XBOOT: Free-XOR Gates for CKKS with Applications to Transciphering

Chao Niu, Zhicong Huang, Zhaomin Yang, Yi Chen, Liang Kong, Cheng Hong* and Tao Wei

Ant Group
{niuchao.niu,zhicong.hzc,yangzhaomin.yzm,cy459642,kongliang.kong,vince.hc,lenx.wei}@antgroup.com

Abstract. The CKKS scheme is traditionally recognized for approximate homomorphic encryption of real numbers, but BLEACH (Drucker et al., JoC 2024) extends its capabilities to handle exact computations on binary or small integer numbers. Despite this advancement, BLEACH's approach of simulating XOR gates via $(a-b)^2$ incurs one multiplication per gate, which is computationally expensive in homomorphic encryption. To this end, we introduce XBOOT, a new framework built upon BLEACH's blueprint but allows for almost free evaluation of XOR gates. The core concept of XBOOT involves lazy reduction, where XOR operations are simulated with the less costly addition operation, $a+b$, leaving the management of potential overflows to later stages. We carefully handle the modulus chain and scale factors to ensure that the overflows would be conveniently rounded during the CKKS bootstrapping phase without extra cost. We use AES-CKKS transciphering as a benchmark to test the capability of XBOOT, and achieve a throughput exceeding one kilobyte per second, which represents a 2.5× improvement over the state-of-the-art (Aharoni et al., HES 2023). Moreover, XBOOT enables the practical execution of tasks with extensive XOR operations that were previously challenging for CKKS. For example, we can do Rasta-CKKS transciphering at over two kilobytes per second, more than 10× faster than the baseline without XBOOT.

Keywords: homomorphic encryption · bootstrapping · advanced encryption standard · hybrid homomorphic encryption · transciphering

1 Introduction

Fully Homomorphic Encryption (FHE) [Gen09] represents a groundbreaking paradigm in cryptography, allowing computations to be performed directly on encrypted data without decryption. Initially regarded as purely theoretical and impractical, FHE has seen significant advancements in recent years, leading to the proposal of numerous modern FHE schemes [BV11, FV12, GSW13, CCK+13, BGV14, BV14, DM15, CKKS17, CGGI17, CGGI20] with markedly improved efficiency. Among the various FHE schemes, the Cheon-Kim-Kim-Song (CKKS) scheme [CKKS17] stands out as one of the most efficient, and has been extensively utilized in works such as privacy-preserving machine learning [LLL+22, LLKN23, JPK+24].

However, CKKS is inherently an approximate homomorphic encryption scheme; it introduces noise in the computation, which accumulates with each operation, potentially leading to loss of precision over multiple computational layers. As a result, it was believed that CKKS could only be used for applications where exact precision is not

*Corresponding author: Cheng Hong (vince.hc@antgroup.com) always necessary, and should not be used in applications that are sensitive to even slight variations. To mitigate the limitations posed by CKKS's approximation, the BLEACH methodology [DMPS24] was introduced. BLEACH provides an efficient CKKS framework for bit operations by representing Boolean values with real numbers around 0 or 1 (e.g., 0.002 or 0.997), and simulate an AND gate by multiplication $a * b$, and an XOR gate by the operation $(a - b)^2$. It employs a "cleanup" function $h_1(x) = 3x^2 - 2x^3$ to reduce errors accumulated, enabling CKKS to maintain a high accuracy for discrete computations, particularly for Boolean circuits. This innovation extends the usage of CKKS into areas previously dominated by exact FHE schemes like BFV/BGV or TFHE.

One significant application in these areas is **transciphering**. In this methodology, clients can utilize inexpensive symmetric encryption (such as AES) instead of FHE, transmitting the symmetric ciphertext to the server, who then converts this ciphertext into an FHE ciphertext to enable subsequent computations. Transciphering significantly lowers both the computational and communication demands on the client, thereby offering considerable advantages for resource-limited devices like embedded systems. The core principle of transciphering involves the homomorphic evaluation of the symmetric decryption function, which is generally treated as a Boolean circuit. Due to the exceptional efficiency of the CKKS scheme, the throughput of AES-CKKS transciphering [ADE+23] using BLEACH is often higher compared to AES transciphering to other FHE schemes [GHS12, SMK22, TCBS23, WLW+24, BPR24, SAP24].

Despite the advancements, a significant drawback in BLEACH is the cost associated with the XOR operation. Specifically, since CKKS only supports addition and multiplication, BLEACH has to simulate the XOR gate using the formula $(a - b)^2$, which requires one multiplication operation per XOR gate. Given that ciphertext-ciphertext multiplications in FHE are expensive, the number of XOR gates can lead to performance bottlenecks.

What's worse, cryptography researchers tend to focus on optimizing algorithms by reducing the number of non-XOR gates, and pay less attention to XOR gates since they are cheap in secure multi-party computation (MPC) or exact FHE schemes like BGV. For example, the optimized AES circuit described in [KSS12] comprises 9,100 non-XOR gates and 21,628 XOR gates. The complexity of LowMC [ARS+15] and Rasta [DEG+18] ciphers is estimated solely based on AND gates without consideration of XOR gates. This discrepancy presents an ongoing challenge when handling Boolean circuits with CKKS, and we pose the following question:

Can we homomorphically evaluate Boolean circuits in a way that leverages the high efficiency of CKKS while reducing the cost of XOR gates?

1.1 Our Contributions

In this work, we propose XBOOT, a new CKKS framework for computation on binary values. XBOOT successfully solved the above issues by enabling almost "free" XOR gates. Our contributions are as follows:

Free-XOR framework for CKKS Building upon the work of BLEACH, we introduce a lazy reduction technique where XOR gates are substituted with simple additions. This method allows intermediate results to temporarily exceed their intended range. Each such result can be conceptualized as consisting of two segments: the Least Significant Bit (LSB) part, which retains the actual result, and a redundant part. Then we carefully design the CKKS modulus chain and scale factors so that the redundant part would be effectively removed at the SlotsToCoeffs step of CKKS bootstrapping process. Thereby the LSB is automatically extracted, achieving the XOR functionality almost "for free".

Highly efficient transciphering. We use transciphering as a representative application to demonstrate the efficiency of XBOOT. The state-of-the-art on AES-CKKS transciphering [ADE+23] consumes 9 multiplication depths per round, while with XBOOT it will only require 3 multiplication depths per round. With the reduced multiplication depth, and combining the optimizations from CKKS bootstrapping for bits [BCKS24], the bit length of CKKS ciphertext modulus can be efficiently reduced to 830 bits from 1555 bits, and the cyclotomic ring degree N can be reduced to 2^{15} from 2^{16} while still maintaining 128-bit security. Experiments show that we are able to do AES transciphering at the speed of 256KB per 236 seconds, achieving a $5\times$ improvement in latency and $2.5\times$ in throughput compared to [ADE+23].

Furthermore, we highlight that numerous applications involve an extensive number of XOR gates, rendering them impractical for evaluation under traditional CKKS frameworks. XBOOT could address this limitation, thereby unlocking new possibilities for such tasks. For instance, the Rasta cipher [DEG+18], which incorporates nearly a million XOR gates, serves as a compelling example. By applying XBOOT to Rasta-CKKS transciphering, we achieve a remarkable throughput of over two kilobytes per second. This would outperform the other Rasta transciphering methods by more than one order of magnitude.

1.2 Related Work

CKKS bootstrapping. Bootstrapping for CKKS was first introduced in [CHK+18]. Since then, continuous advancements have been made. Notable progress includes optimizations in homomorphic modular reduction [CHK+18, LLL+21, JM22, LLK+22], as well as complexity reductions for linear transformations, with FFT-like decomposition applied in [CCS19, HK20, BMTH21].

Several novel CKKS bootstrapping techniques have been proposed, including methods for achieving unlimited precision [BCC+22], bootstrapping BFV/BGV schemes using CKKS to remove restrictions on plaintext modulus [KDE+24], and approaches for DM/CGGI functional bootstrapping with LUTs that incorporate CKKS within the bootstrapping procedure [AKP24].

More recently, new techniques have been developed [BCKS24, BKSS24] that enhance functionality by implementing polynomial approximations for various trigonometric functions. Additionally, an iterative most significant bit (MSB) bootstrapping method [KN24] has been proposed for bootstrapping integers within the large range of $[0, 2^{20})$, further enhancing the capabilities of the CKKS scheme.

