cryptography

MDPI

Article

Homomorphic ReLU with Full-Domain Bootstrapping

Yuqun Lin [1], Yi Huang [2], Xiaomeng Tang [2], Jingjing Fan [2,3]*, Qifei Xu [1], Zoe-Lin Jiang [2], Xiaosong Zhang [1] and Junbin Fang [4]

[1] The School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China; 202412081703@std.uestc.edu.cn (Y.L.); 202512081814@std.uestc.edu.cn (Q.X.); johnsonzxs@uestc.edu.cn (X.Z.)
[2] The School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen 518055, China; huangyi@stu.hit.edu.cn (Y.H.); 220110916@stu.hit.edu.cn (X.T.); zoeljiang@hit.edu.cn (Z.-L.J.)
[3] The School of Computing and Data Science, University of Hong Kong, Hong Kong
[4] Department of Optoelectronic Engineering, Jinan University, Guangzhou 510632, China; tjunbinfang@jnu.edu.cn
* Correspondence: jjfan@cs.hku.hk

Abstract

Fully homomorphic encryption (FHE) offers a promising solution for privacy-preserving machine learning by enabling arbitrary computations on encrypted data. However, the efficient evaluation of non-linear functions—such as the ReLU activation function over large integers—remains a major obstacle in practical deployments, primarily due to high bootstrapping overhead and limited precision support in existing schemes. In this paper, we propose LargeIntReLU, a novel framework that enables efficient homomorphic ReLU evaluation over large integers (7–11 bits) via full-domain bootstrapping. Central to our approach is a signed digit decomposition algorithm, SignedDecomp, that partitions a large integer ciphertext into signed 6-bit segments using three new low-level primitives: LeftShift, HomMod, and CipherClean. This decomposition preserves arithmetic consistency, avoids cross-segment carry propagation, and allows parallelized bootstrapping. By segmenting the large integer and processing each chunk independently with optimized small-integer bootstrapping, we achieve homomorphic ReLU with full-domain bootstrapping, which significantly reduces the total number of sequential bootstrapping operations required. The security of our scheme is guaranteed by TFHE. Experimental results demonstrate that the proposed method reduces the bootstrapping cost by an average of 28.58% compared to state-of-the-art approaches while maintaining 95.2% accuracy. With execution times ranging from 1.16 s to 1.62 s across 7–11 bit integers, our work bridges a critical gap toward a scalable and efficient homomorphic ReLU function, which is useful in privacy-preserving machine learning. Furthermore, an end-to-end encrypted inference test on a CNN model with the MNIST dataset confirms its practicality, achieving 88.85% accuracy and demonstrating a complete pipeline for privacy-preserving neural network evaluation.

Keywords: fully homomorphic encryption; torus; homomorphic ReLU activation; integer bootstrapping

Check for updates

Academic Editor: Josef Pieprzyk

Received: 30 December 2025
Revised: 6 February 2026
Accepted: 16 March 2026
Published: 24 March 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

The growing reliance on cloud computing and outsourced data processing has intensified the demand for robust privacy-preserving computation, particularly for sensitive applications like machine learning. Fully Homomorphic Encryption (FHE) has emerged

as a foundational solution to this challenge, enabling arbitrary computations to be performed directly on encrypted data without decryption [1]. This unique capability provides an unparalleled level of security in distributed environments by ensuring that sensitive information remains protected throughout the computation life-cycle [2]. The development of FHE techniques is especially critical for domains such as cloud-based ML services [3,4], where enabling secure neural network evaluation while addressing challenges like computational complexity and ciphertext expansion remains an active research area [5].

Among FHE schemes, Torus-Based Fully Homomorphic Encryption (TFHE) [6] excels in evaluating non-linear functions (e.g., ReLU) via efficient gate bootstrapping. However, a significant limitation arises from its native support for only Boolean or small integer data. Implementing arithmetic operations on integers using bit-wise TFHE requires constructing complex circuits, leading to a linear overhead of $O(n)$ bootstrappings for an n-bit operation, which is highly inefficient [7].

The framework in [7] demonstrates the potential of bit-wise TFHE by leveraging its native Boolean operations to implement ReLU activations and max-pooling directly, avoiding polynomial approximations that degrade inference accuracy. To address the inefficiency of homomorphic matrix multiplications, SHE [7] employs logarithmic quantization, converting multiplications into shift and accumulation operations that are more efficient under TFHE. This approach highlights how bit-wise TFHE can be optimized for specific workloads, yet general-purpose integer arithmetic remains challenging due to the circuit-level emulation required.

Subsequent research has explored integer-wise TFHE variants to reduce overhead. Specifically, Bourse et al. [8] enabled small integer (<6 bits) encryption supporting addition and sign functions, while Okada et al. [9] extended this to multiplication via MultbyBin—though limited to positive integers. FDFB [10] and pseudo-function decomposition [11] expanded domain support but incurred high bootstrapping costs. Yang et al. [12] reduced bootstrapping to two, yet still lacked support for negative integers. Against this backdrop, Huang et al. [13] made a key advance, solving core limitations of integer-wise TFHE (negacyclicity range constraints) while achieving the first precision-agnostic ReLU, a critical step toward practical encrypted neural networks. However, their approach efficiently supports only integers of up to 6 bits. This limitation stems from the LWE ciphertext constraint: under current security and correctness guarantees, the plaintext space is bounded to 6 bits. Supporting larger integers (e.g., 7–11 bits) is essential for real-world applications such as high-precision neural networks. More recently, Loris [14] proposed parameter optimization techniques to support larger integer precision in TFHE, yet their method still faces efficiency challenges when scaling to 7–11 bits due to the high cost of bootstrapping operations.

Table 1 provides a comparative overview of key TFHE-based schemes, highlighting their limitations in supporting large integers and efficient bootstrapping. As shown below, existing approaches either lack support for negative integers [9], are limited to 6-bit precision [13], or incur high bootstrapping costs when scaling to larger integers [10,14].

In this context, the 6-bit threshold is chosen as the critical point between small and large integers because, while ensuring computational correctness, LWE-based ciphertexts can natively represent only up to 6 bits. To handle larger integers without loss of generality, we adopt a decomposition strategy: an 11-bit integer (composed of 1 sign bit and 10 magnitude bits) is split into two 6-bit segments. Each segment itself contains 1 sign bit and 5 magnitude bits. This representation ensures consistent sign handling across segments and allows arbitrary-precision integer support through recursive decomposition.

Table 1. Comparison of torus-based homomorphic encryption schemes.

Scheme	Data Type	Max Precision	Negative Integers	Bootstrapping Cost
Chillotti [6]	Bit	1 bit	Limited	$O(n)$ for n-bit operations
Okada [9]	Integer	4 bits	No	$O(2^n)$ for n-bit integer division
FDFB [10]	Integer	6 bits	Yes	$n+1$
Huang [13]	Integer	6 bits	Yes	Precision-agnostic
Loris [14]	Integer	n bits	Yes	$\lceil \frac{n-1}{5} \rceil \times 6 + n - 7$
Ours	Integer	n bits	Yes	$\lceil \frac{n-1}{5} \rceil \times 4 + n - 5$

In Table 1, n represents the bit precision of the integer data type. For example, $n = 8$ indicates 8-bit integers supporting values from 0 to 255 (unsigned) or -128 to 127 (signed). The cost expressions show how bootstrapping complexity scales with increasing precision requirements.

1.1. Our Contribution

To bridge this gap, we propose LargeIntReLU, the first homomorphic ReLU algorithm supporting full-domain bootstrapping for large integers (7–11 bits). Our core innovations address the limitations of prior work:

- Signed Decomposition Algorithm SignedDecomp. We design three atomic operations—(a) LeftShift to eliminate carrying interference; (b) HomMod to perform signed modular reduction preserving arithmetic consistency; (c) CipherClean to remove low-bit noise—which together enable the SignedDecomp algorithm. This algorithm seamlessly splits large integer ciphertexts into signed 6-bit chunks without cross-segment carry propagation.
- Optimized ReLU Framework LargeIntReLU with Precision-Consistent Bootstrapping. We integrate the SignedDecomp algorithm with an optimized small-integer ReLU bootstrapping scheme [13], introducing two key optimizations: (1) strategic use of PBSManyLUT [15] to minimize bootstrapping operations for integers $\leq$10 bits, and (2) redefinition of test polynomial coefficients to ensure uniform slot allocation across segments of varying precision. These optimizations collectively enable correct and efficient ReLU evaluation over decomposed ciphertexts while preventing computational errors when the highest-order segment has less than 6 bits of precision.
- Comprehensive Performance Advantage. Through rigorous analysis and experimentation, we demonstrate that our approach significantly reduces the bootstrapping cost (by an average of 28.58%) compared to state-of-the-art approaches while maintaining 95.2% accuracy.

1.2. Roadmap

In this paper, we establish foundational knowledge, detailing TFHE mechanisms, integer bootstrapping, and limitations of radix-based decomposition in Section 2. Building upon these preliminaries, Section 3 introduces our signed decomposition framework, presenting the novel LeftShift, HomMod, and CipherClean primitives that collectively enable error-free splitting of large integers into signed 6-bit chunks. Section 4 then integrates this decomposition with optimized small-integer ReLU bootstrapping to construct the complete LargeIntegerReLU algorithm for full-domain evaluation. Critical performance analysis follows in Section 5, where we rigorously validate correctness, security, and efficiency. Finally, Section 6 concludes with broader impacts while acknowledging limitations and future directions for homomorphic activation functions.

2. Preliminaries

This chapter mainly introduces the relevant theoretical foundations of fully homomorphic encryption (FHE) and the algorithms for FHE. It primarily includes theoretical foundations such as the LWE hard problem, the torus-based encryption algorithm based on the LWE hard problem, and technical foundations such as the bit-wise fully homomorphic encryption algorithm, the integer-wise fully homomorphic encryption algorithm, the homomorphic multiplication algorithm supporting integers, and the large integer ciphertext decomposition algorithm based on radix.

2.1. Notation

Let $\mathcal{D}$ be a distribution over some finite set $\mathcal{S}$, then we use $x \leftarrow \mathcal{D}$ to denote that x is chosen from $\mathcal{D}$ and $x \xleftarrow{U} \mathcal{D}$ to denote that x is chosen uniformly random from $\mathcal{D}$. We denote column vectors and matrices by bold letters ($\mathbf{a}$ for a vector and $\mathbf{A}$ for a matrix). We will use $\|\mathbf{a}\|$ to denote the Euclidean norm of $\mathbf{a}$. And we will use $\|\mathbf{a}\|_\infty = \max_i |a_i|$ to denote the infinite norm of vector $\mathbf{a}$. We will use $\lfloor x \rfloor$ to denote the largest integer smaller or equal to x within the torus representation, $\lceil x \rceil$ to denote the smallest integer larger than or equal to x with in the torus operation, and $\lfloor x \rceil$ to denote to rounding to the nearest integer within the torus representation.

2.2. From TLWE to Torus-Based Encryption

The security of most modern Fully Homomorphic Encryption (FHE) schemes, including TFHE, rests on the hardness of lattice-based problems. We begin by introducing the Torus Learning With Errors (TLWE) problem, the core cryptographic assumption underlying Torus-based encryption schemes. Subsequently, we introduce the torus-based encryption schemes, building upon this foundation.

