder), and add the quotient to $(m_1 + \hat{m}_1)$, then divide by $\frac{Q}{\bar{B}}$ to get the new m_1 (the remainder).

We implement $\mathcal{D}_{add}(\cdot)$ using 64-bit floating-point operations to improve efficiency. To handle numerical errors from adding large and small numbers, we use two divisions: the first constrains $\hat{m}_i$'s range. These divisions are optimized with bitwise operations since the divisor is a power of two. For more details, see the full paper.

Modified Procedure of Bootstrapping. The bootstrapping procedure from [8] involves applying Boots$[f]$, followed by key switching (from (Q, N) to (Q, n)) and modulus switching (from (Q, n) to (q, n)). As proposed in [26], we add an initial modulus switch from (Q, N) to (Q_{ks}, N) before key switching, as depicted in Figure 1a. Here, $Q > Q_{\text{ks}} > q$ and $N > n$. Our modified procedure first switches the modulus from (Q, N) to (Q_{ks}, N), then performs key switching to (Q, n), followed by a final modulus switch to (q, n), and concludes with Boots$_{\text{d}}[f]$. This sequence maintains the modulus Q for the input ciphertext, allowing a larger plaintext space or more operations during processing. The modified procedure is shown in Figure 1b.

4.2 Error Analysis of the Proposed Bootstrapping

In this section, we analyze the error of LWE ciphertexts before and after the Boots$_{\text{d}}$ operation. The input LWE ciphertext for Boots$_{\text{d}}$ encapsulates cumulative errors from both key switch and modulus switch operations. Let σ_{A}^2 denote the variance attributed to ciphertext arithmetic operations preceding the bootstrapping procedure. Let σ_{MS1}^2, σ_{KS}^2, and σ_{MS2}^2 represent variances introduced by the first modulus switching, key switching, and subsequent modulus switching operations, respectively. Using the analytical framework from [26], we model the error in the input LWE ciphertext for Boots$_{\text{d}}$ as a Gaussian standard deviation:

$$\beta = \sqrt{\frac{q^2}{Q_{\text{ks}}^2}\left(\frac{Q_{ks}^2}{Q^2}\sigma_{\text{A}}^2 + \sigma_{\text{MS1}}^2 + \sigma_{\text{KS}}^2\right) + \sigma_{\text{MS2}}^2}.$$

After applying the Boots$_{\text{d}}[f]$ operation, the error in a refreshed ciphertext is modeled as a Gaussian standard deviation β_{ACC}. This standard deviation β_{ACC} includes contributions from σ_{GINX} arising from the GINX bootstrapping procedure [12] employing ternary CMUX optimization [2] and an approximate gadget decomposition method [21]. Additionally, due to the FFT-based implementation, an additional FFT bias Δ_{fft} stems from double-precision FFT computations. Therefore, we express the adjusted standard deviation β_{ACC} as:

$$\beta_{\text{ACC}} = \sigma_{\text{GINX}} + \Delta_{\text{fft}}.$$

Considering our adoption of an approximate gadget decomposition omitting the first digit, σ_{GINX} incorporates the variance of information loss associated with the discarded first digit $\text{Var}(\text{t}_0\text{m})$. Therefore, σ_{GINX} is formulated as:

$$\sigma_{\text{GINX}} = \sqrt{2u(d_{\text{g}} - 1)nN\frac{B^2}{6}\sigma^2 + 2un \cdot \text{Var}(\text{t}_0\text{m})}.$$

This paper treats the formula of σ_{GINX} as a black box; for detailed information, please refer to [26] and [21].

The FFT bias Δ_{fft} is computed from the LWE decryption $(b - \langle \vec{a} \cdot \vec{s} \rangle)$ for LWE ciphertext $(\vec{a}, b)$ after the Boots$_{\text{d}}[f]$ operation:

$$\Delta_{\text{fft}} = |(1 - \|s_N\|)e_{\text{fft}}|,$$

where $\|s_N\| \leq \sqrt{\frac{N}{2}}$ as noted in [26]. The term e_{fft}, representing the numerical error of each element in the LWE ciphertext $(\vec{a}, b)$ due to FFT with double precision, is expressed as:

$$e_{\text{fft}} = \sqrt{n}\left(\left\lceil \sqrt{2ud_{\text{g}}} \cdot t_0 \cdot \epsilon \right\rfloor + \left\lceil \sqrt{2ud_{\text{g}}} \cdot t_1 \cdot \epsilon \right\rfloor \bar{B}\right),$$