Transciphering. Since Gentry et al. introduced the first homomorphic computation for AES, transciphering for AES serves as a benchmark for evaluating the efficiency of FHE. The specific evaluation strategies depend on the unique characteristics of each FHE scheme.

Besides CKKS, the TFHE scheme, especially with functional bootstrapping [CGGI20], has been extensively studied for its application in transciphering. Stracovsky et al. [SMK22] evaluated an AES block in 4 minutes using 16 threads. More recently, Trama et al. [TCBS23] proposed a faster AES evaluation using functional and multi-value bootstrapping techniques. Recent optimization works, such as [WWL+23, WWL+24], have reported even better results based on the rich bootstrapping features of the TFHE scheme.

Note that several works have leveraged the plaintext encoding space to enable free XOR in TFHE by automatically applying modulo two in the plaintext domain [WWL+23, CLW+24, SAP24]. However, since ciphertext-ciphertext multiplications are inefficient in TFHE, a drawback is that they require additional efforts to handle AND gates over modulo two. In contrast, our free XOR framework does not introduce any negative side effects for AND gates.

Generally speaking, compared to CKKS transciphering, TFHE transciphering typically offers less throughput due to the absence of SIMD properties. And efficient handling of XOR gates is also a hot research topic in TFHE.

2 Preliminaries

In this section, we outline the essential background knowledge of FHE and transciphering and establish the notations used throughout the paper. For a power-of-two integer N, we define the N-degree polynomial ring as $\mathcal{R}_N = \mathbb{Z}[X]/(X^N + 1)$ and the residual ring $\mathcal{R}_{Q,N} = \mathcal{R}_N / Q\mathcal{R}_N$, which is formed by taking $\mathcal{R}_N$ modulo an integer Q.

2.1 Approximate Homomorphic Encryption (CKKS)

We revisit the CKKS scheme, which is defined within this polynomial space.

Notations. We denote individual elements—such as numbers or polynomials—in italics, e.g. a and b, and their vectors in bold, e.g., $\mathbf{a}$ and $\mathbf{b}$. The notation $\langle \mathbf{a}, \mathbf{b} \rangle$ represents the inner product between the two vectors. We denote $\|a\|_\infty$ as the infinity norm of the vector (or polynomial) a in the power basis and $\mathsf{hw}(a)$ as the Hamming weight of the vector (or polynomial) a. We denote $x \mod Q$ as the remainder of x modulo Q, $\lfloor x \rfloor$, $\lceil x \rceil$, and $\lfloor x \rceil$ as the rounding of x to the nearest previous, next and closest integer, respectively. If x is a polynomial, these operations are performed on each coefficient.

Given an integer a, its modulo q is denoted as $[a]_q$. If $\mathcal{C} = \{q_0, q_1, \ldots, q_L\}$ is a set of pairwise co-prime positive integers (modulus chain), and $a \in \mathbb{Z}_{Q_L}$ where $Q_L = \prod_{i=0}^{L} q_i$, is the top modulus. The modulus at the level $\ell \in \{0, 1, ..., L\}$ is denoted as $Q_\ell = \prod_{i=0}^{\ell} q_i$. Then the RNS representation of a with respect to $\mathcal{C}$ is denoted by $[a]_\mathcal{C} = ([a]_{q_0}, [a]_{q_1}, \ldots, [a]_{q_L}) \in \mathbb{Z}_{q_0} \times \cdots \times \mathbb{Z}_{q_L}$. The base of the logarithm in this paper is two.

Let $\mathsf{DFT} : \mathbb{R}[X]/X^N + 1 \to \mathbb{C}^{N/2}$ be a variant of the discrete Fourier transform defined as evaluation of $N/2$ points:

$$\forall p \in \mathcal{R}_N : \mathsf{DFT}(p) = (p(\zeta^{5^i}))_{0 \le i < N/2},$$

where ζ is a primitive complex $(2N)$-th root of unity. The inverse of this process is denoted as iDFT: $\mathbb{C}^{N/2} \to \mathbb{R}[X]/X^N + 1$.

Basic operations. We first recall some necessary operations defined in the CKKS scheme [CKKS17, CHK+18].

- $\mathsf{Ecd}(\mathbf{m},\Delta,N)$ (*coefficient→slots*) Given a message $\mathbf{m} \in \mathbb{C}^{N/2}$, the canonical embedding $\mathbb{C}^{N/2} \to \mathbb{R}[X]/X^N + 1 \to \mathcal{R}_N$ is defined as:

$$\forall \mathbf{m} \in \mathbb{C}^{N/2} : \mathsf{Ecd}(\mathbf{m}) = \lfloor \Delta \cdot \mathsf{iDFT}(\mathbf{m}) \rceil$$

  where Δ represents the scaling factor that controls the encoding precision. After this process, the data $\mathbf{m} \in \mathbb{C}^{N/2}$ to be operated on is stored in the slots, with the plaintext polynomial rounded to $\mathsf{Ecd}(\mathbf{m})$. Ultimately, the multiplication of polynomial is mapped as component-wise multiplication of scaled $\mathbf{m}$.

- $\mathsf{Dcd}(p,\Delta,N)$ (*slots→coefficient*) Given polynomial $p \in \mathcal{R}_N$, apply the inverse process of Ecd, defined as:

$$\forall p \in \mathcal{R}_N : \mathsf{Dcd}(p) = \frac{1}{\Delta} \cdot \mathsf{DFT}(p).$$

Recall that the ciphertext $\mathsf{ct} = (b, a) \in \mathcal{R}^2_{Q,N}$ decrypts to the plaintext polynomial under a secret key $\mathsf{sk} = (1, s)$ as $\langle \mathsf{ct}, \mathsf{sk} \rangle = b + as = m$, where $m \in \mathcal{R}_N$.

Key switching (KS) is required after homomorphic multiplication and automorphism. It is defined as $\mathsf{KS}(\mathsf{ct}, \mathsf{swk})$, where swk is generated from the previous secret key sk to facilitate multiplication, conjunction, and rotations. We briefly recall the homomorphic operations as follows.

- $\mathsf{Add}(\mathsf{ct}_1,\mathsf{ct}_2)$: For $\mathsf{ct}_1, \mathsf{ct}_2 \in \mathcal{R}^2_{Q_\ell}$, output $\mathsf{ct}_{\text{add}} = \mathsf{ct}_1 + \mathsf{ct}_2 \pmod{Q_\ell}$.
- $\mathsf{Mult}(\mathsf{ct}_1,\mathsf{ct}_2)$: Given $\mathsf{ct}_1, \mathsf{ct}_2 \in \mathcal{R}^2_{Q_\ell}$, output a level-downed ciphertext $\mathsf{ct}_{\text{mult}} \in \mathcal{R}^2_{Q_{\ell-1}}$. This operation applies tensor product of $\mathsf{ct}_{\text{new}} = \mathsf{ct}_1 \otimes \mathsf{ct}_2 \in \mathcal{R}^3_{Q_\ell}$, relinearization $\mathsf{ct}_{\text{rel}} = \mathsf{KS}(\mathsf{ct}_{\text{new}}, \mathsf{swk}_{\text{rel}}) \in \mathcal{R}^2_{Q_\ell}$ and rescale $\mathsf{ct}_{\text{mult}} = \mathsf{RS}(\mathsf{ct}_{\text{rel}}) \in \mathcal{R}^2_{Q_{\ell-1}}$.
- $\mathsf{cMult}(\mathsf{ct}, a)$: For $a \in \mathbb{Z}$, output $\mathsf{ct}_{\text{cmult}} = a \cdot \text{ct}$ without applying $\mathsf{Ecd}(\cdot)$, relinearization and rescale $\mathsf{RS}(\mathsf{ct})$ after Mult.
- $\mathsf{RS}(\mathsf{ct})$: For $\mathsf{ct} \in R^2_{Q_\ell}$, output $\mathsf{ct}_{\mathsf{RS}} = \lfloor q_\ell^{-1} \cdot \mathsf{ct} \rceil \pmod{Q_{\ell-1}}$. The rescaling process serves two purposes: it reduces the error size and maintains the scaling factor for each slot. Rescale is alongside the reduction of the modulus chain.