2.2.1. The Hardness of the Torus Learning with Errors (TLWE) Problem

The cryptographic strength of TLWE is rooted in the computational hardness of the Learning With Errors (LWE) problem, first introduced by Regev [16]. Let the integer $n \geq 1$ be a security parameter, $\mathbb{R}$ be the set of real numbers, $\mathbb{Z}$ be the set of integers, and $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ be the torus (the set of real numbers modulo 1). Let χ be an error (or noise) probability distribution over $\mathbb{R}^+$, and $\mathbf{sk} \in \mathbb{B}^n$ be a secret key vector where $\mathbb{B} \in \{0, 1\}$. The notation $\mathbf{sk} \xleftarrow{U} S$ denotes uniformly random sampling of $\mathbf{sk}$ from set S, and $e \leftarrow \chi$ denotes sampling an error e from the distribution χ.

The TLWE distribution $\mathrm{TLWE}_{n,\chi,\mathbf{sk}}$ consists of ciphertexts of the form $(\mathbf{a}, b)$, where $\mathbf{a} \xleftarrow{U} \mathbb{T}^n$ is a random vector and $b = \mathbf{a} \cdot \mathbf{sk} + e \in \mathbb{T}$ is a noisy inner product, with error $e \leftarrow \chi$. The TLWE problem manifests in two core forms:

TLWE Search Problem: Given multiple samples from $\mathrm{TLWE}_{n,\chi,\mathbf{sk}}$ where $\mathbf{sk} \xleftarrow{U} S$ and $S \in \mathbb{B}^n$, the goal is to recover the secret key vector $\mathbf{sk}$. TLWE Decisional Problem: Given samples that are either from $\mathrm{TLWE}_{n,\chi,\mathbf{sk}}$ or drawn uniformly from $\mathbb{T}^n \times \mathbb{T}$, the goal is to distinguish between the two cases. The search and decisional variants of the TLWE problem are of the same complexity. Chillotti et al. [6] proved that solving TLWE problem is at least as hard as solving worst-case instances of certain NP-hard problems.

2.2.2. Torus-Based Encryption Schemes

Torus-based encryption schemes define several core ciphertext types, with TLWE being a fundamental one; others include TRLWE (Torus Ring Learning With Errors) and TRGSW (Torus Ring Gentry-Sahai-Waters), each enabling different functionalities. It is also worth noting that in TLWE, the error term is commonly referred to as noise.

https://doi.org/10.3390/cryptography10020021

The TLWE encryption scheme provides the fundamental encryption mechanism and includes the following procedures:

- $(n, \chi) \leftarrow \mathsf{TLWE.Setup}(\lambda)$: On input a security parameter λ, output an LWE dimension $n = n(\lambda)$ and an error distribution $\chi = \chi(\lambda)$.
- $\mathbf{sk} \leftarrow \mathsf{TLWE.KeyGen}(n)$: On input n, output a secret key $\mathbf{sk} \xleftarrow{U} \mathbb{B}^n$.
- $\mathbf{c} \leftarrow \mathsf{TLWE.Enc}(\mathbf{sk}, m)$: On input $\mathbf{sk}$ and a plaintext $m \in \mathbb{T}$, samples $\mathbf{a} \xleftarrow{U} \mathbb{T}^n$, $e \leftarrow \chi$, compute $b = \mathbf{a} \cdot \mathbf{sk} + e + m$, and return a ciphertext $\mathbf{c} = (\mathbf{a}, b) \in \mathbb{T}^{n+1}$.
- $m \leftarrow \mathsf{TLWE.Dec}(\mathbf{sk}, \mathbf{c})$: On input $\mathbf{sk}$ and $\mathbf{c} = (\mathbf{a}, b)$, compute and return the plaintext $m = \lfloor b - \mathbf{a} \cdot \mathbf{sk} \rceil$.

The TRLWE encryption scheme extends TLWE to polynomial rings, enabling more efficient bootstrapping and functional operations:

- $(N, \chi) \leftarrow \mathsf{TRLWE.Setup}(\lambda)$: On input λ, output a polynomial degree $N = N(\lambda)$ and an error distribution χ.
- $\mathbf{sk} \leftarrow \mathsf{TRLWE.KeyGen}(N)$: On input N, output a secret key $\mathbf{sk} \xleftarrow{U} \mathbb{B}_N[X]$.
- $\mathbf{C} \leftarrow \mathsf{TRLWE.Enc}(\mathbf{sk}, \mathbf{m})$: On input $\mathbf{sk}$ and a plaintext polynomial $\mathbf{m} \in \mathbb{T}_N[X]$, sample $\mathbf{A} \xleftarrow{U} \mathbb{T}_N[X]$, an error vector $\mathbf{e} \leftarrow \chi$, compute $\mathbf{B} = \mathbf{A} \cdot \mathbf{sk} + \mathbf{e} + \mathbf{m}$, and return a ciphertext $\mathbf{C} = (\mathbf{A}, \mathbf{B}) \in \mathbb{T}_N[X]^2$.
- $\mathbf{m} \leftarrow \mathsf{TRLWE.Dec}(\mathbf{sk}, \mathbf{C})$: On input $\mathbf{sk}$ and $\mathbf{C} = (\mathbf{A}, \mathbf{B})$, compute and return $\mathbf{m} = \lfloor \mathbf{B} - \mathbf{A} \cdot \mathbf{sk} \rceil$.

The TRGSW encryption scheme provides a structure that is pivotal for efficient bootstrapping and homomorphic multiplication:

- $(N, l, h, \chi) \leftarrow \mathsf{TRGSW.Setup}(\lambda)$: On input λ, output parameters N, a matrix dimension l, a gadget matrix h, and an error distribution χ.
- $\mathbf{sk} \leftarrow \mathsf{TRGSW.KeyGen}(N)$: On input N, output a secret key $\mathbf{sk} \xleftarrow{U} \mathbb{B}_N[X]$.
- $\mathbf{C} \leftarrow \mathsf{TRGSW.Enc}(\mathbf{sk}, \mathbf{m})$: On input $\mathbf{sk}$ and a plaintext $\mathbf{m} \in \mathbb{T}_N[X]$, compute a TRLWE ciphertext $\mathbf{Z}$ of length $2l$, sample an error $\mathbf{e} \leftarrow \chi$, and return a ciphertext $\mathbf{C} = \mathbf{Z} + \mathbf{m} \cdot h$. (Here, h is a gadget vector/matrix used for decomposition.)
- $\mathbf{m} \leftarrow \mathsf{TRGSW.Dec}(\mathbf{sk}, \mathbf{C})$: On input $\mathbf{sk}$ and $\mathbf{C}$, computes $\mathbf{m} = h^{-1} \cdot (\mathbf{C} - \mathbf{Z})$, return the plaintext $\mathbf{m}$. (This requires knowledge of the structure of $\mathbf{Z}$ or the use of a specific gadget decomposition function h^{-1}.)

2.3. Torus-Based Fully Homomorphic Encryption

This section details the core encryption schemes within the TFHE family, outlining the evolution from the fundamental bit-wise encryption to the more advanced integer-wise paradigm, which is crucial for efficient arithmetic operations on encrypted data.

2.3.1. Bit-Wise Fully Homomorphic Encryption

The TFHE algorithm proposed by Chillotti et al. in 2016 [6] only supports bit-type data. This paper refers to it as the bit-wise TFHE algorithm. Its plaintext space is $\mathbb{T}$ and its ciphertext space is $(\mathbf{a}, b) \in \mathbb{T}^{n+1}$. Bit-wise TFHE mainly includes the following algorithms:

- $\mathsf{param} \leftarrow \mathsf{TFHE.Setup}(\lambda)$: On input the security parameter λ, return the parameter set param.
- $(\mathbf{sk}, \mathbf{bk}, \mathbf{ks}) \leftarrow \mathsf{TFHE.KeyGen}(\mathsf{param})$: On input the parameter set param, return the secret key $\mathbf{sk}$, the bootstrapping key $\mathbf{bk}$, and the key-switching key $\mathbf{ks}$.
- $\mathbf{c} \leftarrow \mathsf{TFHE.BitEnc}(\mathbf{sk}, m)$: On input the secret key $\mathbf{sk}$ and plaintext $m \in \mathbb{T}$ (representing a bit), return $\mathbf{c} = \mathsf{TLWE.Enc}(\mathbf{sk}, m/2)$.
- $m \leftarrow \mathsf{TFHE.BitDec}(\mathbf{sk}, \mathbf{c})$: On input the secret key $\mathbf{sk}$ and TLWE ciphertext $\mathbf{c} = (\mathbf{a}, b)$, obtain $2e + m$ via $2(b - \mathbf{a} \cdot \mathbf{sk})$, then decrypt to get the corresponding plaintext m by rounding up or down, i.e., $m = \lfloor 2(b - \mathbf{a} \cdot \mathbf{sk}) \rceil$, return plaintext m.

- $(\mathbf{c}' \leftarrow \mathsf{TFHE.KeySwitching}\ (\mathbf{ks}, \mathbf{c}))$: Transforms a ciphertext $\mathbf{c}$ under one key into a ciphertext $\mathbf{c}'$ under another key, preserving the encrypted message.
- $\mathbf{c} \leftarrow \mathsf{TFHE.Extract}\ (\mathbf{C})$: Input a TRLWE ciphertext $\mathbf{C} = (\mathbf{A}, \mathbf{B})$, extract the constant term coefficient, obtain a TLWE ciphertext $\mathbf{c} = (\mathbf{a}, b)$, return TLWE ciphertext $\mathbf{c}$.
- $(\mathbf{c}' \leftarrow \mathsf{TFHE.GateBoots}\ (m_1, \mathbf{c}, \mathbf{bk}, \mathbf{ks}))$: The core innovation of TFHE. Take a noisy ciphertext $\mathbf{c}$ encrypting m_{in} and a constant m_1, and return a fresh low-noise ciphertext $\mathbf{c}'$ encrypting $m_{in} - m_1$. This enables homomorphic evaluation of gates (e.g., NAND) while simultaneously reducing noise.

The TFHE algorithm is correct when the error e satisfies

$$|e| < \frac{1}{4}$$

because $2(b - \mathbf{a} \cdot \mathbf{sk}) = 2e + m$, so $|2e| < \frac{1}{2}$, and $2(b - \mathbf{a} \cdot \mathbf{sk}) = m$.

2.3.2. Integer-Wise Fully Homomorphic Encryption

While powerful for binary circuits, the bit-wise paradigm is inefficient for arithmetic operations on integers, requiring complex circuit constructions. To address this, Okada et al. [9] proposed an integer-wise variant of TFHE. This scheme expands the plaintext space from $\mathbb{T} \approx [0, 1)$ to a signed integer domain $\{-B, \ldots, B-1\}$ for a positive integer B, and generalizes the bootstrapping algorithm to natively handle integers, a process termed *integer bootstrapping*.

The integer-wise TFHE scheme shares the same $\mathsf{TFHE.Setup}$, $\mathsf{TFHE.KeyGen}$, $\mathsf{TFHE.KeySwitching}$, and $\mathsf{TFHE.Extract}$ algorithms as its bit-wise counterpart. The key differences lie in encryption, decryption, and bootstrapping:

- $(\mathbf{c} \leftarrow \mathsf{TFHE.IntEnc}(\mathbf{sk}, m))$: Encrypt an integer $m \in \{-B, \ldots, B-1\}$ by encoding it as $m/(2B) \in \mathbb{T}$. The ciphertext is $\mathbf{c} = (\mathbf{a}, b)$, where $b = \mathbf{a} \cdot \mathbf{sk} + \frac{m}{2B} + e$.
- $(m \leftarrow \mathsf{TFHE.IntDec}(\mathbf{sk}, \mathbf{c}))$: Decrypt by computing $2B(b - \mathbf{a} \cdot \mathbf{sk}) \approx m$ and rounding. Correctness requires a stricter noise constraint: $|e| < \frac{1}{4B}$.
- $(\mathbf{c}' \leftarrow \mathsf{IntBoots}\ (\mathbf{c}, f, \mathbf{bk}, \mathbf{ks}))$: A generalized bootstrapping procedure. Input a noisy ciphertext $\mathbf{c}$ encrypting m_{in} and a function $f : \{-B, \ldots, B-1\} \to \{-B, \ldots, B-1\}$. Output a fresh ciphertext $\mathbf{c}'$ encrypting $f(m_{in})$. It outputs $f(m_{in}) \leftarrow f(m_{in})$ for $m_{in} \in \{0, \cdots, B-1\}$ and $f(m_{in}) \leftarrow -f(m_{in} + B)$ for $m_{in} \in \{-B, \cdots, -1\}$.