where $t_0 = \frac{B\cdot\bar{B}\sqrt{N}}{4}$ and $t_1 = \frac{B\cdot Q\sqrt{N}}{4\bar{B}}$, with ϵ representing the relative error and $\lceil \cdot \rfloor$ denoting the rounding operation from floating point to integer after the iFFT. Here, t_i represents the magnitude of each coefficient after multiplying two rings as noted in [8]. The numerical error from polynomial multiplication can be modeled as independent Gaussian distributions. Therefore, after $2ud_{\text{g}}$ polynomial multiplications accumulate during each ACC update, as shown in line 10 of Algorithm 3, the numerical error is $\sqrt{2ud_{\text{g}}} \cdot t_i \cdot \epsilon$. After n ACC updates, the numerical error is multiplied by $\sqrt{n}$. When $\sqrt{2ud_{\text{g}}} \cdot t_1 \cdot \epsilon < 0.5$, e_{fft} simplifies to:

$$e_{\text{fft}} = \sqrt{n}\left\lceil \sqrt{2ud_{\text{g}}} \cdot t_0 \cdot \epsilon \right\rfloor. \tag{1}$$

We also validate the constraint $\sqrt{2ud_{\text{g}}} \cdot t_1 \cdot \epsilon < 0.5$ through experiments involving 2048 polynomial multiplications. We measure the mean μ and standard deviation σ of $t_1 \cdot \epsilon$. The mean and standard deviation of $\sqrt{2ud_{\text{g}}} \cdot t_1 \cdot \epsilon$ are $\sqrt{2ud_{\text{g}}} \cdot \mu$ and $\sqrt{2ud_{\text{g}}} \cdot \sigma$, respectively. Within the parameter range in Table 2, this constraint is at least 18 standard deviations away from the mean. Based on the Gaussian distribution, being within 18 standard deviations corresponds to an overwhelming probability, thus ensuring that $\sqrt{2ud_{\text{g}}} \cdot t_1 \cdot \epsilon < 0.5$ almost always holds. Based on this approach, TFHE similarly ignores FFT-induced numerical error for smaller parameters. As shown in Figure 2, we observe that when $\bar{B}$ is slightly greater than $\frac{\sqrt{2ud_{\text{g}}N}\cdot B\cdot Q\cdot\epsilon}{2}$, *e.g.*, 2^{46}, the noise significantly decreases. When it exceeds 2^{46}, the noise increases again as it becomes proportional to $\bar{B}$ according to Equation 1. Therefore, $\bar{B}$ must be greater than $\frac{\sqrt{2ud_{\text{g}}N}\cdot B\cdot Q\cdot\epsilon}{2}$ but should not exceed this value too much.

5 EXPERIMENTAL EVALUATION

In this work, we extended the OpenFHE v1.1.0 framework [1] by integrating our proposed method and benchmarking it against the existing bootstrapping technique in OpenFHE v1.1.0. OpenFHE was compiled using g++ version 11.4.0 with the −ffast − math flag and configured with NATIVE_SIZE = 128 and WITH_NATIVEOPT = ON. FFTW v3.3.10 [11] was incorporated for FFT implementation

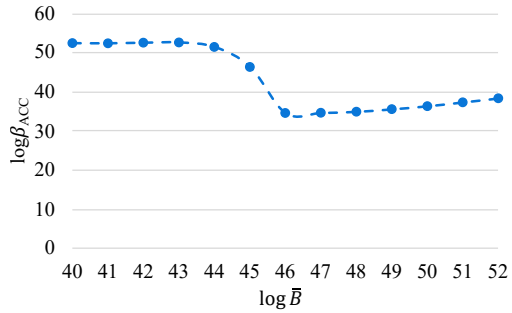


Figure 2: The standard deviation of the bootstrapping noise β_{ACC} varies with decomposition base $\bar{B}$.

to accelerate polynomial multiplication. Experiments were performed on an AMD Ryzen Threadripper PRO 5975WX processor with 32 cores and 256GB of RAM, running Ubuntu 22.04.2 LTS, using OpenMP 4.5 for parallel computation, with one thread per bootstrapping operation.

§ 5.1 discusses the performance analysis of OpenFHE and other implementations. Our method is suitable for large number analysis and high-precision machine learning applications. In § 5.2 and § 5.3, we validate our method's correctness by comparing execution time and accuracy with OpenFHE in secure decision tree and secure DNN inference applications, using the same larger ciphertext modulus. In these experiments, our method utilizes FFTW with AVX optimization. Accuracy is measured as the ratio of correctly predicted classes, with homomorphic inference accuracy denoted as HE $\mathcal{A}$ and plaintext inference accuracy as Pt. $\mathcal{A}$.