Bootstrapping. The bootstrapping procedure of the CKKS scheme [CHK+18] seeks to elevate the ciphertext to a higher modulus, enabling further homomorphic evaluation. The CKKS bootstrapping produces a ciphertext that increases the modulus back to certain point, with $Q_{\text{rem}} \gg q_0$, where Q_{rem} is the modulus after bootstrapping. The procedure consists of four steps: $\mathsf{SlotsToCoeffs}$, $\mathsf{ModRaise}$, $\mathsf{CoeffsToSlots}$ and $\mathsf{EvalMod}$. We borrow the notations in [BCKS24] and briefly explain these steps for clarity.

$$\mathbf{z} \xrightarrow{\mathsf{StC}} z(x) \xrightarrow{\mathsf{ModRaise}} z(x) + q_0 I(x) \xrightarrow{\mathsf{CtS}} \mathbf{z} + q_0 \mathbf{I} \xrightarrow{\mathsf{EvalMod}} \mathbf{z}$$

- **SlotsToCoeffs:** The inverse of the canonical embedding evaluates DFT matrix multiplication homomorphically on the encrypted ciphertext that can decrypt to the vector $\mathbf{z}$. This operation converts it into the polynomial form $z(x)$.
- **ModRaise:** The ciphertext at modulus q_0 is expressed in the larger modulus $Q \gg q_0$. This results in a ciphertext that decrypts to $[\langle \text{ct}, \text{sk} \rangle]_Q = z(x) + q_0 I(x)$, where $q_0 I(x)$ is an integer polynomial with coefficients that are small multiples of the base modulus q_0.

The remaining steps of the bootstrapping remove this redundant $q_0 I(x)$ polynomial by homomorphically evaluating a modular reduction by q_0 on $z(x)$.

- **CoeffsToSlots:** The canonical embedding evaluates the iDFT matrix multiplication homomorphically on $z(x) + q_0 I(x)$. To enable parallel (slot-wise) evaluation, the polynomial must be encoded in the slot domain, converting it to a ciphertext that decrypts to $\mathbf{z} + q_0 \mathbf{I}$.
- **EvalMod:** A polynomial approximation of the modular reduction by q_0 is homomorphically evaluated, thus removing the redundant $q_0 \mathbf{I}$ and obtaining the result ciphertext decrypted to $\mathbf{z}$.

2.2 Symmetric-Cipher-to-CKKS Transciphering Methods

Advanced transciphering methods for CKKS often function like a stream cipher [CCF+16], where the plaintext is masked with a keystream (e.g., XORed). The server pre-generates an encrypted keystream homomorphically using the FHE-encrypted secret key, enabling it to decrypt incoming symmetric ciphertext by attaching the keystream to cancel the mask and recover the plaintext under FHE encryption.

The Real-to-Finite-field (RtF) framework [CHK+21] proposed the method of transciphering to BFV and then bootstrapping the BFV ciphertext into CKKS ciphertext. This approach is used with the specially designed symmetric cipher HERA and Rubato, where the ciphertext domain is modular p, matching the plaintext domain of BFV. This alignment is particularly suitable for applications involving numerical computing tasks. However, it is not compatible with standard symmetric ciphers like AES, which may raise additional security concerns on the client side. For example, the 192-bit security version of HERA was broken in [LKSM24], and the non-prime $\mathbb{F}_p$ parameter Rubato was broken by [GAH+23]. In this paper, we focus on the case of Boolean circuit, and thus do not compare with this line of works.

BLEACH methods. We recall the strategy introduced in [DMPS24], which allows Boolean operations using CKKS. In this approach, true and false are represented by 1 and 0, respectively. Utilizing addition, subtraction, and multiplication over real (or complex) numbers, any Boolean gate can be emulated, e.g., XOR, AND, OR, and NOT gates are defined as follows:

$$x \oplus y = (x-y)^2, \quad x \wedge y = x \cdot y, \quad x \vee y = x + y - x \cdot y, \quad \neg x = 1 - x.$$

Except for the free NOT gate, other gates would consume one multiplication depth. Besides, due to the approximate inputs $x + \epsilon_1$ and $y + \epsilon_2$, where $|\epsilon_1|, |\epsilon_2| < 1/4$, the output error is upper bounded by $5\max(|\epsilon_1|, |\epsilon_2|)$ according to [DMPS24, Le. 2]. After multiple sequential operations, this emulated error can grow and must be reduced by the *cleanup* functions, which move values closer to 0 or 1. Specifically, they use the function $h_1(x) = 3x^2 - 2x^3$ from [CKK20] for this cleanup functionality, i.e., $h_1(0 \pm \epsilon) \simeq 0$ and $h_1(1 \pm \epsilon) \simeq 1$.

BLEACH for AES-CKKS transciphering. In [ADE+23], the BLEACH method was utilized to manipulate Boolean bits emulated by real numbers, enhancing the throughput of AES-CKKS transciphering. Since the core of transciphering is establishing the homomorphic circuits for AES, we briefly recall the AES encryption for better understanding. To begin with encryption, the plaintext block is XORed with the 128-bit key in the whitening step. Then, the AES round function is applied to update the resulting 128-bit state. Each round of AES consists of the following operations:

$$\underset{1}{\text{AddRoundKey}} \circ \underset{3}{\text{MixColumns}} \circ \underset{0}{\text{ShiftRows}} \circ \underset{3}{\text{SubBytes}}(\cdot).$$

This round function is repeated 9, 11, or 13 times for AES-128, AES-192, and AES-256, respectively. The MixColumn operation is excluded in the final round. After operating the remaining update function, the resulting state is output as the ciphertext.

AES-CKKS transciphering is ideally suited for the counter (AES-CTR) mode, where multiple AES blocks—each employing a 32-bit nonce and a 96-bit counter as input—are allocated to distinct slots in the CKKS plaintext, thereby enabling Single Instruction Multiple Data (SIMD) operations. This strategy allows each AES block to be processed independently, eliminating the need for costly cross-slot operations. The homomorphic look-up table (LUT) is employed for the AES 8-bit SubBytes operation, requiring 255 multiplications with a depth of three in their implementation, although the theoretical lower bound is $n - \log n - 1 = 247$. It is important to note that in the BLEACH method, the XOR operation, defined as $(x-y)^2$, is not free. The MixColumn and AddRoundKey operations, which involve XOR, consume multiplication depths of 3 and 1, respectively. Additionally, following several Boolean gate operations, the clean function $h_1(x) = 3x^2 - 2x^3$, which has a depth of two, is executed in every AES round. Overall, each AES round requires a multiplication depth of $3 + 3 + 1 + 2 = 9$. Consequently, ten layers of bootstrapping are necessary.

2.3 Revisiting AES-CKKS Transciphering with Details

SubBytes is a key step in AES, but the authors of [ADE+23] just mentioned an indicator-based SubBytes homomorphic LUT method, and neither the algorithm nor the source code was available. To fill this gap, we introduced a homomorphic LUT algorithm that achieved the theoretical lower bound of 247 multiplications.

Let $x_0, x_1, \ldots, x_{\lceil \log n \rceil - 1}$ represent the input bits of a LUT with n entries. The index of a single entry in this table can be expressed as $x = \sum_{i=0}^{\lceil \log n \rceil - 1} x_i \cdot 2^i$. Each bit x_i is an encrypted $\{0, 1\}$ value, with its inverse given by $\overline{x_i} = 1 - x_i$. We have the LUT result written as:

$$\begin{aligned} f(x) = {} & (\overline{x_{\log n-1}} \cdots \overline{x_1 x_0}) \cdot f(0) + (\overline{x_{\log n-1}} \cdots \overline{x_1} x_0) \cdot f(1) \\ & + (\overline{x_{\log n-1}} \cdots \overline{x_2} x_1 \overline{x_0}) \cdot f(2) + (\overline{x_{\log n-1}} \cdots \overline{x_2} x_1 x_0) \cdot f(3) \\ & + \cdots + (x_{\log n-1} \cdots x_1 x_0) \cdot f(n-1), \end{aligned}$$

where $f(x)$ denotes the result of this bijective look-up table. It can be observed that iff the binary representation matches the input index x (denoted as $x \Leftrightarrow (x_0, x_1, \ldots, x_{\lceil \log n \rceil - 1})$), the cumulative product within the parentheses equals one. The LUT result can be obtained without revealing the indices by summing all these values.