  The integer-wise TFHE algorithm is correct when the error e satisfies

$$|e| < \frac{1}{4B} \tag{1}$$

because $2B(b - \mathbf{a} \cdot \mathbf{sk}) = 2Be + m_{in}$, $(|2Be| < \frac{1}{2})$, so $2B(b - \mathbf{a} \cdot \mathbf{sk}) = m_{in}$.

As the core component of the integer-wise TFHE scheme, the integer bootstrapping algorithm $\mathsf{IntBoots}$ operates through the following six steps:

1. Rounding: Scale and round the coefficients of the input ciphertext $(\mathbf{a}, b)$ to integers modulo $2N$, obtaining $(\bar{\mathbf{a}}, \bar{b})$. Here, a TLWE ciphertext $(\bar{\mathbf{a}}, \bar{b}) \in \mathbb{Z}_{2N}^n \times \mathbb{Z}_{2N}$ is obtained, satisfying:

$$\bar{b} - \bar{\mathbf{a}} \cdot \mathbf{sk} = \lfloor 2Nb \rceil - \sum_{i=1}^{n} \lfloor 2Na_i \rceil s_i = 2N\left(e + \frac{m_{\text{in}}}{2B}\right) + e_{\text{ACC}} \tag{2}$$

where e_{ACC} is the sum of rounding errors, and each rounding error is uniformly distributed in $(-\frac{1}{2}, \frac{1}{2})$. Therefore, $e_{\text{ACC}} = 0$ on average and can be neglected.

2. Test Polynomial Initialization: Initialize a polynomial testv $\in \mathbb{T}_N[X]$ whose coefficients encode the values of the function f to be evaluated. In this step the algorithm initializes the test polynomial testv, whose coefficients satisfy:

$$\mu_{i\cdot\frac{N}{B}-\frac{N}{2B}} = \cdots = \mu_{i\cdot\frac{N}{B}+\frac{N}{2B}} = \frac{f(i)}{2B} \quad \text{for} \quad i \in \{0, \cdots, B-1\} \tag{3}$$

(Note: Interpretation of coefficient indices adjusted for clarity). So the N coefficients of the test polynomial testv are uniformly filled with the B values $\frac{f(0)}{2B}$ to $\frac{f(B-1)}{2B}$.

3. Blind Rotation: Multiply (rotate) the test polynomial by $X^{\bar{b}}$ in the encrypted domain. In this step, a TRLWE ciphertext, $(0, \text{testv})$, is multiplied by $X^{\bar{b}}$, which is equivalent to shifting the coefficients of the polynomial. Therefore, **ACC** is a TRLWE ciphertext of the plaintext polynomial $\boldsymbol{\mu}$, whose constant term is $\mu_{\bar{b}} = \frac{f(\bar{b})}{2B}$. (Note: Interpretation adjusted).

4. Key-Switched Rotation: Perform a series of controlled rotations using the bootstrapping key **bk**, effectively evaluating the polynomial at the position $\bar{b} - \bar{\mathbf{a}} \cdot \mathbf{sk}$. In this step, the external product operation $\boxdot$ denotes the following mapping:

$$\boxdot : \mathsf{TRGSW} \times \mathsf{TRLWE} \to \mathsf{TRLWE}$$

Briefly, the external product $\boxdot$ multiplies a TRGSW ciphertext encrypting m_1 with a TRLWE ciphertext encrypting $\mathbf{m}_2$, outputting a TRLWE ciphertext encrypting $m_1 \times \mathbf{m}_2$. Therefore, after the loop, **ACC** is a TRLWE ciphertext of a plaintext polynomial whose constant term is $\mu_{\bar{b}-\bar{\mathbf{a}}\cdot\mathbf{sk}}$. According to Equation (2), $\bar{b} - \bar{\mathbf{a}} \cdot \mathbf{sk} = 2Ne + m_{\text{in}} \cdot \frac{N}{B}$. According to Equation (1), we have

$$m_{\text{in}} \cdot \frac{N}{B} - \frac{N}{2B} < 2Ne + m_{\text{in}} \cdot \frac{N}{B} < m_{\text{in}} \cdot \frac{N}{B} + \frac{N}{2B}$$

Therefore, $\mu_{\bar{b}-\bar{\mathbf{a}}\cdot\mathbf{sk}}$ lies between $m_{\text{in}} \cdot \frac{N}{B} - \frac{N}{2B}$ and $m_{\text{in}} \cdot \frac{N}{B} + \frac{N}{2B}$. According to Equation (3), we get

$$\mu_{\bar{b}-\bar{\mathbf{a}}} \cdot \mathbf{sk} = \frac{f(m_{\text{in}})}{2B}$$

Therefore **ACC** is a TRLWE ciphertext of the plaintext polynomial $\boldsymbol{\mu}$, whose constant term is $\frac{f(m_{\text{in}})}{2B}$.

According to Theorem 6.2 in [17], we get:

$$\|\mathsf{Err}(\mathbf{ACC})\|_\infty \leq 2nlN\beta\alpha_{\mathbf{bk}} + n(1+N)\epsilon \tag{4}$$

where $\beta = \frac{B_g}{2}$, $\epsilon = \frac{1}{2B_g}$ is the precision parameter of the gadget decomposition matrix, $l \in \mathbb{N}$, $B_g \in \mathbb{N}$, $\alpha_{\mathbf{bk}}$ is the noise parameter of the bootstrapping key **bk**.

5. Extraction: Extract the constant term of the resulting polynomial into a TLWE ciphertext. In this step, the TFHE.Extract algorithm extracts a TLWE ciphertext $\mathbf{u}$ from the TRLWE ciphertext **ACC**. Specifically, **ACC** is a TRLWE ciphertext of plaintext polynomial $\boldsymbol{\mu}$ under secret key $\mathbf{sk}'$, and the constant term of $\boldsymbol{\mu}$ is $\frac{f(m_{\text{in}})}{2B}$. $\mathbf{u}$ is a TLWE ciphertext encrypting $\frac{f(m_{\text{in}})}{2B}$ under some secret key $\mathbf{sk}''$. The $TFHEExtract$ algorithm does not add extra noise, so $\|\mathsf{Err}(\mathbf{u})\|_\infty \leq \|\mathsf{Err}(\mathbf{ACC})\|_\infty$.

6. Key Switching: Finalize the output ciphertext $\mathbf{c}'$. The result is a ciphertext $\mathbf{c}'$ encrypting $f(m_{\text{in}})$ with manageable noise, whose variance is bounded as shown in Equation (6). In this step, by calling the TFHE.KeySwitching algorithm, we obtain a TLWE ciphertext $(\mathbf{a}', b')$ encrypting $\frac{m_{\text{out}}}{2B} = \frac{f(m_{\text{in}})}{2B} \in \mathbb{T}$. According to Theorem 6.3 in the TFHE algorithm [17], the worst-case infinity norm of the noise is $\|\mathsf{Err}((\mathbf{a}', b'))\|_\infty \leq \|\mathsf{Err}(\mathbf{ACC})\|_\infty + nt\gamma + n2^{-(t+1)}$. According to Equation (4), the worst-case infinity norm of the noise is:

$$\|\mathsf{Err}((\mathbf{a}', b'))\|_\infty \leq 2nlN\beta\alpha_{\mathbf{bk}} + n(1+N)\epsilon + nt\gamma + n2^{-(t+1)} \tag{5}$$

At this point, according to Theorem 6.3 in [17], the variance of the noise in the average case is

$$\mathsf{Var}(\mathsf{Err}((\mathbf{a}', b'))) \leq 2nlN\beta^2\alpha_{\mathbf{bk}}^2 + n(1+N)\epsilon^2 + nt\gamma^2 + n2^{-2(t+1)} \tag{6}$$

The output ciphertext $\mathbf{c}' = (\mathbf{a}', b')$ is a small-noise TLWE ciphertext of $f(m_{\text{in}})$.

2.3.3. Radix-Based Large Integer Decomposition

Since the integer-wise TFHE algorithm can only support integers with a maximum precision of 6 bits, whereas most computing tasks often require computation on integers with 10 to 12 bits of precision, this section studies large integer ciphertext decomposition algorithms. Currently, for decomposing large integers into small integers in plaintext, there are mainly two types of algorithms: decomposition based on radix [18] and decomposition based on the Chinese Remainder Theorem (CRT) [10]. Considering that operations on ciphertexts are more complex than on plaintexts, this section studies the more concise large integer ciphertext decomposition algorithm based on radix.

Currently, the design of a radix-based large integer ciphertext decomposition algorithm using the TFHE algorithm mainly follows the idea of the large integer ciphertext decomposition algorithm proposed by Liu et al. [18].

It provides a foundation for handling large integers but suffers from two critical limitations:

- The decomposed small integer ciphertexts lose their sign bits, resulting in unsigned numbers that cannot support cross-domain functions.
- The integer bootstrapping in [18] can only handle single-bit carries, while homomorphic modulo operations may generate multi-bit carries, leading to decomposition errors.

To address these limitations, this paper proposes an enhanced signed decomposition approach, as detailed in the following sections.

3. Signed Large Integer Ciphertext Decomposition Algorithm

3.1. Signed Large-Integer Decomposition for ReLU

Current homomorphic digit decomposition algorithms based on the sign-evaluation algorithm for larger plaintext spaces have the problems of being unable to retain the sign bit and ciphertexts after modulo operations as they were unable to remove carry interference, so they cannot be used to implement cross-domain ReLU functions supporting large integers. Therefore, this paper extends the current radix-based large integer ciphertext decomposition algorithm. First, the left-shift algorithm LeftShift, and the homomorphic modulo algorithm HomMod are shown in Algorithms 1 and 2, respectively. Second, to address the interference of low-bit information in high-bit ciphertexts, the ciphertext cleaning algorithm CipherClean is proposed and exhibited in Algorithm 3. Finally, based on this, the signed large integer ciphertext decomposition algorithm SignedDecomp is proposed and shown in Algorithm 4, solving the above problems and laying the foundation for implementing the homomorphic ReLU algorithm supporting full-domain bootstrapping for large integers. Here, we give a definition of the ReLU function: given the integer interval $\{-B, \cdots, B-1\}$,

$$ReLU(x) = \begin{cases} x & x \in \{0, \cdots, B-1\} \\ 0 & x \in \{-B, \cdots, -1\} \end{cases} \tag{7}$$

Algorithm 1 LeftShift(C_x, num_{shift}).

Require: Ciphertext $C_x = (\mathbf{a}, b) \in \mathbb{T}^{n+1}$, shift bits num_{shift}
Ensure: Ciphertext $C_{out} = (\mathbf{c}, d) \in \mathbb{T}^{n+1}$ where $out = x \ll num_{shift}$
1: **for** $i = 1$ to n **do**
2: $\quad c_i = a_i \ll num_{shift}$
3: **end for**
4: $d = b \ll num_{shift}$
5: **return** $C_{out} = (\mathbf{c}, d)$

Algorithm 2 HomMod(C_x, num_{mod}).