5.1 Efficiency Evaluation

In the efficiency analysis for supporting larger modulus, our method is compared with three different bootstrapping implementations: the one in the OpenFHE library, the single long double approach, and the Residue Number System (RNS) scheme [13].

Comparison with OpenFHE. Functional bootstrapping is an advanced form of bootstrapping that constructs a lookup table for a specific function. In FHEW/TFHE, both bootstrapping and functional bootstrapping are consistent processes, meaning that improvements in bootstrapping also enhance functional bootstrapping. We evaluated our method using the EvalFunc operation [1], a general functional bootstrapping method for evaluating arbitrary functions. The comparative results with a batch size of 2,048 are presented in Table 2, using parameters $n = 1,305$, $p = 16$, $q = 4,096$, and $N = 4,096$, meeting 128-bit security with a modulus between 2^{62} and 2^{76}. Here, $\beta_{\mathbf{std}}$ represents the standard deviation of the error introduced by functional bootstrapping. Smaller $\beta_{\mathbf{std}}$ values allow more homomorphic operations. For a ciphertext with modulus Q, message modulus P, and noise β, it supports either $\lfloor \log_P \frac{Q}{2\beta} - 1 \rfloor$ ciphertext-plaintext multiplications or $\lfloor \log_{P\sqrt{\bar{N}}} \frac{Q}{2P\cdot\beta} \rfloor$ multiplications of a ciphertext vector (dimension $\bar{N}$) with a plaintext matrix ($\bar{N} \times \bar{N}$). For example, with $Q = 2^{76}$ and $\bar{N} = 2048$, when $\beta_{\mathbf{std}} = 2^{51}$, it supports either five ciphertext-plaintext multiplications or two ciphertext-vector with plaintext-matrix multiplications. If $\beta_{\mathbf{std}}$ reduces to 2^{38}, it supports either eight ciphertext-plaintext multiplications or three ciphertext-vector with plaintext-matrix multiplications.

Table 2: Execution time comparison of EvalFunc using a single CPU thread (1T) and 32 CPU threads (32T). Bold values indicate the execution times of our method under the same noise level as OpenFHE [1].

Method	$\log Q$	d_g	$\log \bar{B}$	$\log \beta_{\mathbf{std}}$	1T (s)	32T (s)
OpenFHE	64	3	N/A	34	8.24 (1x)	0.28 (1x)
Ours	64	3	44	34	**0.98 (8.4x)**	**0.04 (7x)**
Ours	64	4	44	29	1.25 (6.6x)	0.05 (5.6x)
OpenFHE	66	3	N/A	35	8.54 (1x)	0.29 (1x)
Ours	66	3	46	35	**0.95 (9x)**	**0.04 (7.3x)**
Ours	66	4	46	30	1.23 (6.9x)	0.05 (5.8x)
OpenFHE	68	3	N/A	35	8.53 (1x)	0.29 (1x)
Ours	68	3	49	36	0.97 (8.8x)	0.04 (7.3x)
Ours	68	4	49	31	**1.13 (7.5x)**	**0.05 (5.8x)**
OpenFHE	72	3	N/A	37	8.55 (1x)	0.29 (1x)
Ours	72	3	54	42	0.95 (9x)	0.04 (7.3x)
Ours	72	4	53	36	**1.11 (7.7x)**	**0.05 (5.8x)**
OpenFHE	76	3	N/A	38	8.52 (1x)	0.29 (1x)
Ours	76	3	61	51	0.97 (8.8x)	0.04 (7.3x)
Ours	76	4	54	38	**1.18 (7.2x)**	**0.05 (5.8x)**

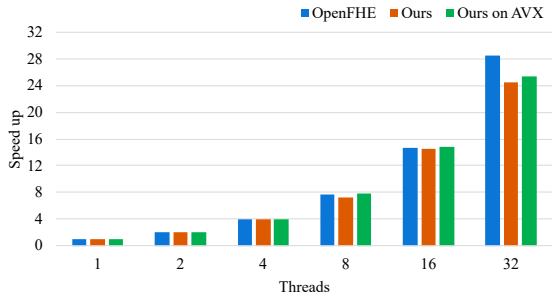


Figure 3: Speedup comparison of bootstrapping for OpenFHE, our method, and our method using FFTW with AVX across various thread counts.