Given the definition $\overline{x_i} = 1 - x_i$, we can observe that each output bit of the homomorphic LUT for the S-boxes is the sum of multiple monomials of the input bits. Therefore, once all the monomials are computed, the output bit can be obtained through homomorphic additions. The monomials corresponding to the input bits are as follows:

$$x_0, x_1, \ldots, x_{\log n-1}, x_0 x_1, x_0 x_2, \ldots, \prod_{i=0}^{\log n-1} x_i.$$

We present a recursive algorithm in Algorithm 1 to achieve this lower bound for an 8-bit input with 247 multiplications and a depth of three. This algorithm is later adapted to operate on real FHE ciphertexts. To clarify, we illustrate a toy example of 4 input variables in Fig. 1.

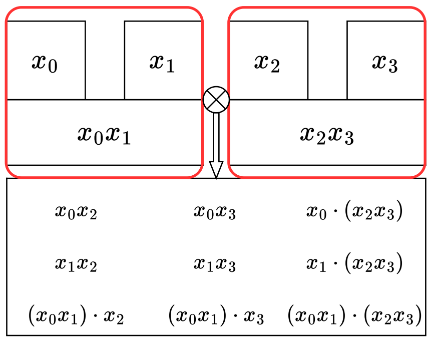

Figure 1: Toy example of Four input variables for EvalRecurse algorithm.

As depicted in Fig. 1, the input variables are divided into 2 sets, each containing two variables. This division corresponds to the deepest layer of the EvalRecurse algorithm. After multiplying each element in the set, the degree-two monomials are merged with the

Algorithm 1: EvalRecurse

```
   Input: Monomial set $\mathcal{M}_{\text{in}}\{x_0, x_1, \ldots, x_{\log n-1}\}$
   Output: Monomial set $\mathcal{M}_{\text{out}}\{x_1, \ldots, x_{\log n}, x_1x_2, x_1x_3, \ldots, \prod_{i=0}^{\log n-1} x_i\}$
1  if $|\mathcal{M}_{\text{in}}| = 2$ then
2      return $\{\mathcal{M}_{\text{in}}[0], \mathcal{M}_{\text{in}}[1], \mathsf{Mult}(\mathcal{M}_{\text{in}}[0], \mathcal{M}_{\text{in}}[1])\}$
3  $idx \leftarrow \frac{|\mathcal{M}_{\text{in}}|}{2}$
4  $\mathcal{M}_L \leftarrow \mathsf{EvalRecurse}(\mathcal{M}_{\text{in}}[:idx])$
5  $\mathcal{M}_R \leftarrow \mathsf{EvalRecurse}(\mathcal{M}_{\text{in}}[idx:])$
6  $\mathcal{M}_{\text{out}} \leftarrow \{\}$
7  for $\mathsf{mon}_i \in \mathcal{M}_L$ do
8      for $\mathsf{mon}_j \in \mathcal{M}_R$ do
9          $\mathsf{mon}_{\text{new}} \leftarrow \mathsf{Mult}(\mathsf{mon}_i, \mathsf{mon}_j)$
10         $\mathcal{M}_{\text{out}} \leftarrow \mathcal{M}_{\text{out}} \cup \mathsf{mon}_{\text{new}}$
11 $\mathcal{M}_{\text{out}} \leftarrow \mathcal{M}_{\text{out}} \cup \mathcal{M}_L \cup \mathcal{M}_R$
12 return $\mathcal{M}_{\text{out}}$
```

original variables and sent to the next recursion. Finally, each element in the set obtained at level one is multiplied, merging with the set obtained in the last recursion. The total multiplications consumed are $n - \log n - 1 = 11$ with depth 2.

To combine all the monomials into the AES S-box output bit, we define the Boolean function in Definition 1 as follows and explain the detailed AES SubBytes operation in Sect. 4.

Definition 1. Boolean Function: Let $f : \mathbb{F}_2^n \to \mathbb{F}_2$ be a Boolean function with intermediate coefficients in $\mathbb{Z}$:

$$f(\mathbf{x}) = f(x_0, x_1, \ldots, x_{n-1}) = \sum_{\mathbf{u} \in \mathbb{F}_2^n} c_{\mathbf{u}} \prod_{i=0}^{n-1} x_i^{u_i},$$

where $c_{\mathbf{u}} \in \mathbb{Z}$, and

$$\mathbf{x}^{\mathbf{u}} = \prod_{i=0}^{n-1} x_i^{u_i} \quad \text{with} \quad x_i^{u_i} = \begin{cases} x_i, & \text{if } u_i = 1, \\ 1, & \text{if } u_i = 0. \end{cases}$$

denotes the monomials that comprise the polynomial of the output bit.

3 The Free-XOR Technique

At a high level, our idea is based on two key observations: (i) The XOR of multiple binary elements can be translated into the least significant bit (LSB) extraction, which is equivalent to performing a modular reduction by 2 on their aggregated sum. (ii) Since CKKS operations are defined over the ciphertext modulus q, any overflows beyond q are automatically eliminated.

In this section, we introduce our strategy that automatically handles the overflows and achieves Free-XOR.

3.1 Automated Overflow in CKKS

We recall the bootstrapping procedure defined in Sect. 2.1. In this process, the ciphertext, which decrypts to a scaled message $\mathbf{z} + q_0\mathbf{I}$, is processed during the EvalMod step. A key aspect of bootstrapping is the homomorphic removal of the redundant term $q_0\mathbf{I}$, which involves the computationally intensive task of performing homomorphic modular reduction by q_0.

By observing overflow behavior when the encrypted message exceeds the modulus q, we propose a framework that initially accumulates Boolean values into an integer and then applies modular reduction by retaining only the LSB through the overflow. This approach integrates seamlessly into the bootstrapping procedure, occurring naturally during the SlotsToCoeffs operations. The overflow mechanism leverages the inherent modular q operation, adding no significant complexity to the original bootstrapping process and thus providing a free functional enhancement.

Modulus and scale management of SlotsToCoeffs **for automated overflow.** Consider a scenario where τ consecutive XOR operations are performed, resulting in a large integer within the range $[0, \tau]$. To handle values that exceed the modulus capacity, we shift the least significant bit (LSB) to the most significant bit (MSB). This operation involves multiplying ciphertexts by constant plaintext values, which is most efficiently achieved by merging these values and pre-multiplying the resulting constants with the DFT matrices, as suggested by [BMTH21].

Specifically, we employ the SlotsToCoeffs-first bootstrapping approach, which requires starting the process with a modulus larger than q_0. Assume the transformation matrix for DFT is factorized into deg matrices and the difference in the scaling factor between the shifted ciphertexts and the current one is given by $\delta_{\mathsf{diff}} = q_0/2\Delta_{\mathsf{StC}}$, where the ciphertext at the StC level has a scale of Δ_{StC}. Consequently, each factorized matrix is scaled by $\mu_{\mathsf{StC}} = (\delta_{\mathsf{diff}})^{\frac{1}{\mathsf{deg}}}$. Then, we integrate the automated overflow into SlotsToCoeffs as follows.

Starting with the input ciphertext at the StC level, which decrypts to

$$\mathbf{z} = \Delta_{\mathsf{StC}}\mathbf{m} \mod Q_{\mathsf{StC}}, \quad \text{where} \quad Q_{\mathsf{StC}} = q_0 \cdots q_{\mathsf{deg}},$$

we scale each factorized matrix by the factor μ_{StC}, such that $M_i' \leftarrow \mu_{\mathsf{StC}} \times M_i$. The encoded DFT submatrices $\widetilde{M_1}, \ldots, \widetilde{M_{\mathsf{deg}}}$, with $\widetilde{M_i} = \mathsf{Ecd}(M_i', q_{i+1}, N)$, are then applied as follows:

$$\begin{aligned} \mathsf{ct}_{\mathsf{deg}-1} \leftarrow \mathsf{RS}_{\Delta_{\mathsf{deg}}}(\widetilde{M_{\mathsf{deg}}} \cdot \mathsf{ct}_{\mathsf{deg}}) \quad &\text{with} \quad \mathsf{ct}_{\mathsf{deg}-1} = \mathrm{Enc}_{\mathsf{sk}}\left(\Delta_{\mathsf{deg}-1}(\mu_{\mathsf{StC}} \cdot \mathbf{m}')\right), \\ &\vdots \\ \mathsf{ct}_0 \leftarrow \mathsf{RS}_{\Delta_1}(\widetilde{M_1} \cdot \mathsf{ct}_1) \quad &\text{with} \quad \mathsf{ct}_0 = \mathrm{Enc}_{\mathsf{sk}}\left(\Delta_0 \cdot m(x)\right). \end{aligned}$$