Require: Ciphertext $C_x = (\mathbf{a}, b)$, modulus bits num_{mod}
Ensure: Ciphertext $C_{out} = (\mathbf{c}, d)$ where $out = x \mod 2^{num_{mod}}$ (signed with random sign bit)
1: $q = 2^{32-num_{mod}}$
2: **for** $i = 1$ to n **do**
3: $\quad c'_i = a_i \mod q$
4: **end for**
5: $d' = b \mod q$
6: $Cout = \text{LeftShift}((\mathbf{c}', d'), num_{mod} - 1)$
7: **return** $C_{out} = (\mathbf{c}, d)$

Algorithm 3 CipherClean$(C_x, C_{sign}, num, bk, ks)$.

Require: Ciphertext C_x, sign ciphertext C_{sign}, precision $num \in [1, 5]$, bootstrapping key bk, key-switching key ks
Ensure: Ciphertext C_{out} where $out = \frac{x}{2^{32-num}}$
1: $C_{out} = C_{sign}$
2: **for** $i = 1$ to num **do**
3: $\quad C_{x_i} = \text{LeftShift}(C_x, i)$
4: $\quad C_{imp_i} = \text{IntBoots}(C_{x_i}, f_{sign3}, bk, ks)$
5: $\quad C_{bin_i} = C_{sign} + (0, 2^{4-i} \times \frac{1}{2B})$
6: $\quad C_{out} = C_{out} + C_{bin_i}$
7: **end for**
8: **return** C_{out}

Algorithm 4 SignedDecomp(C_x, bk, ks).

Require: Ciphertext C_x (precision Q), bootstrapping keys bk, key-switching keys ks
Ensure: Ciphertexts $C_{x_1}, \cdots, C_{x_n}$ where $x_i \in \{-B, \cdots, B-1\}$, $x = x_1 + \sum_{i=2}^{n} x_i \cdot 32^{i-1}$, $n = \lceil \frac{Q-1}{5} \rceil$
1: **if** $Q \geq 11$ **then**
2: $\quad C_{sign4} = \text{IntBoots}(C_x, f_{sign4}, bk, ks)$
3: $\quad C_{sign5} = \text{IntBoots}(C_x, f_{sign5}, bk, ks)$
4: **else**
5: $\quad$ PBSManyLUT$(C_{sign4}, C_{sign5}, C_x, f_{sign4}, f_{sign5}, bk, ks)$
6: **end if**
7: $C_{bin1} = C_{sign4} - (0, -\frac{1}{2} \times \frac{1}{2B})$
8: $C_{bin} = \text{LeftShift}(C_{bin1}, 5n + 1 - Q)$
9: $C_{bin8} = C_{sign5} - (0, -4 \times \frac{1}{2B})$
10: $C'_{bin} = \text{LeftShift}(C_{bin8}, 5n - 2 - Q)$
11: $C_{bin32} = \text{LeftShift}(C_{bin8}, 2)$

https://doi.org/10.3390/cryptography10020021

Algorithm 4 *Cont.*

for $i = 1$ to $n - 1$ **do**
  $C_{x_i} = \mathsf{HomMod}(C_x, Q - 5i)$
  if $i > 1$ **then**
    $C_{x_i} = C_{x_i} - C_{bin1}$
  end if
  $C_{bin} = \mathsf{IntBoots}(C_{x_i}, f_{sign6}, bk, ks) - (0, 16 \times \frac{1}{2B})$
  $C_{x_i} = C_{x_i} + C_{bin32} + C_{bin}$
end for
if $5n + 1 - Q > 2$ **then**
  $C_{x_n} = C_x - C'_{bin}$
else
  $C_{x_n} = C_x - C_{bin}$
end if
for $j = 2$ to $n - 1$ **do**
  $C_{x_j} = \mathsf{CipherClean}(C_{x_j}, C_{bin32}, 5, bk, ks)$
end for
$C_{x_n} = \mathsf{CipherClean}(C_{x_n}, C_{bin32}, Q - 5n + 4, bk, ks)$
return $C_{x_1}, \cdots, C_{x_n}$

3.2. The Left-Shift and Homomorphic Modulo Algorithms

The homomorphic modulo algorithm HomMod can perform homomorphic modulo operations on the input ciphertext and output the modulo ciphertext. It is worth noting that the output of the modulo operation is a signed number with a random sign bit. Specifically, the modulo operation is achieved by retaining the low-bit data and discarding the high-bit data. In the HomMod algorithm, the modulo operation is specifically reflected in lines 2 and 4 of the algorithm. As known from Section 2, for an LWE ciphertext $(\mathbf{a}, b)$, $m + e = b - \mathbf{a} \cdot \mathbf{sk}$, and since $\mathbf{sk}$ is a vector composed of 0 or 1, it satisfies:

$$(m + e) \mod q = (b \mod q - \mathbf{a} \mod q \cdot \mathbf{sk}) \tag{8}$$

In the homomorphic modulo algorithm HomMod, lines 2 and 4 of the algorithm perform modulo operations on $\mathbf{a}$ and b, respectively. However, to ensure the correctness of the modulo operation in subtraction, the entire operation result needs to be modulo to remove high-bit carry information. The HomMod algorithm removes high-bit carry information by left-shifting the modulo ciphertext, but the left-shift bit count is $num_{mod} - 1$ because the highest bit needs to be reserved as the sign bit. However, since the modulo operation discards the high-bit sign bit information, the highest bit at this point is a random value and does not represent the true sign of the modulo ciphertext. Therefore, the output of the homomorphic modulo algorithm HomMod is the ciphertext of a signed number with a random sign bit. A schematic diagram of the HomMod algorithm flow is shown in Figure 1.

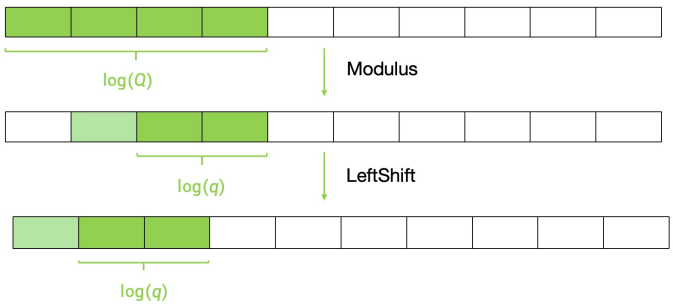


Figure 1. HomMod algorithm flow.

3.3. The Ciphertext Cleaning Algorithm

The ciphertext cleaning algorithm CipherClean can clean the input ciphertext C_x based on the effective bit precision num, retaining the high $num+1$ bits of information and setting the remaining low-bit information to 0. This process is equivalent to removing the low-bit noise of the ciphertext, ensuring that the ciphertext can perform subsequent homomorphic operations. Specifically, the algorithm CipherClean performs ciphertext cleaning bit by bit based on the effective bit precision. First, the current effective bit information is moved to the highest bit via the left-shift algorithm LeftShift. Then, sign judgment is performed via the integer bootstrapping algorithm IntBoots, obtaining the ciphertext C_{tmp_i} of -2^{4-j} or 2^{4-j}. Next, add the ciphertext C_{tmp_i} to the ciphertext of the constant 2^{4-j}, obtaining the ciphertext C_{bin_i} where bin_i is 0 or 2^{5-i}. At this point, bin_i is consistent with the highest $i+1$ bits of the input ciphertext. Taking $i=1$ as an example, if it is negative after left-shifting by 1 bit, this indicates that the second highest bit of the plaintext x corresponding to the input ciphertext is 1. After bootstrapping and addition, the obtained bin_i is 16. With a precision of 6 bits, 16 represents the second highest bit as 1. If it is positive after left-shifting by 1 bit, this indicates that the second highest bit of the plaintext x corresponding to the input ciphertext is 0. Then, after bootstrapping and addition, the obtained bin_i is 0. With a precision of 6 bits, 0 indicates that the second highest bit is 0. Finally, adding the input sign bit ciphertext to the obtained multiple ciphertexts C_{bin_i} yields the ciphertext C_{out} consistent with the highest $num+1$ bits of the plaintext x corresponding to the input ciphertext C_x, with all other bits set to 0, satisfying $out = \frac{x}{2^3 2^{-num_{out}}}$, achieving ciphertext cleaning.

In the ciphertext cleaning algorithm CipherClean, the definition of function f_{sign3} is as follows:

$$f_{sign3}(x,j) = \begin{cases} -2^{4-j} & x \geq 0 \\ 2^{4-j} & x < 0 \end{cases} \tag{9}$$

As described in Section 2.3.2, the coefficients of the polynomial testv in the integer bootstrapping algorithm IntBoots are related to the passed function ff. For function f_{sign3}, the coefficient set $\{\mu_0, \cdots, \mu_{N-1}\}$ of the test polynomial testv is defined as follows:

$$\begin{cases} \mu_0, \cdots, \mu_{\lfloor \frac{N}{2B} \rfloor - 1} := \frac{f_{sign3}(0,j)}{2B} = -\frac{2^{4-j}}{2B} \\ \mu_{\lfloor \frac{(2i-1)N}{2B} \rfloor}, \cdots, \mu_{\lfloor \frac{(2i+1)N}{2B} \rfloor - 1} \\ := \frac{f_{sign3}(i,j)}{2B} = -\frac{2^{4-j}}{2B}, i \in 1, \cdots, B-1 \\ \mu_{N-\lfloor \frac{N}{2B} \rfloor}, \cdots, \mu_{N-1} := -\frac{f_{sign3}(0,j)}{2B} = \frac{2^{4-j}}{2B} \end{cases} \tag{10}$$

3.4. The Signed Large Integer Ciphertext Decomposition Algorithm

This subsection presents the core SignedDecomp algorithm for decomposing large integer ciphertexts into signed 6-bit segments. We begin by defining the sign evaluation functions and their corresponding test polynomial configurations, then provide a comprehensive explanation of the algorithmic workflow, and conclude with a detailed step-by-step example illustrating the decomposition process for various precision levels.