In Table 2, for the modulus Q ranging from 2^{64} to 2^{66} under the same d_g, the noise $\beta_{\mathbf{std}}$ of our method is comparable to that of OpenFHE. This indicates that the noise from the approximate gadget decomposition is similar to the numerical error from FFT with double precision. When Q exceeds 2^{66} under the same d_g, the error from FFT with double precision becomes the dominant factor. We can increase d_g to achieve similar noise levels, but this increases execution time and the size of the bootstrapping key.

Under similar noise conditions, our method shows a 7.2x to 9x speedup in single-threaded mode and a 5.8x to 7.3x speedup with 32 threads, compared to OpenFHE for Q between 2^{64} and 2^{76}. Using Advanced Vector Extensions (AVX) optimized FFTW, our method's performance improves by 14% to 17% in single-threaded execution and by 25% to 33% with 32 threads. Figure 3 shows that the speedup of our method using 32 threads is lower than that of OpenFHE. This is due to our larger bootstrapping key, which uses complex numbers with double precision, compared to OpenFHE's bootstrapping key using 128-bit integers. Consequently, our method hits the memory bottleneck sooner, especially noticeable with 32 threads.

Comparison with Long Double. Using a long double is inefficient on 64-bit architectures due to their 128-bit representation,

Table 3: Comparison of secure decision tree evaluation time per instance against OpenFHE [1] on the Iris dataset using 32 CPU threads.

Method	$\log Q$	d_g	HE $\mathcal{A}$.	Pt. $\mathcal{A}$.	Time (s)
OpenFHE	66	3	98%	100%	18.39 (1X)
Ours on AVX	66	3	98%	100%	1.84 (10X)

limiting speed improvements. For example, with a modulus of 2^{66}, the speedup is only 2.2x in single-threaded mode, while our method achieves a 9x speed improvement without AVX optimization. Our method, which decomposes a large number into two doubles, is faster than using two 64-bit doubles to simulate a 128-bit long double. This efficiency is because our approach uses only one double for ring coefficients generated by gadget decomposition, offering greater flexibility and efficiency compared to using two 64-bit doubles to mimic a 128-bit long double.

Comparison with the RNS Scheme. The Residue Number System (RNS) is a number representation system that decomposes a large number into smaller modulus to simplify arithmetic operations, often combined with NTT. In comparison, while our method's ACC addition after the inverse transform is slower than that of RNS, it offers three key advantages. Firstly, our method uses 64-bit operations for gadget decomposition, whereas RNS requires 128-bit operations. Secondly, we perform only half as many FFT operations as the NTT operations required by RNS. Thirdly, in FFT/NTT format, our method handles polynomials with half the dimension of those in RNS. Consequently, with similar noise levels, our method is estimated to be faster than RNS for ciphertext modulus $Q \leq 2^{76}$. Detailed comparisons are provided in the full version of this paper.

5.2 Evaluation on Secure Decision Tree Models

Secure decision trees protect data privacy during inference, crucial for applications like medical diagnostics and financial evaluations. Recent studies, including [15, 24, 27], have explored the use of FHE for this purpose. We assessed our FHE-based decision tree using the Iris dataset [10] from the UCI repository. Table 3 compares our method with the OpenFHE implementation. We used the non-interactive secure decision tree algorithm from [15], which encrypts only the input data and processes the rest in plaintext. This algorithm involves $O(N)$ EvalFunc, where N is the number of decision tree nodes. In our experiments, we used EvalFunc with the message modulus $p = 2^{11}$. Our model has eight internal nodes and nine leaves. The Iris dataset contains three classes with 50 instances each, and each instance has four features. We randomly selected 100 instances for training and 50 for testing. As shown in Table 3, our method achieves a 10x speed improvement over the OpenFHE implementation while maintaining equivalent accuracy in homomorphic inference.

5.3 Evaluation on Secure DNN Inference

FHE-based encrypted DNN inference enables secure DNN inference within an encrypted domain, offering a promising solution for data privacy. This approach has been extensively studied [3, 17–19]. We assessed our method using the MNIST [20] and Fashion-MNIST [28] datasets, comparing it with OpenFHE as shown in Tables 4 and 5, respectively. We used the FHE parameters and model architecture from [18], with a message modulus of $p = 2^{16}$ and EvalFunc for the ReLU activation function.

Table 4: Comparison of secure DNN inference time per instance against OpenFHE [1] on MNIST dataset using 32 CPU threads.