This scale management enables homomorphic decoding without sacrificing the precision of SlotsToCoeffs, as summarized by the following formula:

$$\mathbf{z} = \Delta_{\mathsf{StC}}\mathbf{m} \mod Q_{\mathsf{StC}} \xrightarrow{\mathsf{SlotsToCoeffs}} \Delta_0 b(x) = \Delta_0 m(x) \mod q_0.$$

After this operation, the message is transformed into its coefficient form, and the modulus is reduced to the lowest level. Each coefficient is observed to decompose into two parts: the binary part and the even-number part, as defined by

$$\Delta_0 m(x) \mod q_0 = \Delta_0(b(x) + 2I'(x)) \mod q_0,$$

where $I'(x)$ is a polynomial with integer coefficients bounded by $\lceil \frac{\tau-1}{2} \rceil$, and $b(x)$ is a polynomial with coefficients in $\{0,1\}^N$. Since the LSB has been shifted to the MSB (i.e., $\Delta_0 = q_0/2$), we obtain

$$\frac{q_0}{2} \cdot b(x) \mod q_0 \leftarrow \frac{q_0}{2} \cdot b(x) + \frac{q_0}{2} \cdot 2I'(x) \mod q_0.$$

This reduction occurs automatically because the redundant polynomial $I'(x)$ has coefficients that are multiples of the base modulus q_0. In other words, this overflow happens during the SlotsToCoeffs operation by utilizing the scale management technique. Subsequently, the bootstrapping procedures are applied in a manner similar to Boolean bootstrapping [BCKS24], as follows:

$$\frac{q_0}{2}b(x) \xrightarrow{\mathsf{ModRaise}} \frac{q_0}{2}b(x) + q_0 I(x) \xrightarrow{\mathsf{CoeffsToSlots}} q_0\left(\frac{1}{2}\mathbf{b} + \mathbf{I}\right).$$

After applying the approximated trigonometric function $\frac{1}{2}(1-\cos(2\pi x))$ with the clean functionality, we can remove the redundant term $\mathbf{I}$. The base prime q_0 serves as the scale factor during the EvalMod phase and is often adjusted to match the modulus at this level.

Other methods on modular reduction in CKKS. A very recent work by Kim et al. [KN24] explores the general case of modular reduction in CKKS. Since their work focuses on computation over integers, they suggest to multiply an encrypted value of scale q_0/t^l by t^{l-1} after SlotsToCoeffs to extract its LSB. However, this approach increases the noise by t^{l-1} times and might not be suitable for Boolean circuits. Take transciphering as an example, if an AES-encrypted bit serves as the Most Significant Bit (MSB) of an input of the subsequent privacy-preserving application, even minor noise on that bit could result in large errors in the function's output.

3.2 XBOOT: Automated Overflow Combined with Bootstrapping

Algorithm 2: XBOOT

```
   Input: ct=$\mathrm{Enc}_{\mathsf{sk}}(\Delta_{\mathsf{StC}}\mathbf{m}+\epsilon)$ with $\|\epsilon\|_\infty \ll 1$, where $\mathsf{ct} \in \mathcal{R}^2_{Q_{\mathsf{StC}}}$
   Output: $\mathsf{ct}_{\mathrm{out}} \in \mathcal{R}^2_{Q_{\mathrm{rem}}}$
1  Setting: $\delta_{\mathrm{diff}} = q_0/2\Delta_{\mathsf{StC}}$
2  $\mathsf{ct}_{\mathrm{bin}} \leftarrow \mathsf{SlotsToCoeffs}(\mathsf{ct})$// Automated overflow.
3  $\mathsf{ct}' \leftarrow \mathsf{CoeffsToSlots} \circ \mathsf{ModRaise}(\mathsf{ct}_{\mathrm{bin}})$
4  $\mathsf{ct}_{\mathrm{real}} \leftarrow (\mathsf{conj}(\mathsf{ct}') + \mathsf{ct}')/2$
5  $\mathsf{ct}_{\mathrm{out}} \leftarrow \mathsf{EvalMod}_{f_{\mathsf{Bin}}}(\mathsf{ct}_{\mathrm{real}})$ // $f_{\mathsf{Bin}} = \frac{1}{2}(1-\cos(2\pi x))$.
6  return $\mathsf{ct}_{\mathrm{out}}$
```

We describe a combination of automated overflow and binary bootstrapping in Algorithm 2. It takes as input a ciphertext $\mathsf{ct} = \mathrm{Enc}_{\mathsf{sk}}(\Delta_{\mathsf{StC}}\mathbf{m}+\epsilon)$, which decrypts to a vector of integers. After applying homomorphic decoding with automated overflow in line 2, we obtain $\mathsf{ct}_{\mathrm{bin}} = \mathrm{Enc}_{\mathsf{sk}}\left(\frac{b(x)}{2} + \epsilon'(x)\right)$, where the message's LSB is automatically shifted to the MSB of the base modulus such that

$$\frac{q_0}{2}b(x) \leftarrow \frac{q_0}{2}b(x) + q_0 I'(x) \mod q_0.$$

After line 3, we have $\mathsf{ct}' = \mathrm{Enc}_{\mathsf{sk}}\left(\frac{\mathbf{b}+\epsilon_c}{2} + \mathbf{I}\right)$, where $\mathbf{I} \in \mathbb{Z}^{N/2}$ is an integer vector of small magnitude and $\epsilon_c \in \mathbb{C}^{N/2}$ represents noise related to the original ϵ' and the homomorphic operations. In line 4, the real part of ct' is extracted, yielding $\mathsf{ct}_{\mathrm{real}} = \mathrm{Enc}_{\mathsf{sk}}\left(\frac{\mathbf{b}+\epsilon_r}{2} + \mathbf{I}\right)$, where $\epsilon_r \in \mathbb{R}^{N/2}$.

Finally, after the homomorphic evaluation of the polynomial approximation of $f_{\mathsf{Bin}} = \frac{1}{2}(1-\cos(2\pi x))$, we obtain $\mathsf{ct}_{\mathrm{out}} = \mathrm{Enc}_{\mathsf{sk}}(\mathbf{b}+\epsilon_{\mathrm{out}})$ at a higher modulus Q_{rem}. We omit the noise analysis as it is similar to the noise estimation in the original binary bootstrapping [BCKS24, Thm 1].

Reduced interpolation interval. In XBOOT, a sparser secret key and a lower ring degree are utilized, allowing us to reduce the interpolation interval without compromising the failure probability of bootstrapping. We describe the rationale behind as follows.

Recall that each coefficient of the redundant polynomial after the ModRaise step follows the shifted Irwin–Hall distribution [LLL+21] over the interval $[-1/2, 1/2]$. The failure probability can be calculated using the adapted cumulative distribution function (CDF) corresponding to the binomial distribution for N trials, as specified in Bossuat et al. [BMTH21, Eq. 1].

Table 1: Adapted K of $\Pr[\|I(x)\|_\infty > K] \approx 2^{-16}$ with different Hamming distance h.

$\log N$	15									
$\log(h)$	5	6	7	8	9	10	11	12	13	14
K	10	14	20	28	40	56	80	113	160	226
$\log(\Pr[\|I(x)\|_\infty > K])$	-17.8	-15.6	-15.6	-14.7	-15.2	-14.5	-15.1	-15.0	-15.0	-15.0
$K/\sqrt{h}$	1.77	1.75	1.77	1.75	1.77	1.75	1.77	1.77	1.77	1.77

$\log N$	14									
$\log(h)$	4	5	6	7	8	9	10	11	12	13
K	7	10	14	20	28	39	56	79	111	158
$\log(\Pr[\|I(x)\|_\infty > K])$	-23.4	-18.8	-16.6	-16.6	-15.7	-14.8	-15.5	-15.4	-15.0	-15.3
$K/\sqrt{h}$	1.75	1.77	1.75	1.77	1.75	1.72	1.75	1.75	1.73	1.75

A lower ring degree could lead to a reduced interpolation interval due to fewer trials in the binomial distribution. We revisit the distribution to compute the adapted K values, as shown in Table 1.