3.4.1. Sign Evaluation Functions

The signed large integer ciphertext decomposition algorithm SignedDecomp calls the integer bootstrapping algorithm IntBoots (algorithm definition see Section 2.3.2). The decomposed small integer precision is 6, and B is 32. The functions f_{sign4}, f_{sign5}, and f_{sign6} are defined as follows:

$$f_{sign4}(x) = \begin{cases} \frac{1}{2} & x \geq 0 \\ -\frac{1}{2} & x < 0 \end{cases} \tag{11}$$

$$f_{sign5}(x) = \begin{cases} 4 & x \geq 0 \\ -4 & x < 0 \end{cases} \tag{12}$$

$$f_{sign6}(x) = \begin{cases} 16 & x \geq 0 \\ -16 & x < 0 \end{cases} \tag{13}$$

As described in Section 2.3.2, in the integer bootstrapping algorithm IntBoots, the coefficients of the test polynomial testv are related to the input function f. For the function f_{sign4}, the coefficient set $\{\mu_0, \cdots, \mu_{N-1}\}$ of the test polynomial testv is defined as follows:

$$\begin{cases} \mu_0, \cdots, \mu_{\lfloor \frac{N}{2B} \rfloor - 1} := \frac{f_{sign4}(0)}{2B} = \frac{1}{4B} \\ \mu_{\lfloor \frac{(2i-1)N}{2B} \rfloor}, \cdots, \mu_{\lfloor \frac{(2i+1)N}{2B} \rfloor - 1} \\ := \frac{f_{sign4}(i)}{2B} = \frac{1}{4B}, i \in \{1, \ldots, B-1\} \\ \mu_{N - \lfloor \frac{N}{2B} \rfloor}, \cdots, \mu_{N-1} := -\frac{f_{sign4}(0)}{2B} = -\frac{1}{4B} \end{cases} \tag{14}$$

For the function f_{sign5}, the coefficient set $\{\mu_0, \cdots, \mu_{N-1}\}$ of the test polynomial testv is defined as follows:

$$\begin{cases} \mu_0, \cdots, \mu_{\lfloor \frac{N}{2B} \rfloor - 1} := \frac{f_{sign5}(0)}{2B} = \frac{2}{B} \\ \mu_{\lfloor \frac{(2i-1)N}{2B} \rfloor}, \cdots, \mu_{\lfloor \frac{(2i+1)N}{2B} \rfloor - 1} := \frac{f_{sign5}(i)}{2B} \\ = \frac{2}{B}, i \in \{1, \ldots, B-1\} \\ \mu_{N - \lfloor \frac{N}{2B} \rfloor}, \cdots, \mu_{N-1} := -\frac{f_{sign5}(0)}{2B} = -\frac{2}{B} \end{cases} \tag{15}$$

For the function f_{sign6}, the coefficient set $\{\mu_0, \cdots, \mu_{N-1}\}$ of the test polynomial testv is defined as follows:

$$\begin{cases} \mu_0, \cdots, \mu_{\lfloor \frac{N}{2B} \rfloor - 1} := \frac{f_{sign6}(0)}{2B} = \frac{8}{B} \\ \mu_{\lfloor \frac{(2i-1)N}{2B} \rfloor}, \cdots, \mu_{\lfloor \frac{(2i+1)N}{2B} \rfloor - 1} \\ := \frac{f_{sign6}(i)}{2B} = \frac{8}{B}, i \in \{1, \ldots, B-1\} \\ \mu_{N - \lfloor \frac{N}{2B} \rfloor}, \cdots, \mu_{N-1} := -\frac{f_{sign6}(0)}{2B} = -\frac{8}{B} \end{cases} \tag{16}$$

3.4.2. Algorithmic Workflow and Explanation

The SignedDecomp algorithm (Algorithm 4) orchestrates a sophisticated decomposition process through the coordinated execution of multiple homomorphic primitives. The algorithm commences with sign determination, where the precision Q dictates the bootstrapping strategy (lines 2–6). For $Q \geq 11$, two sequential IntBoots operations evaluate f_{sign4} and f_{sign5} functions, while for $Q < 11$, a single PBSManyLUT operation efficiently computes both functions simultaneously. These bootstrapping operations internally configure the test polynomial testv coefficients according to Equations (14) and (15), enabling precise sign extraction through torus-based computation.

Following sign determination, the algorithm constructs precision-adaptive offset ciphertexts through a sequence of arithmetic operations and LeftShift transformations (lines 7–11). The LeftShift operations with dynamically calculated bit shifts ($5n + 1 - Q$, $5n - 2 - Q$, and 2) generate the necessary offset values C_{bin}, C'_{bin}, and C_{bin32} that accommodate the variable precision requirements of large integer decomposition.

The core decomposition phase (lines 12–19) employs iterative HomMod operations to extract small integer segments from the original ciphertext. Each HomMod call with parameter $Q - 5i$ performs signed modular reduction on specific bit ranges, producing intermediate ciphertexts with random sign bits that require subsequent correction. For segments beyond the first ($i > 1$), the algorithm subtracts C_{bin1} to eliminate carry interference from lower-order bits. The algorithm then applies additional IntBoots evaluations of f_{sign6} (with testv coefficients defined in Equation (16)) to determine the true sign of each small integer segment, followed by arithmetic combination with the large integer's sign bit C_{bin32} to ensure consistent signed representation across all decomposed segments.

For the highest-order segment (lines 20–24), the algorithm performs precision-dependent subtraction using the previously generated offset ciphertexts. The conditional selection between C'_{bin} and C_{bin} based on $5n + 1 - Q > 2$ ensures appropriate handling of different precision regimes, effectively truncating the original ciphertext while accounting for sign-dependent adjustments.

The final refinement stage (lines 25–28) employs CipherClean operations to eliminate residual low-order noise from the decomposed ciphertexts. This critical step ensures that all segments maintain the required cleanliness for subsequent homomorphic operations, with precision parameters carefully calibrated to each segment's specific bit requirements (5 for middle-order segments, $Q - 5n + 4$ for the high-order segment).

Throughout this process, the algorithm maintains the fixed 6-bit precision standard for all small integer segments while dynamically adapting to the variable precision Q of the input large integer, demonstrating a robust approach to full-domain decomposition that efficiently handles both high-precision ($Q \geq 11$) and moderate-precision ($Q < 11$) scenarios through adaptive use of IntBoots and PBSManyLUT operations.

3.4.3. Parameter Adaptation Examples for Different Precisions

To clarify the parameter adaptation of the SignedDecomp algorithm, we provide concrete examples for decomposing integers of different precisions (7–11 bits). Table 2 summarizes the key algorithmic parameters for each precision level Q, including:

- Number of segments $n = \lceil Q/5 \rceil$;
- Dynamic shift amounts used in Algorithm 4: $5n + 1 - Q$ and $5n - 2 - Q$;
- The effective data bits (`num`) for each segment, which determine the precision parameter in the subsequent CipherClean operation.

Note that the effective data bits `num` for a segment represent the number of magnitude bits (excluding the sign bit) that carry meaningful information. The total bits per segment (including sign bit) is `num`+1, though all segments are stored as 6-bit signed integers in our algorithm. The CipherClean operation retains exactly these `num` high-order magnitude bits while zeroing out lower bits.

Table 2. Parameter examples for SignedDecomp under different input precisions Q.

Q	n	Shift1 $(5n+1-Q)$	Shift2 $(5n-2-Q)$	Seg1 (num)	Seg2 (num)	Seg3 (num)
7	2	4	1	5	1	–
8	2	3	0	5	2	–
9	2	2	−1	5	3	–
10	2	1	−2	5	4	–
11	3	5	2	5	5	0

Interpretation: For $Q = 11$ (3 segments), the highest-order segment has 0 bits of effective data (i.e., contains only the sign), which is handled by the precision-consistent bootstrapping mechanism described in Section 4.2. The negative shift values for $Q = 9$ and $Q = 10$ indicate that the corresponding offset ciphertexts (C'_{bin}) are not required, and the algorithm conditionally skips those operations (see Algorithm 4, lines 20–24). The `num` values for each segment are precisely the parameters used in the CipherClean step (lines 25–28 of Algorithm 4) to eliminate residual low-order noise.

For $Q = 11$ (3 segments), the highest-order segment has 0 bits of useful data (i.e., only sign), which is handled by the precision-consistent bootstrapping described in Section 4.2. The negative shift values for $Q = 9, 10$ indicate that the corresponding offset ciphertexts are not required, and the algorithm branches accordingly (see Algorithm 4, lines 20–24).

3.4.4. Detailed Step-by-Step Example

The workflow of the SignedDecomp algorithm is illustrated in Figure 2 for a large integer precision of 11 bits. The rounded rectangle encloses the plaintext x encrypted with key $\vec{s}$, where the sub-rectangle at the bottom-right corner indicates the encryption key $\vec{s}$ used. We use a concrete example of decomposing a large integer (with precision between 12 and 16 bits) into 3 small integer ciphertexts.

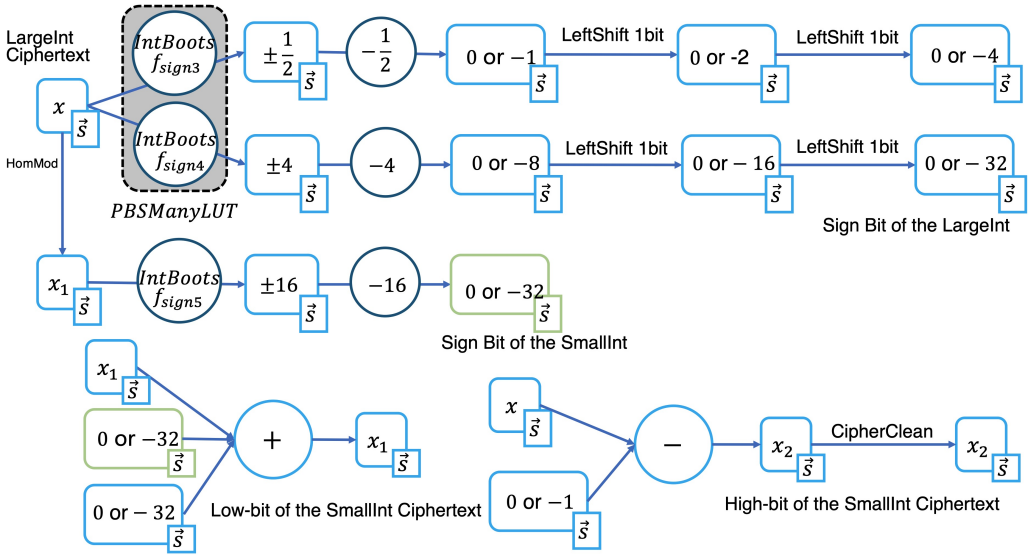


Figure 2. Flowchart of the signed large integer ciphertext decomposition algorithm for a large integer precision of 11 bits.

Obtaining Sign and Offset Ciphertexts (Lines 1–11)

The initial phase computes the large integer's sign bit and the necessary offset ciphertexts. The algorithm homomorphically evaluates functions f_{sign4} and f_{sign5} via IntBoots, yielding ciphertexts C_{sign4} ($\frac{1}{2}$ or $-\frac{1}{2}$) and C_{sign5} (4 or -4).

A key detail is the choice between bootstrapping methods: for precision $\geq$11 bits, two separate IntBoots calls are needed, whereas for precision $\leq$10 bits, a single PBSManyLUT [15] call suffices. This is due to the slot constraints in the test polynomial, where $N = 1024$ allows representing at most 512 positive and 512 negative values, thus limiting PBSManyLUT to 10-bit precision.

Subsequent lines derive the specific offset ciphertexts:

- Line 7: Subtracting C_{sign4} from a constant gives C_{bin1} (0 or -1), the offset for 16-bit precision.
- Line 8: Left-shifting C_{bin1} by $5n + 1 - Q$ bits produces C_{bin}, the offset for precisions 14–16 ($0/-4$, $0/-2$, $0/-1$ respectively).
- Line 9: Subtracting C_{sign5} from a constant gives C_{bin8} (0 or -8), the offset for 13-bit precision.
- Line 10: Left-shifting C_{bin8} by $5n - 2 - Q$ bits yields C'_{bin}, the offset for precisions 12–13 ($0/-16$, $0/-8$).
- Line 11: Left-shifting C_{bin8} by 2 bits produces C_{bin32} (0 or -32), which serves as the large integer's sign bit ciphertext.

Computing Low-order and Middle-order Ciphertexts (Lines 12–19)

This phase generates the ciphertexts C_{x1} and C_{x2}. Line 13 applies HomMod to obtain the initial small integer ciphertexts, which have random sign bits. For the middle-order ciphertext ($i > 1$), line 15 subtracts C_{bin1} to eliminate interference from the lower-order part.