Method	$\log Q$	d_g	HE $\mathcal{A}$.	Pt. $\mathcal{A}$.	Time (s)
OpenFHE	66	3	96.63%	97%	11.39 (1X)
Ours on AVX	66	3	96.69%	97%	2.35 (4.8X)

Table 5: Comparison of secure DNN inference time per instance against OpenFHE [1] on Fashion-MNIST dataset using 32 CPU threads.

Method	$\log Q$	d_g	HE $\mathcal{A}$.	Pt. $\mathcal{A}$.	Time (s)
OpenFHE	66	3	88.71%	89%	48.3 (1X)
Ours on AVX	66	3	88.72%	89%	9.47 (5.1X)

The MNIST dataset contains 28x28 pixel grayscale images of handwritten digits, categorized into ten classes (0-9), with 60,000 instances (50,000 for training and 10,000 for testing). Our model consists of an input layer with 784 neurons (28x28 pixels), a fully connected hidden layer with 30 neurons using the ReLU activation function, and a fully connected output layer with ten neurons. As shown in Table 4, our method shows a 4.8x speedup over OpenFHE with comparable accuracy in homomorphic inference.

The Fashion-MNIST dataset contains 28x28 pixel grayscale images of various clothing items categorized into ten classes (e.g., t-shirts, trousers, pullovers, dresses). It has 70,000 instances (60,000 for training and 10,000 for testing). Our model for this dataset consists of an input layer with 784 neurons, a fully connected hidden layer with 128 neurons using the ReLU activation function, and a fully connected output layer with ten neurons. As shown in Table 5, our method achieves a 5.1x speedup over OpenFHE while maintaining similar accuracy in homomorphic inference.

6 CONCLUSION

We integrate FHEW-like bootstrapping with large-number decomposition to achieve efficient FFT-based bootstrapping with minimal numerical error. Our method improves Evalfunc operation speed by 5-7x over OpenFHE while maintaining similar error levels. Specifically, in secure decision trees and neural network inference, our method is 10x and 5x faster than OpenFHE, respectively, with comparable accuracy. Additionally, our approach can expand the message space for methods like CKKS and FHEW/TFHE [15] and large precision sign function [22], suggesting broader applicability. Finally, our implementation is well-suited for parallelization, making it ideal for hardware acceleration, including GPUs.

ACKNOWLEDGMENTS

This work was primarily conducted at Inventec Corporation. We gratefully acknowledge the support of the National Science and Technology Council, Taiwan, under grant NSTC 112-2221-E-002 -159 -MY3 and 113-2634-F-002-001 -MBK. Feng-Hao Liu would like to thank NSF Career Award CNS-2402031. Their support was instrumental in developing critical preliminary results for this work.

REFERENCES

[1] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, et al. 2022. Openfhe: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 53–63.

[2] Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. 2022. FINAL: Faster FHE Instantiated with NTRU and LWE. In *ASIACRYPT 2022, Part II (LNCS, Vol. 13792)*, Shweta Agrawal and Dongdai Lin (Eds.). Springer, Cham, 188–215. https://doi.org/10.1007/978-3-031-22966-4_7

[3] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. 2018. Fast Homomorphic Evaluation of Deep Discretized Neural Networks. In *CRYPTO 2018, Part III (LNCS, Vol. 10993)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, Cham, 483–512. https://doi.org/10.1007/978-3-319-96878-0_17

[4] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS 2012*, Shafi Goldwasser (Ed.). ACM, 309–325. https://doi.org/10.1145/2090236.2090262

[5] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *ASIACRYPT 2017, Part I (LNCS, Vol. 10624)*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer, Cham, 409–437. https://doi.org/10.1007/978-3-319-70694-8_15

[6] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In *ASIACRYPT 2016, Part I (LNCS, Vol. 10031)*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.). Springer, Berlin, Heidelberg, 3–33. https://doi.org/10.1007/978-3-662-53887-6_1

[7] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology* 33, 1 (Jan. 2020), 34–91. https://doi.org/10.1007/s00145-019-09319-x

[8] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *EUROCRYPT 2015, Part I (LNCS, Vol. 9056)*, Elisabeth Oswald and Marc Fischlin (Eds.). Springer, Berlin, Heidelberg, 617–640. https://doi.org/10.1007/978-3-662-46800-5_24

[9] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144. https://eprint.iacr.org/2012/144.

[10] R. A. Fisher. 1988. Iris. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C56C76.

[11] Matteo Frigo and Steven G Johnson. 2005. The design and implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231.