Additionally, [BTH22] suggests adopting a sparse secret key to reduce the complexity of the EvalMod step. This approach does not compromise security, as the sparse operates at a smaller modulus. XBOOT can significantly benefit from this technique, especially given its smaller base modulus. When combined with a smaller ring degree ($\log N \leq 15$), it results in a reduced interpolation interval K, thereby enhancing bootstrapping efficiency.

3.3 Possible optimizations and limitations

Lazier reduction. The previous sections focused on the specific scenario where bootstrapping is applied immediately after the XOR gates. However, it is important to note that XBOOT is also applicable in cases where XOR and AND gates are interleaved before bootstrapping. Consider a toy example of two values x, y, each results from k instances of dense XOR operations on Boolean values. Then an AND operation is applied to these two values. We can deduce the following:

$$\left(x = \bigoplus_{i=0}^{k} x_i\right) \wedge \left(y = \bigoplus_{i=0}^{k} y_i\right) \equiv \mathrm{LSB}\left(\left(x = \sum_{i=0}^{k} x_i\right) \times \left(y = \sum_{i=0}^{k} y_i\right)\right).$$

This equivalence holds because

$$x \wedge y = 1 \iff \mathrm{LSB}\left(\sum_{i=0}^{k} x_i\right) = 1 \text{ and } \mathrm{LSB}\left(\sum_{i=0}^{k} y_i\right) = 1,$$

Therefore, we do not need to insert XBOOT immediately after the XOR gates; instead, we can be lazier and insert XBOOT after the AND gate. Similar formulas would apply to any Boolean circuit, where we can replace all XOR gates in the circuit with additions, analyze the dependencies of the computation graph, and strategically adjust the placement of XBOOT without changing the original circuit's topology.

Limitations. The BLEACH method of cleaning noise using $h_1(x) = 3x^2 - 2x^3$ only works with Boolean values and is incompatible with integer values. Consequently, we can only rely on bootstrapping for noise cleaning. In very rare cases, such as circuits primarily composed of AND gates with only a few XOR gates, our framework may not outperform BLEACH. However, for most practical scenarios where the number of XOR gates is higher or comparable to that of AND gates, our framework consistently performs better.

4 Application to AES Transciphering

XBOOT immediately increases the efficiency for CKKS computations on Boolean circuits. Consider the well-known AES transciphering as an example: the AES round function consists of four key steps—SubBytes, ShiftRows, MixColumns, and AddRoundKey. The best previous work on AES-CKKS transciphering [ADE+23] requires 9 multiplication depths plus one bootstrapping per round. We can transform the XOR operations in MixColumns and AddRoundKey into inexpensive homomorphic additions, followed by XBOOT. This transformation reduces the cost per round to 3 multiplication depths plus one bootstrapping. We describe the details in the next subsections.

4.1 Details of SubBytes Operation

Algorithm 3: SubBytes (Homomorphic LUT)

```
Input: Encrypted bits $\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_7$ with each bit $\mathbf{x}_i \in \{0+\epsilon, 1+\epsilon\}^{N/2}$
Output: sbox as the LUT output such that $\{\mathbf{y}_0, \mathbf{y}_1, ..., \mathbf{y}_7\} = \text{S-box}(\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_7)$

   // Obtain the monomial table for each AES S-box output bit.
1  for $i = 0$ to 7 do
2      $T_i \leftarrow$ AES S-box LUT

   // Compute all required monomials homomorphically.
3  $\mathcal{M} \leftarrow \mathsf{EvalRecurse}(\mathbf{x}_0, ..., \mathbf{x}_7)$ // $|\mathcal{M}| = 255$

   // Initialize each bit with the constant term of the monomial.
4  $\mathbf{sbox}[i] \leftarrow \{\mathsf{Ecd}(\mathbf{const}_i)\}$ for $i \in \{0, 1, .., 7\}$
5  for $i = 0$ to 7 do
       // Gather coefficients for each output bit
6      for $j = 1$ to 255 do
7          if $T_i[j] \neq 0$ then
8              mon $\leftarrow \mathsf{cMult}(\mathcal{M}[j], T_i[j])$ // mon is of the form $c_{\mathbf{u}} \cdot (\prod_{i=0}^{7} \mathbf{x}_i^{u_i})$
9          $\mathbf{sbox}[i] \leftarrow \mathsf{Add}(\mathbf{sbox}[i], \mathsf{mon})$
10 return sbox
```

As described in Section 2.3, we employ a recursive method in Algorithm 1 to obtain all the monomials required for the AES S-box LUT using 247 multiplications with a depth of 3. To achieve the final result of the SubBytes, we utilize a strategy that combines all the single terms of monomials into the Boolean function in Definition 1 for each S-box's output bit. The procedure is detailed in Algorithm 3 and described as follows.

The input to the SubBytes operation consists of encrypted 8-bit ciphertexts. Each ciphertext decrypts to a Boolean vector $\mathbf{x}_i$, where each element represents a specific bit of the state from $N/2$ different blocks. These blocks are processed using SIMD operations, resulting in the encrypted 8-bit output $\{\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_7\}$.

- **Preprocess phase:** As shown in lines 1–2, we precompute the coefficients of the Boolean function for each output bit of the AES S-box and store them in the plaintext table T_i. This procedure involves applying the AES S-box LUT to the input variable and storing the result in the coefficient table.

- **Monomial making:** This procedure is the computational bottleneck, requiring 247 multiplications with a depth consumption of 3. After applying the EvalRecurse function homomorphically, we obtain all monomials in the set $\mathcal{M}$.

- **Boolean function:** We consider each monomial in the obtained set $\mathbf{x}^{\mathbf{u}} \in \mathcal{M}$ as defined in Definition 1. This procedure involves combining these monomials with corresponding coefficients and summing them together. Specifically, we apply a scalar multiplication cMult to each monomial $\mathcal{M}[j]$ using the constants from the pre-computed coefficient table T_i, corresponding to the AES S-box. By summing all these coefficient-adjusted monomials mon, we derive the Boolean function for a specific output bit.

Note that the **Boolean function** phase of our SubBytes operation is almost computationally free, as cMult with integer inputs is processed similarly as multiple additions of the ciphertext itself. This implementation is more efficient and software-friendly because the coefficients in the table T_i can include large numbers in $\mathbb{Z}$, eliminating the need for numerous homomorphic addition operations that would otherwise be required.

4.2 Free XOR AES Evaluation

We compare our free XOR procedure with the original procedure in Fig. 2 and provide a detailed implementation in Algorithm 4, which is described below.

As shown in Fig. 2, the newly proposed free XOR AES evaluation uses the same input as the original one. Specifically, each 128-bit AES block takes a 32-bit nonce and a 96-bit counter as input. The vector of these blocks is denoted as $\mathbf{S}$, where each element $\mathbf{S}[i] = \mathbf{B} = \{B_0, B_1, \ldots, B_{\frac{N}{2}}\}$ for $0 \leq i < 128$. This notation can also be expressed by gathering $\frac{N}{2}$ nonces and counters as $\mathbf{S} = \{n \| c\}^{N/2}$. During the encryption process, the currently processed cipher block is also called the cipher's state.

After applying $\mathsf{Ecd}(\cdot)$ to the batched state bits one by one, we obtain 128 plaintext polynomials in $\mathcal{R}_N$, storing all $N/2$ blocks of AES-CTR input values, with each slot containing a noised Boolean value. First, we perform the AddWhiteKey operation by using the XOR operation on the obtained plaintext polynomial $\mathbf{S}$ with the duplicated encrypted symmetric key $\mathbf{K}_0$ provided by the client. This symmetric key, encrypted under sk, is denoted as $\mathbf{K}_0 = \{\underbrace{\mathrm{k}_0, \mathrm{k}_0, \ldots, \mathrm{k}_0}_{N/2 \text{ times}}\}$, which represents $N/2$ repetitions of the symmetric key.

Afterwards, we apply the 9-round AES free XOR function to update the state.

- **SubBytes:** As shown in Algorithm 3, the input consists of encrypted bits, and the output is the encrypted 8-bit AES S-box look-up table (LUT) result.

- **ShiftRows:** Each bit of the state is encrypted in a different FHE ciphertext, which only involves changing the indices of the ciphertext.