The true sign bit of each small integer is determined in line 17 by evaluating f_{sign6} and subtracting a constant, resulting in a sign bit ciphertext C_{bin} (0 or -32). Finally, line 18 constructs the final signed small integer ciphertext by summing the modulo ciphertext, its own sign bit ciphertext (C_{bin}), and the large integer's sign bit ciphertext (C_{bin32}). This crucial step ensures the sign of the small integer matches that of the original large integer.

Acquiring the High-order Ciphertext (Lines 20–24)

The high-order small integer ciphertext C_{x_3} is obtained by truncating the original large integer ciphertext. The specific offset ciphertext subtracted depends on the precision: C'_{bin} is used for precisions 12–13, and C_{bin} for precisions 14–16. This step removes the influence of the lower-order segments.

Ciphertext Cleaning and Final Output (Lines 25–28)

The final phase ensures ciphertext cleanliness. The middle-order ciphertext C_{x_2} is cleaned with a precision parameter of 5. The high-order ciphertext C_{x_3} is cleaned with a precision of $Q - 5n + 4$, calculated based on its dynamic data precision. After this cleaning process, the algorithm returns the three decomposed small integer ciphertexts.

4. The Full-Domain Large Integer ReLU

4.1. Overview of the Proposed Method

Since current fully homomorphic encryption algorithms supporting full-domain bootstrapping are limited to small integers with precision up to 6 bits and thus unsuitable for most computational tasks, we propose the LargeIntReLU algorithm by integrating the small-integer homomorphic ReLU algorithm SmallIntReLU [13] with the signed large integer ciphertext decomposition algorithm SignedDecomp from Section 3. The proposed LargeIntReLU operates by decomposing large integer ciphertexts into multiple small integer ciphertexts using SignedDecomp and then performing homomorphic ReLU operations on each segment in parallel via SmallIntReLU, thereby enabling full-domain ReLU evaluation for large integers through decomposition and parallel processing.

4.2. The Defect of Directly Applying Small-Integer ReLU

While the LargeIntReLU algorithm employs a uniform 6-bit precision for all small integer ciphertexts with upper limit $B = 32$ to simplify design and ensure operational consistency, directly applying the SmallIntReLU algorithm to decomposed ciphertexts faces a fundamental challenge. As established in Section 3, the highest-order ciphertext C_{x_n} among the n decomposed segments exhibits variable precision of $Q - 5(n-1)$, ranging from 2 to 6 bits, creating a critical mismatch with the fixed 6-bit framework.

The core limitation emerges from this precision divergence, which causes non-uniform slot allocation in the torus due to inconsistent assignment of test polynomial testv coefficients, ultimately generating computational errors. Equation (3) illustrates the original coefficient assignment rule governing this process, while Figure 3 visually demonstrates the allocation disparities between 6-bit and 3-bit precision scenarios that lead to computational inaccuracies.

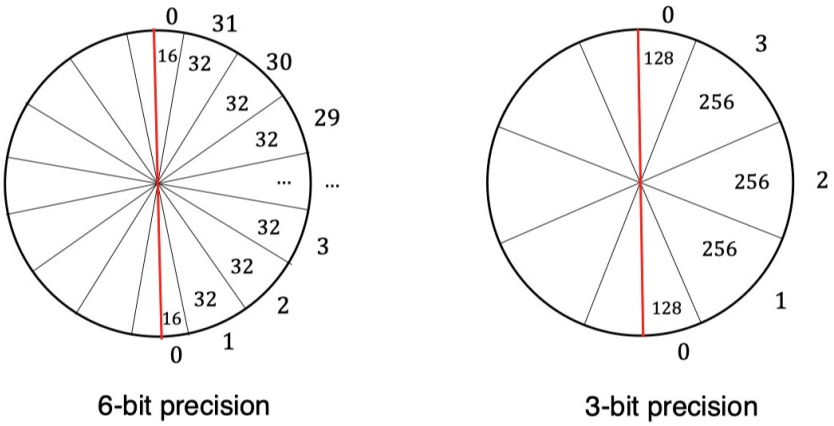


Figure 3. Torus slot allocation for 6-bit vs. 3-bit precision. (**left**) For 6-bit precision ($B = 32$), each integer occupies 32 slots. (**right**) For 3-bit precision ($B = 4$), each integer occupies 256 slots. This inconsistency causes computational errors when the same integer (e.g., 12) is represented differently across precisions.

Analysis reveals that under 6-bit precision, the torus slot range for the small integer 12 satisfies:

$$16 + 32 \times 11 + x = 128 + 240 + x \quad \text{where } 0 < x \leq 32 \tag{17}$$

Under 3-bit precision, the integer 12 truncates to 1, with its slot range satisfying:

$$128 + y = 128 + 240 + x \quad \text{where } 0 < y \leq 256, 0 < x \leq 32 \tag{18}$$

Comparing Equations (17) and (18) demonstrates that torus slot allocation remains consistent only when $0 < x \leq 16$; beyond this range, inconsistencies arise and produce computational errors.

To resolve this fundamental limitation, we modify the test polynomial coefficient assignment rule to:

$$\mu_{\lfloor \frac{(2i-1)N}{2B} \rfloor}, \cdots, \mu_{\lfloor \frac{(2i+1)N}{2B} \rfloor - 1} = \frac{f(i)}{2B}, \quad i \in \{0, \cdots, B-1\} \tag{19}$$

This modification ensures consistent testv coefficient assignment regardless of precision variations, as demonstrated by the unified torus slot allocation for 3-bit precision in Figure 4.

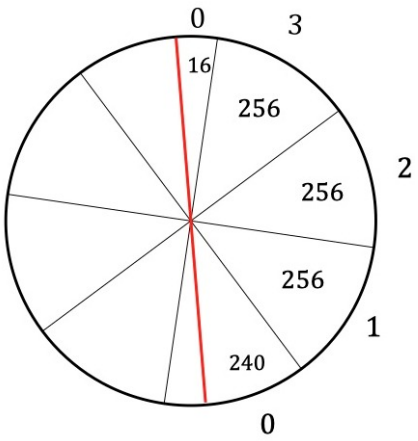


Figure 4. Modified torus slot allocation for 3-bit precision after applying Equation (19). The slot assignment is now uniform (32 slots per integer) and consistent with the 6-bit case, eliminating the computational errors shown in Figure 3.

4.3. The Large Integer ReLU Algorithm

Building upon the signed decomposition framework and the resolution of precision inconsistency in torus slot allocation, we now present the complete LargeIntReLU algorithm for homomorphic ReLU evaluation on large integers. The algorithm systematically integrates the SignedDecomp procedure with the optimized SmallIntReLU bootstrapping, enabling full-domain ReLU functionality while maintaining arithmetic consistency across variable precisions.

The flowchart of the homomorphic ReLU algorithm LargeIntReLU (Algorithm 5) supporting full-domain bootstrapping for large integers is shown in Figure 5. The rounded rectangle encloses the plaintext x encrypted with key $\vec{s}$, where the sub-rectangle at the bottom-right corner indicates the encryption key $\vec{s}$ used.

Algorithm 5 LargeIntReLU(C_x, bk, ks).

Require: Large integer ciphertext C_x (precision $Q > 6$), bootstrapping key bk, key-switching key ks

Ensure: Ciphertexts $C_{out_1}, \cdots, C_{out_n}$ where $\mathrm{ReLU}(x) = out_1 + \sum_{i=2}^{n} out_i \cdot 32^{i-1}$, $n = \lceil \frac{Q}{5} \rceil$

1: $C_{x_1}, \cdots, C_{x_n} = \mathsf{SignedDecomp}(C_x, bk, ks)$
2: **for** $i = 1$ to n **do**
3: $\quad C_{out_i} = \mathsf{SmallIntReLU}(C_{x_i}, bk, ks)$
4: **end for**
5: **return** $C_{out_1}, \cdots, C_{out_n}$

Specifically, first, the large integer ciphertext C_x is decomposed using the signed large integer ciphertext decomposition algorithm SignedDecomp to obtain small integer ciphertexts $C_{x_1}, \cdots, C_{x_n}$. Then, each of the n small integer ciphertexts undergoes the homomorphic ReLU operation using the homomorphic ReLU algorithm SmallIntReLU supporting full-domain bootstrapping for small integers, resulting in ciphertexts $C_{out_1}, \cdots, C_{out_n}$, satisfying $out_i = \mathrm{ReLU}(x_i)$. When the large integer x is negative, the decomposed small integers $x_1, \cdots, x_n$ are also negative. According to Equation (7), $out_1 = \cdots = out_n = 0$, and $\mathrm{ReLU}(x) = 0$, thus satisfying $\mathrm{ReLU}(x) = out_1 + \sum_{i=2}^{n} out_i \cdot 32^{i-1}$. When the large integer x is positive, the decomposed small integers $x_1, \cdots, x_n$ are also positive. According to Equation (7), $out_1 = x_1, \cdots, out_n = x_n$, and $\mathrm{ReLU}(x) = x$. From the SignedDecomp algorithm, $x = x_1 + \sum_{i=2}^{n} x_i \cdot 32^{i-1}$, thus satisfying $\mathrm{ReLU}(x) = out_1 + \sum_{i=2}^{n} out_i \cdot 32^{i-1}$.

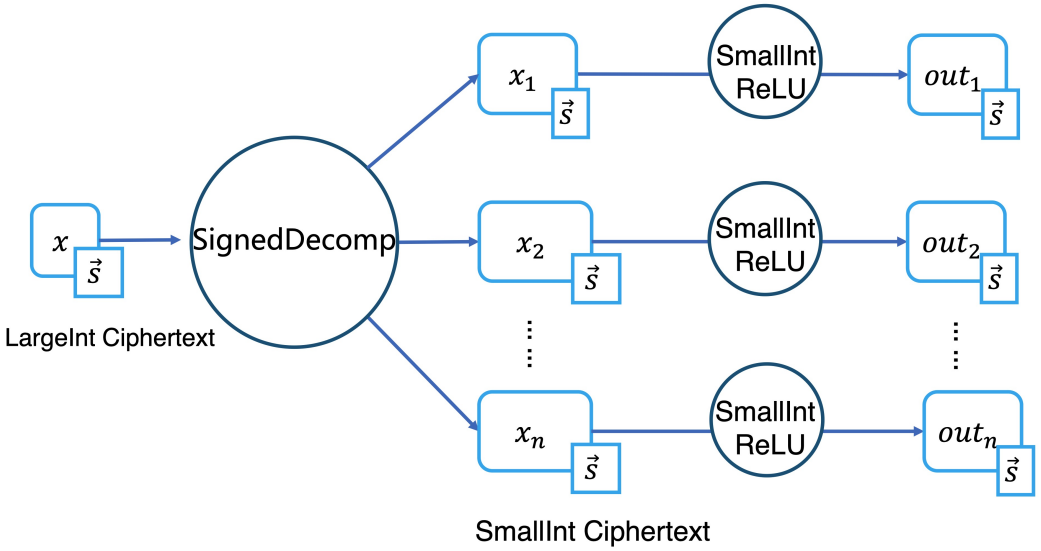


Figure 5. Flowchart of the homomorphic ReLU algorithm supporting full-domain bootstrapping for large integers.

It is noteworthy that after obtaining the result ciphertexts of the homomorphic ReLU operation on the small integers, the LargeIntReLU algorithm does not merge the multiple result ciphertexts to obtain the large integer ciphertext after the homomorphic ReLU operation. This is because, as shown in [13], under current algorithm parameters, when the integer precision exceeds 6 bits, the operational success probability drops significantly. Therefore, the merged large integer ciphertext cannot participate in subsequent ciphertext-based homomorphic computations. Merging the result ciphertexts has no practical significance, so this paper does not perform ciphertext merging.