[12] Nicolas Gama, Malika Izabachène, Phong Q. Nguyen, and Xiang Xie. 2016. Structural Lattice Reduction: Generalized Worst-Case to Average-Case Reductions and Homomorphic Cryptosystems. In *EUROCRYPT 2016, Part II (LNCS, Vol. 9666)*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer, Berlin, Heidelberg, 528–558. https://doi.org/10.1007/978-3-662-49896-5_19

[13] Harvey L Garner. 1959. The residue number system. In *Papers presented at the the March 3-5, 1959, western joint computer conference*. 146–153.

[14] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *41st ACM STOC*, Michael Mitzenmacher (Ed.). ACM Press, 169–178. https://doi.org/10.1145/1536414.1536440

[15] Wen jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. 2021. PEGASUS: Bridging Polynomial and Non-polynomial Evaluations in Homomorphic Encryption. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1057–1073. https://doi.org/10.1109/SP40001.2021.00043

[16] Jakub Klemsa. 2021. Fast and error-free negacyclic integer convolution using extended fourier transform. In *International Symposium on Cyber Security Cryptography and Machine Learning*. Springer, 282–300.

[17] Kamil Kluczniak and Leonard Schild. 2023. FDFB: Full Domain Functional Bootstrapping Towards Practical Fully Homomorphic Encryption. *IACR TCHES* 2023, 1 (2023), 501–537. https://doi.org/10.46586/tches.v2023.i1.501-537

[18] Yu-Te Ku, Feng-Hao Liu, Yu Xiao, Ming-Ching Chang, Chih-Fan Hsu, I-Ping Tu, Shih-Hao Hung, and Wei-Chao Chen. 2024. Efficient Third Generation FHE Based Non-Interactive Encrypted DNN Inference with FHE-Aware Training. In *Unpublished Manuscript (2024)*.

[19] Kwok-Yan Lam, Xianhui Lu, Linru Zhang, Xiangning Wang, Huaxiong Wang, and Si Qi Goh. 2023. Efficient fhe-based privacy-enhanced neural network for ai-as-a-service. *Cryptology ePrint Archive* (2023).

[20] Yann LeCun, Corinna Cortes, and CJ Burges. 2010. MNIST handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* 2 (2010).

[21] Yongwoo Lee, Daniele Micciancio, Andrey Kim, Rakyong Choi, Maxim Deryabin, Jieun Eom, and Donghoon Yoo. 2023. Efficient FHEW Bootstrapping with Small Evaluation Keys, and Applications to Threshold Homomorphic Encryption. In *EUROCRYPT 2023, Part III (LNCS, Vol. 14006)*, Carmit Hazay and Martijn Stam (Eds.). Springer, Cham, 227–256. https://doi.org/10.1007/978-3-031-30620-4_8

[22] Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. 2022. Large-Precision Homomorphic Sign Evaluation Using FHEW/TFHE Bootstrapping. In *ASIACRYPT 2022, Part II (LNCS, Vol. 13792)*, Shweta Agrawal and Dongdai Lin (Eds.). Springer, Cham, 130–160. https://doi.org/10.1007/978-3-031-22966-4_5

[23] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. 2012. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *44th ACM STOC*, Howard J. Karloff and Toniann Pitassi (Eds.). ACM Press, 1219–1234. https://doi.org/10.1145/2213977.2214086

[24] Wenjie Lu, Jun-Jie Zhou, and Jun Sakuma. 2018. Non-interactive and Output Expressive Private Comparison from Homomorphic Encryption. In *ASIACCS 18*, Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim (Eds.). ACM Press, 67–74. https://doi.org/10.1145/3196494.3196503

[25] Chiara Marcolla, Victor Sucasas, Marc Manzano, Riccardo Bassoli, Frank HP Fitzek, and Najwa Aaraj. 2022. Survey on fully homomorphic encryption, theory, and applications. *Proc. IEEE* 110, 10 (2022), 1572–1609.

[26] Daniele Micciancio and Yuriy Polyakov. 2021. Bootstrapping in FHEW-like cryptosystems. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 17–28.

[27] Anselme Tueno, Yordan Boev, and Florian Kerschbaum. 2020. Non-interactive private decision tree evaluation. In *Data and Applications Security and Privacy XXXIV: 34th Annual IFIP WG 11.3 Conference, DBSec 2020, Regensburg, Germany, June 25–26, 2020, Proceedings 34*. Springer, 174–194.

[28] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *CoRR* abs/1708.07747 (2017). arXiv:1708.07747 http://arxiv.org/abs/1708.07747