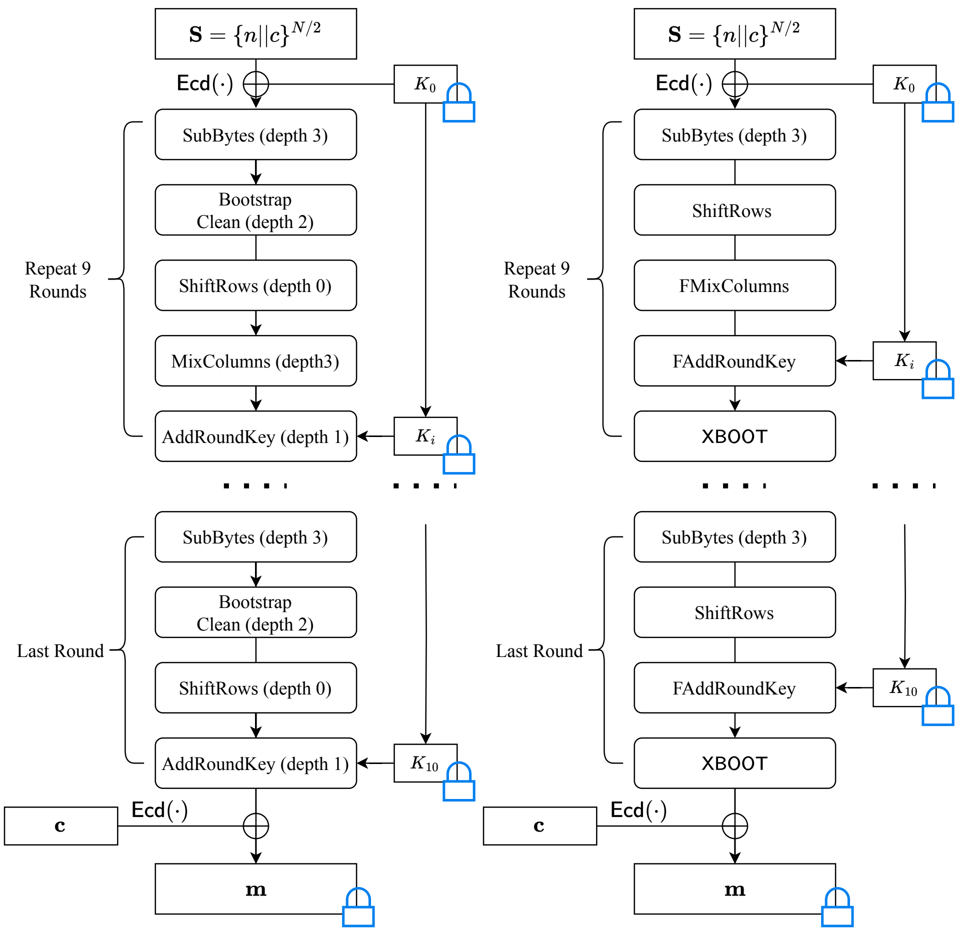

Figure 2: Original AES evaluation procedure (left) and our free XOR AES evaluation procedure (right).

- **FMixColumns:** Let each column of the state as $[b_0, b_1, b_2, b_3]^\top$. According to [FRL19], we have the updated column as:

$$
\begin{aligned}
\mathrm{b}'_0 &= x \cdot (b_0 + b_1) + b_1 + b_2 + b_3 \\
\mathrm{b}'_1 &= x \cdot (b_1 + b_2) + b_2 + b_3 + b_0 \\
\mathrm{b}'_2 &= x \cdot (b_2 + b_3) + b_3 + b_0 + b_1 \\
\mathrm{b}'_3 &= x \cdot (b_3 + b_0) + b_0 + b_1 + b_2.
\end{aligned}
$$

Let $(a_7, a_6, ..., a_0)$ denote the 8 bits of the byte. Multiply by x in GF(2^8) is

$$(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0) = (a_6, a_5, a_4, a_3 + a_7, a_2 + a_7, a_0 + a_7, a_7).$$

In FMixColumns, the $+$ operation is translated into homomorphic addition, resulting in a small integer in each slot of the FHE ciphertext.

- **FAddRoundKey:** 128 paralleled homomorphic additions are applied to the state.

Algorithm 4: Homomorphic AES Evaluation

```
   Input: $\mathbf{S} = \{n||c\}^{N/2}$, where each block takes a 32-bit nonce and a 96-bit counter.
   Input: $\mathbf{K}_i$, where $0 \leq i \leq 10$ are the round keys duplicated for $N/2$ times.
   Output: $\mathsf{ct}_{\text{out}} = \text{Enc}_{\mathsf{sk}}(\mathbf{keystream} + \epsilon_{tr})$, FHE encrypted keystreams.
1  $\mathbf{S} \leftarrow \mathsf{Ecd}(\mathbf{n}||\mathbf{c})$
2  $\mathbf{state} \leftarrow \mathsf{AddWhiteKey}(\mathbf{S}, \mathbf{K}_0)$// Apply whitening key with XOR
3  for $i = 1$ to $9$ do
4      for $j = 0$ to $15$ do
5          $\mathbf{state}[8*j : 8*(j+1)] \leftarrow \mathsf{SubBytes}(\mathbf{state}[8*j : 8*(j+1)])$
6      $\mathbf{state} \leftarrow \mathsf{ShiftRows}(\mathbf{state})$
7      for $j = 0$ to $3$ do
           // Apply homomorphic addition instead of XOR
8          $\mathbf{col} \leftarrow \mathsf{FMixColumns}(\mathbf{state}[32*j : 32*(j+1)])$
9          $\mathbf{state}[32*j : 32*(j+1)] \leftarrow \mathbf{col}$
10     $\mathbf{state} \leftarrow \mathsf{FAddRoundKey}(\mathbf{state}, \mathbf{K}_i)$
11     $\mathbf{state} \leftarrow \mathsf{XBOOT}(\mathbf{state})$
   // Last Round
12 for $i = 0$ to $15$ do
13     $\mathbf{state}[8*i : 8*(i+1)] \leftarrow \mathsf{SubBytes}(\mathbf{state}[8*i : 8*(i+1)])$
14 $\mathbf{state} \leftarrow \mathsf{ShiftRows}(\mathbf{state})$
15 $\mathbf{state} \leftarrow \mathsf{FAddRoundKey}(\mathbf{state}, \mathbf{K}_{10})$
16 $\mathsf{ct}_{\text{out}} \leftarrow \mathsf{XBOOT}(\mathbf{state})$
17 return $\mathsf{ct}_{\text{out}}$
```

- **XBOOT:** During StC, the bounded small integer in each slot is transformed into a Boolean value through automated overflow. This operation includes bootstrapping, ensuring clean functionality.

By repeating the round function described above for 9 rounds and then applying a final round that performs the same operations except for **FMixColumns**, one can obtain $N/2$ blocks of the keystream, denoted as ct_{out} in Algorithm 4. It is important to note that this procedure is independent of the messages and is considered part of the offline phase. After homomorphically XORing with the encoded $N/2$ blocks of the symmetric ciphertext $\mathsf{Ecd}(\mathbf{c})$, the FHE-encrypted messages can be obtained.

4.3 Further Optimization

In each AES round function, 128 FHE ciphertexts are updated with each operation defined above, resulting in 128 parallel XBOOT calls, which can be effectively multi-threaded.

We pack two ciphertexts that encrypt real numbers into a single complex number ciphertext to reduce the XBOOT calls. After CoeffsToSlots, both the real and imaginary parts of the ciphertext are extracted. We then apply EvalMod to both parts, obtaining two ciphertexts from a single BatchXBOOT operation in Algorithm 5. As a result, we can decrease the 128 XBOOT operations required in one AES round to 64 BatchXBOOT operations.