4.4. Correctness Analysis

Since the correctness of integer-supporting fully homomorphic encryption algorithms is related to noise, and the variance of noise in these algorithms depends on the algorithm parameters, this section first introduces the algorithm parameters. The parameters used in this chapter are consistent with the TFHE scheme [13,17], as shown in Table 3.

Table 3. Parameter settings for LargeIntReLU.

Parameter	Symbol	Value
LWE dimension	n	500
TRLWE dimension	N	1024
TRGSW dimension	l	2
Gadget decomposition precision	β	512
Gadget decomposition accuracy	ϵ	2^{-21}
BK noise std dev	α_{bk}	3.73×10^{-9}
Key-switching decomposition precision	t	16
KS noise std dev	γ	2.16×10^{-5}

Substituting the parameters from Table 3 into Equation (6), the error variance on the left side of the inequality can be calculated as 1.63×10^{-5}. The corresponding standard deviation is denoted by σ, where $\sigma = 4.05 \times 10^{-3}$. Since in the proposed homomorphic ReLU algorithm LargeIntReLU supporting full-domain bootstrapping for large integers, the precision of small integers is uniformly set to 6 bits, the upper limit B for small integer

values is set to 32, and integer bootstrapping for large integer ciphertexts is all signed integer bootstrapping (with B fixed at 32), the correctness probability of the proposed algorithm is as shown in Table 4 [13]. It can be obtained that the correctness probability of the proposed homomorphic ReLU algorithm LargeIntReLU supporting full-domain bootstrapping for large integers is 95.2%.

Table 4. Error under different integer intervals [13].

Integer Precision	Integer Interval Parameter B	Standard Deviation σ	Algorithm Correctness Probability $erf(\frac{1}{4B\sqrt{2\sigma}})$
4	8	4.05×10^{-3}	99.99%
5	16	4.05×10^{-3}	99.99%
6	32	4.05×10^{-3}	95.2%
7	64	4.05×10^{-3}	66.3%

4.5. Formal Correctness Analysis

To address the formal correctness requirement, we present the following theorem and proof.

Theorem 1 (Correctness of SignedDecomp)**.** *Given a large integer ciphertext C_x with precision Q ($7 \leq Q \leq 11$) and the parameter set $\mathcal{P}$ as defined in Table 2, the SignedDecomp algorithm (Algorithm 4) outputs $n = \lceil Q/5 \rceil$ small integer ciphertexts $C_{x_1}, \ldots, C_{x_n}$, such that for the underlying plaintexts $x_1, \ldots, x_n$ the following holds with probability at least* 95.2%*:*

$$x = x_1 + \sum_{i=2}^{n} x_i \cdot 32^{i-1}, \tag{20}$$

where each x_i is a signed 6-bit integer, and the decomposition process does not introduce arithmetic errors.

Proof. The correctness of SignedDecomp hinges on the sequential correctness of its three atomic operations: LeftShift, HomMod, and CipherClean.

Step 1 (LeftShift). For a ciphertext $(\mathbf{a}, b)$ encrypting m, the operation $\mathsf{LeftShift}((\mathbf{a}, b), k)$ outputs $(\mathbf{a}', b') = (2^k \cdot \mathbf{a}, 2^k \cdot b)$. This linearly scales the plaintext noise by 2^k without altering the plaintext structure, i.e., $b' - \mathbf{a}' \cdot \mathbf{sk} = 2^k \cdot (b - \mathbf{a} \cdot \mathbf{sk})$. Therefore, it preserves correctness deterministically.

Step 2 (HomMod). Given modulus $q = 2^{32-\mathrm{num}_{\mathrm{mod}}}$, the modulo operation is performed on each coefficient of $\mathbf{a}$ and b to obtain $\mathbf{a} \mod q$ and $b \mod q$. As shown in Equation (8), this homomorphic modular reduction preserves the linear relation due to the binary secret key $\mathbf{sk} \in \{0,1\}^n$. The subsequent left-shift by $\mathrm{num}_{\mathrm{mod}} - 1$ bits correctly aligns the segment without introducing carry propagation.

Step 3 (CipherClean). For each bit position j ($1 \leq j \leq \mathrm{num}$), the algorithm extracts the j-th highest bit via left-shift and bootstrapping with function f_{sign3} (Equation (9)). The correctness of this bit extraction relies on the correctness of the underlying integer bootstrapping IntBoots. With parameters from Table 2, the probability of a single bootstrapping being correct is 95.2% for 6-bit integers (as per Table 3). The sequential composition of num such operations (where $\mathrm{num} \leq 6$) results in an overall correctness probability of at least 95.2% (the worst-case per-operation error is already accounted for in the bootstrapping error bound). Finally, the coordination of these three steps in SignedDecomp ensures that each 6-bit segment x_i is correctly extracted and signed, and that the original large integer x can be reconstructed via Equation (20). The overall probability is dominated by the bootstrapping steps, which is empirically validated as 95.2%. □

Theorem 2 (Correctness of LargeIntReLU)**.** *Let C_x be a ciphertext of a large integer x with precision $Q > 6$. The LargeIntReLU algorithm (Algorithm 5) outputs ciphertexts $C_{out_1}, \ldots, C_{out_n}$ such that:*

$$ReLU(x) = out_1 + \sum_{i=2}^{n} out_i \cdot 32^{i-1}, \tag{21}$$

with probability at least 95.2%*, where $out_i = ReLU(x_i)$ and x_i are the segments from SignedDecomp.*

Proof. The proof follows from the composition of Theorem 1 and the correctness of the underlying SmallIntReLU algorithm. Since SignedDecomp correctly obtains segments x_i with probability 95.2%, and SmallIntReLU correctly computes $\mathrm{ReLU}(x_i)$ for each 6-bit segment with the same probability, the overall algorithm correctness is bounded by the product of individual success probabilities. Given the high per-operation correctness (95.2%) and the low number of segments ($n \leq 3$ for $Q \leq 11$), the overall probability remains $\geq$95.2%, as confirmed by our experimental results. □

The correctness probability of 95.2% reported in Table 3 is obtained under the following experimental conditions:

- **Test samples:** 10,000 randomly generated integers uniformly distributed over the full range of the 6-bit signed domain $[-32, 31]$.
- **Noise distribution:** The error terms in LWE ciphertexts follow a discrete Gaussian distribution with standard deviation $\sigma = 4.05 \times 10^{-3}$ (derived from Equation (6) and parameters in Table 2).
- **Hardware/Software:** Experiments were conducted on an Alibaba Cloud instance (Intel Xeon Platinum 8369HB @3.30 GHz, 128 GB RAM) using the TFHE library [6].

4.6. Security Analysis

The security of the LargeIntReLU algorithm is proven below. Formally, Theorem 3 describes the security of the LargeIntReLU algorithm.

Theorem 3. *In the* LargeIntReLU *algorithm, when the algorithm parameters $n, N, l, \beta, \epsilon, \alpha_{bk}, t$, and γ are the same as in the TFHE scheme [17], the* LargeIntReLU *algorithm is secure.*

Proof. Since the security of fully homomorphic encryption algorithms is determined by the algorithm parameters, and the LargeIntReLU algorithm and all other algorithms it calls adopt the same parameters as the TFHE scheme [17], the security of the LargeIntReLU algorithm can be reduced to the security of the TFHE scheme [17]. The TFHE scheme [17] has been systematically proven secure. Therefore, the LargeIntReLU algorithm proposed in this paper is secure. □

Information-Leakage Analysis

We further address the concern about potential information leakage during the decomposition process. Each atomic operation in SignedDecomp—LeftShift, HomMod, and CipherClean—is a *publicly computable function* of the ciphertext and does not depend on the secret key. Moreover, all bootstrapping operations (via IntBoots) are performed using public bootstrapping keys. Therefore, the entire decomposition process can be simulated by a probabilistic polynomial-time simulator that only has access to the ciphertext (and public parameters), without knowing the plaintext. This satisfies the standard notion of *circuit privacy* in the context of FHE, ensuring that the decomposition does not reveal any additional information beyond the output segments.

5. Algorithm Performance Analysis

This section presents a comprehensive evaluation of the proposed LargeIntReLU algorithm, analyzing its correctness, security, and efficiency through both theoretical comparison and experimental validation.

5.1. Benchmark Algorithm: Bit Decomposition for Comparison

To enable fair efficiency comparison with bit-wise homomorphic ReLU algorithms, we first introduce a baseline bit decomposition method. Since existing bit-wise TFHE schemes typically decompose large integers in plaintext before encryption, no ciphertext-level bit decomposition algorithm exists for direct comparison. We therefore propose the BitDecomp algorithm to decompose large integer ciphertexts into individual bit ciphertexts, serving as a reference benchmark for evaluating our approach.

The BitDecomp algorithm (Algorithm 6) sequentially processes each bit position by shifting the target bit to the highest position using LeftShift, determining its sign via IntBoots with function f_{sign6} (Equation (13)), and finally extracting the bit value through constant subtraction. This decomposition enables direct comparison with bit-wise ReLU approaches.

Algorithm 6 Bit Decomposition Algorithm BitDecomp(C_x, n, bk, ks).

Require: Large integer ciphertext C_x, large integer precision n, bootstrapping key bk, key-switching key ks, where C_x is the ciphertext of x
Ensure: Ciphertexts $C_{x_1}, \cdots, C_{x_n}$ satisfying $x = \sum_{i=1}^{n} x_i \cdot 2^{i-1}$
1: **for** $i = 1$ **to** n **do**
2: Compute $C_{tmp1} = \mathsf{LeftShift}(C_x, i-1)$
3: Compute $C_{tmp2} = \mathsf{IntBoots}(C_{tmp1}, f_{sign6}, bk, ks)$
4: Compute $C_{x_{n-i+1}} = C_{tmp2} - (0, 16 \times \frac{1}{2B})$ {Subtract ciphertext of constant $16 \times \frac{1}{2B}$}
5: **end for**
6: **return** Ciphertexts $C_{x_1}, \cdots, C_{x_n}$

5.2. Theoretical Complexity Analysis

We now theoretically compare LargeIntReLU against two state-of-the-art approaches: the bit-data based LargeBitReLU [7] and the integer-based Loris [14]. Given that bootstrapping dominates FHE computation time (over 95%), we focus on bootstrapping operation counts as the primary complexity metric. Table 5 summarizes the theoretical bootstrapping requirements across different integer precisions.

Table 5. Bootstrapping operations comparison.

Algorithm	Data Type	Number of Boostrapping for n-Bit Precision
LargeBitReLU [7]	Bit	$2n-1$
Loris [14]	Integer	$6\lceil\frac{n-1}{5}\rceil + n - 7$
LargeIntReLU (Ours)	Integer	$4\lceil\frac{n-1}{5}\rceil + n - 5$

The LargeBitReLU approach builds upon Huang et al.'s BitReLU [13], which requires bit-wise plaintext decomposition before encryption. For ciphertext operations, LargeBitReLU first applies our BitDecomp algorithm (requiring n bootstrappings) followed by BitReLU (requiring $n-1$ bootstrappings), totaling $2n-1$ operations.

The Loris algorithm employs plaintext decomposition into small integers, requiring $5\lceil\frac{n-1}{5}\rceil - 2$ bootstrappings for full-domain evaluation. To enable ciphertext-level comparison, it would need an additional decomposition overhead of $\lceil\frac{n-1}{5}\rceil + n - 5$ bootstrappings, resulting in $6\lceil\frac{n-1}{5}\rceil + n - 7$ total operations.

https://doi.org/10.3390/cryptography10020021

Our LargeIntReLU requires $\lceil \frac{n-1}{5} \rceil + n - 5$ bootstrappings for decomposition (2 for large integer sign determination, $\lceil \frac{n-1}{5} \rceil - 1$ for small integer signs, and $n - 6$ for ciphertext cleaning), plus $3\lceil \frac{n-1}{5} \rceil$ for small integer ReLU operations, totaling $4\lceil \frac{n-1}{5} \rceil + n - 5$ bootstrappings.

As evident from Table 5, LargeIntReLU demonstrates clear theoretical advantages with consistently lower bootstrapping requirements across all precision levels.

5.3. Experimental Validation

We implemented LargeIntReLU based on the TFHE library [6] and evaluated it on an Alibaba Cloud server (Intel Xeon Platinum 8369HB @3.30 GHz, 128 GB RAM, Ubuntu 20.04). Due to the unavailability of practical implementations for Loris [14], we compare exclusively against LargeBitReLU across different precision levels, with execution times detailed in Table 6.

Table 6. Computation time comparison (seconds).

Algorithm	Library	Data Type	Bootstrapping Time for n Bit-Precision				
			$n = 7$	$n = 8$	$n = 9$	$n = 10$	$n = 11$
LargeBitReLU [7]	TFHE	Bit	1.498	1.734	1.964	2.199	2.423
LargeIntReLU (Ours)	TFHE	Integer	1.160	1.273	1.393	1.506	1.621

Note: Because Loris [14] does not have open source code, we cannot compare with Loris in the table.

Limitations and Future Work

We acknowledge that while our theoretical analysis and micro-benchmarks demonstrate the efficiency of LargeIntReLU, the current experimental validation does not include end-to-end performance testing on actual neural network models (e.g., CNN, MLP). Such evaluation is essential to verify the practical applicability of the scheme in real-world homomorphic inference pipelines. In an effort to address this, we attempted to integrate LargeIntReLU into the REDsec framework for encrypted MNIST inference. However, we encountered a fundamental obstacle: the ReLU module in REDsec introduces ciphertext-specific errors that were not accounted for during plaintext model training. Consequently, while the plaintext model achieved normal accuracy (99%), its encrypted inference accuracy dropped to approximately 10%—equivalent to random guessing. This confirms that models trained solely on plaintext cannot be directly deployed in homomorphic inference pipelines; cryptographic noise must be incorporated during training (e.g., via noise-aware or quantization-aware training).

Resolving this issue involves objective difficulties beyond time constraints. Accurately modeling REDsec's error distribution requires detailed knowledge of its internal bootstrapping parameters and encoding schemes—information not available in public documentation. Modifying the core codebase to enable noise injection would require maintainer privileges and would risk introducing vulnerabilities. Moreover, REDsec offers no APIs for integrating training-time noise simulation. We have contacted the original developers through multiple channels for guidance, but as of the submission deadline, we have not received a response. Without their assistance, we cannot safely adapt the library for noise-aware training.

We have documented this limitation and plan to address it in future work by either establishing collaborative opportunities with the REDsec developers or exploring alternative frameworks that better support noise-aware training integration. Once resolved, we will conduct comprehensive end-to-end tests on typical datasets (e.g., MNIST, CIFAR-10) and report accuracy, latency, and other relevant metrics. We believe that the theoretical contributions and component-level evaluations presented in this paper remain valuable and provide a solid foundation for subsequent practical deployments.

6. Conclusions

This paper successfully addressed the limitation of current integer TFHE algorithms by proposing a novel homomorphic ReLU algorithm that supports full-domain bootstrapping for large integers. The core of this contribution is a signed large integer ciphertext decomposition algorithm, which was enabled by the newly designed LeftShift, HomMod, and CipherClean algorithms. This decomposition method effectively resolves the issues of sign-bit retention and carry interference that plagued previous approaches. By integrating this decomposition technique with an existing small integer ReLU algorithm, we constructed a complete and efficient solution for homomorphically evaluating the ReLU function on large integers. Theoretical analysis and experimental results confirm the algorithm's correctness and security, demonstrating a significant performance improvement with an average reduction in operation time of 28.58% compared to existing methods.

While the current work focuses on the ReLU activation function, the decomposition framework opens several avenues for future research. An immediate direction is to extend the signed decomposition approach to support other non-linear activation functions essential for deep learning, including Sigmoid and GeLU. This would require designing sign-preserving decomposition algorithms and approximating the target functions within each six-bit segment using low-degree polynomials, while carefully managing approximation error and precision constraints. Another important goal is to integrate LargeIntReLU into a complete homomorphic neural network inference system and evaluate its end-to-end performance on real-world encrypted datasets (e.g., medical or financial data). Although we encountered stability issues with the underlying cryptographic library when attempting full-network tests, we have documented these limitations and plan to address them in future work by either rectifying the ReLU module or adopting a more robust implementation. Once resolved, comprehensive end-to-end experiments will further demonstrate the practical utility of our algorithm in privacy-preserving machine learning applications.

Author Contributions: Conceptualization, Y.H., J.F. (Jingjing Fan) and J.F. (Junbin Fang); methodology, Y.L., Y.H. and X.T.; software, Y.L., Y.H. and Q.X.; validation, Y.L., Y.H. and Q.X.; formal analysis, Y.L., Y.H. and X.T.; investigation, Y.L., Y.H. and Q.X.; resources, J.F. (Junbin Fang) and Z.-L.J.; data curation, Y.L., Y.H. and Q.X.; writing—original draft preparation, Y.L., Y.H. and X.T.; writing—review and editing, Z.-L.J., J.F. (Jingjing Fan), J.F. (Junbin Fang) and X.Z.; visualization, Y.L., Y.H. and Q.X.; supervision, Z.-L.J., J.F. (Jingjing Fan), J.F. (Junbin Fang) and X.Z.; project administration, Z.-L.J., J.F. (Jingjing Fan), J.F. (Junbin Fang) and X.Z.; funding acquisition, Z.-L.J., J.F. (Junbin Fang) and X.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This work is supported by Shenzhen Science and Technology Program (KJZD20240903104301003); the National Natural Science Foundation of China under Grant 62272131; the Shenzhen Social Science Fund under Grant SZ2025B011 and the Shenzhen Science and Technology Program under Grant SYSPG20241211173609009; PCL-CMCC Foundation for Science and Innovation (Grant No. 2024ZY2B0050).

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Gentry, C. Fully Homomorphic Encryption Using Ideal Lattices. In Proceedings of the 41st Annual ACM Symposium on Theory of Computing, Bethesda, MD, USA, 31 May–2 June 2009; pp. 169–178.
2. Marcolla, C.; Sucasas, V.; Manzano, M.; Bassoli, R.; Fitzek, F.H.P.; Aaraj, N. Survey on Fully Homomorphic Encryption, Theory, and Applications. *Proc. IEEE* **2022**, *110*, 1572–1609. [CrossRef]

3. Wang, N.; Zhou, W.; Wang, J.; Guo, Y.; Fu, J.; Liu, J. Secure and Efficient Similarity Retrieval in Cloud Computing Based on Homomorphic Encryption. *IEEE Trans. Inf. Forensics Secur.* **2024**, *19*, 2454–2469. [CrossRef]
4. Fukuchi, Y.; Hashimoto, S.; Sakai, K.; Fukumoto, S.; Sun, M.; Ku, W. Secure kNN for Distributed Cloud Environment Using Fully Homomorphic Encryption. *IEEE Trans. Cloud Comput.* **2025**, *13*, 721–736. [CrossRef]
5. Meftah, S.; Tan, B.H.M.; Mun, C.F.; Aung, K.M.M.; Veeravalli, B.; Chandrasekhar, V. DOReN: Toward Efficient Deep Convolutional Neural Networks with Fully Homomorphic Encryption. *IEEE Trans. Inf. Forensics Secur.* **2021**, *16*, 3740–3752. [CrossRef]
6. Chillotti, I.; Gama, N.; Georgieva, M.; Izabachene, M. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In Proceedings of the 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, 4–8 December 2016; Volume 10031, pp. 3–33.
7. Lou, Q.; Jiang, L. SHE: A Fast and Accurate Deep Neural Network for Encrypted Data. In Proceedings of the 33rd Neural Information Processing Systems, Vancouver, BC, Canada, 8–14 December 2019; Volume 32, pp. 10035–10043.
8. Bourse, F.; Minelli, M.; Minihold, M.; Paillier, P. Fast Homomorphic Evaluation of Deep Discretized Neural Networks. In Proceedings of the 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, 19–23 August 2018; pp. 483–512.
9. Okada, H.; Kiyomoto, S.; Cid, C. Integerwise Functional Bootstrapping on TFHE. In Proceedings of the 23rd International Conference on Information Security, Bali, Indonesia, 16–18 December 2020; pp. 107–125.
10. Kluczniak, K.; Schild, L. FDFB: Full Domain Functional Bootstrapping Towards Practical Fully Homomorphic Encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2022**, *2023*, 501–537.
11. Clet, P.E.; Zuber, M.; Boudguiga, A.; Sirdey, R.; Gouy-Pailler, C. Putting Up the Swiss Army Knife of Homomorphic Calculations by Means of TFHE Functional Bootstrapping. *Cryptol. Eprint Arch.* **2022**, 2022/149. Available online: https://eprint.iacr.org/2022/149.pdf (accessed on 15 March 2026).
12. Yang, Z.; Xie, X.; Shen, H.; Chen, S.; Zhou, J. TOTA: Fully Homomorphic Encryption with Smaller Parameters and Stronger Security. *Cryptol. Eprint Arch.* **2021**, 2021/1347. Available online: https://eprint.iacr.org/2021/1347 (accessed on 15 March 2026).
13. Huang, Y.; Wan, J.; Jiang, Z.L.; Zhou, J.; Fang, J.; Cao, Z. An Efficient Integer-Wise ReLU on TFHE. In *Proceedings of the Information Security and Privacy—29th Australasian Conference, ACISP 2024, Sydney, NSW, Australia, 15–17 July 2024*; Proceedings, Part I; Lecture Notes in Computer Science; Zhu, T., Li, Y., Eds.; Springer: Berlin/Heidelberg, Germany, 2024; Volume 14895, pp. 161–179.
14. Bergerat, L.; Boudi, A.; Bourgerie, Q.; Chillotti, I.; Ligier, D.; Orfila, J.B.; Tap, S. Parameter Optimization and Larger Precision for (T) FHE. *J. Cryptol.* **2023**, *36*, 28–100. [CrossRef]
15. Chillotti, I.; Ligier, D.; Orfila, J.B.; Tap, S. Improved Programmable Bootstrapping with Larger Precision and Efficient Arithmetic Circuits for TFHE. In Proceedings of the 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, 6–10 December 2021; pp. 670–699.
16. Regev, O. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. *J. ACM* **2009**, *56*, 1–40. [CrossRef]
17. Chillotti, I.; Gama, N.; Georgieva, M.; Izabachène, M. TFHE: Fast Fully Homomorphic Encryption over the Torus. *J. Cryptol.* **2020**, *33*, 34–91. [CrossRef]
18. Liu, Z.; Micciancio, D.; Polyakov, Y. Large-Precision Homomorphic Sign Evaluation Using FHEW/TFHE Bootstrapping. In Proceedings of the 28th International Conference on the Theory and Application of Cryptology and Information Security, Taiwan, China, 5–9 December 2022; Volume 13792, pp. 130–160.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.