Algorithm 5: BatchXBOOT

```
    Input: $\mathsf{ct}_0, \mathsf{ct}_1 \in \mathcal{R}^2_{Q_{\mathsf{StC}}}$ that decrypts to $(\mathbf{m_0} + \epsilon_\mathbf{0}), (\mathbf{m_1} + \epsilon_\mathbf{1})$ with $\epsilon_\mathbf{0}, \epsilon_\mathbf{1} \in \mathbb{R}^{N/2}$
    Output: $\mathsf{ct}_0^{\mathrm{out}}, \mathsf{ct}_1^{\mathrm{out}} \in \mathcal{R}^2_{Q_{\mathrm{rem}}}$
1   Setting: $\delta_{\mathrm{diff}} = q_0/2\Delta_{\mathsf{StC}}$
2   $\mathsf{ct} = \mathsf{ct}_0 + \mathsf{cMult}(\mathsf{ct}_1, 1i)$ // $\mathsf{ct} = \mathtt{Enc}_{\mathsf{sk}}((\mathbf{m_0} + i\mathbf{m_1}) + \epsilon')$
3   $\mathsf{ct}_{\mathrm{bin}} \leftarrow \mathsf{SlotsToCoeffs}(\mathsf{ct})$
4   $\mathsf{ct}' \leftarrow \mathsf{CoeffsToSlots} \circ \mathsf{ModRaise}(\mathsf{ct}_{\mathrm{bin}})$
5   $\mathsf{ct}_0^{\mathrm{real}} \leftarrow (\mathsf{conj}(\mathsf{ct}') + \mathsf{ct}')/2$
6   $\mathsf{ct}'' = \mathsf{cMult}(\mathsf{ct}', -1i)$ // $\mathsf{ct}'' = \mathtt{Enc}_{\mathsf{sk}}((\mathbf{b_1} - i\mathbf{b_0} + \epsilon'')/2 + \mathbf{I})$
7   $\mathsf{ct}_1^{\mathrm{real}} \leftarrow (\mathsf{conj}(\mathsf{ct}'') + \mathsf{ct}'')/2$
8   $\mathsf{ct}_0^{\mathrm{out}} \leftarrow \mathsf{EvalMod}_{f_{\mathsf{Bin}}}(\mathsf{ct}_0^{\mathrm{real}})$
9   $\mathsf{ct}_1^{\mathrm{out}} \leftarrow \mathsf{EvalMod}_{f_{\mathsf{Bin}}}(\mathsf{ct}_1^{\mathrm{real}})$
10  return $\mathsf{ct}_0^{\mathrm{out}}, \mathsf{ct}_1^{\mathrm{out}}$
```

5 Unlocking New Potentials: Application to Rasta

XBOOT also unlocks new possibilities for CKKS, particularly in scenarios where the number of XOR gates is prohibitively large. For example, several symmetric ciphers have been defined over $\mathbb{Z}_2$ including LowMC [DEM15], Rasta [DEG+18], Dasta [HL20], and Fasta [CIR22]. What these ciphers have in common is their extensive use of XOR gates, which presents significant challenges for their evaluation under CKKS. The designers of these ciphers did not regard this as an issue, as they were primarily designed with BGV or MPC transciphering in mind, where XOR gates are relatively inexpensive. After all, BLEACH's idea of CKKS computation on Boolean circuits had not been proposed at that time.

Since BLEACH's idea is quite new, applications (beyond transciphering) for utilizing CKKS on Boolean circuits are relatively scarce. In this section, we still focus on transciphering and use Rasta as an example to demonstrate how XBOOT could make the difference.

The incompatibility between Rasta and BLEACH. Rasta incorporate randomly generated linear matrices within its linear layers to resist statistical attacks. This approach is motivated by the observation that statistical attacks require numerous queries to the same encryption function to gather sufficient data for formulating a statistical model.

Let $A \in \mathbb{R}^{N/2}_{n \times n}$ represent $\frac{N}{2}$ randomly generated $n \times n$ square matrices. When evaluating matrix multiplication, we consider one column of the matrix $[a_0, a_1, \ldots, a_n]^\top$. The updated column is then given by:

$$y_0 = \bigoplus_{i=0}^{n-1}(a_0 \wedge x_0), \quad y_1 = \bigoplus_{i=0}^{n-1}(a_1 \wedge x_1), \quad \cdots, \quad y_{n-1} = \bigoplus_{i=0}^{n-1}(a_{n-1} \wedge x_{n-1}),$$

where y_i and x_i denote the updated element and input element of a column, respectively.

As shown, each update incurs a substantial number of AND and XOR operations. The AND operations are applied to ciphertexts x and plaintexts a, thus are inexpensive. In contrast, the XOR operations involve ciphertext-ciphertext interactions. In BLEACH, the XOR function is represented as $(x - y)^2$, which consumes one multiplication. A 351-bit state in Rasta would require $351 \times (351 - 1) = 122,850$ homomorphic multiplications for updating a single element of the column if we use BLEACH.

Application of XBOOT to Rasta. We briefly revisit Rasta's keystream generation in Fig. 3. Note that other FHE-friendly $\mathbb{Z}_2$ ciphers that include matrices operations also benefit from XBOOT.

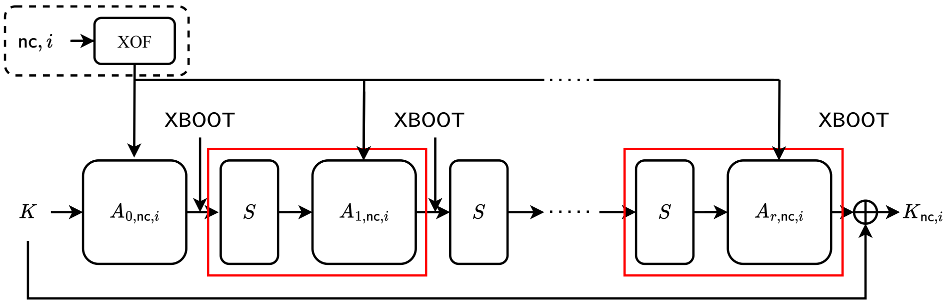

Figure 3: Rasta keystream generate function. $S, A_{r,\mathsf{nc},i}$ denote the nonlinear function and the randomized matrix, respectively.

Rasta employs the randomly generated permutation $P_{\mathsf{nc},i}$ on the symmetric key K and XOR the same K with the output of the permutation to output keystream $K_{\mathsf{nc},i} = K \oplus P_{\mathsf{nc},i}$. The permutation is defined as:

$$P_{\mathsf{nc},i} = A_{r,\mathsf{nc},i} \circ S \circ A_{r-1,\mathsf{nc},i} \circ \cdots \circ S \circ A_{1,\mathsf{nc},i} \circ S \circ A_{0,\mathsf{nc},i}(K),$$

where r denotes the number of rounds based on the cipher version. Each matrix in $A_{r,\mathsf{nc},i}$ is generated by an extendable-output function (XOF) [PF15], depending on the public nonce nc and the block counter i. The affine layer is defined as $A_{r,\mathsf{nc},i} = M_{r,\mathsf{nc},i} \cdot \mathbf{x} + c_{r,\mathsf{nc},i}$, which involves a binary multiplication of the random matrix $M_{r,\mathsf{nc},i}$ followed by the addition of a round constant $c_{r,\mathsf{nc},i}$. The non-linear χ function layer S is defined as $y_i = x_i + x_{i+2} + (x_{i+1} \wedge x_{i+2})$, where $0 \leq i < n$, with indices i taken modulo n. We refer [DEG+18] for details.

In our implementation, we also consider the same data packing method as we did in AES-CKKS transciphering, with each bit of the Rasta state encoded in different FHE ciphertexts. This enables the simultaneous evaluation of $\frac{N}{2}$ cipher blocks. Each bit of the state is updated with S layers, involving one multiplication with a depth of one and producing a small integer bounded by three. This efficiency gain benefits from the interleaved XOR and AND operations in free XOR.

When evaluating the affine layer $A_{r,\mathsf{nc},i} = M_{r,\mathsf{nc},i} \cdot \mathbf{x} + c_{r,\mathsf{nc},i}$, all $\frac{N}{2}$ random matrices are encoded into $(n \times n)$ FHE plaintexts, while the round constants are encoded into n plaintexts. One column of the matrices is updated as

$$\mathbf{y}_0 = \sum_{i=0}^{n-1} (\mathbf{a}_0 \wedge \mathbf{x}_0), \quad \mathbf{y}_1 = \sum_{i=0}^{n-1} (\mathbf{a}_1 \wedge \mathbf{x}_1), \quad \cdots, \quad \mathbf{y}_{n-1} = \sum_{i=0}^{n-1} (\mathbf{a}_{n-1} \wedge \mathbf{x}_{n-1}),$$

where $[\mathbf{a}_0, \mathbf{a}_1, \ldots, \mathbf{a}_{n-1}]^\top$ represents one column of the batched random matrices, with each element $\mathbf{a}_i = \mathsf{Ecd}(a_0^i, a_1^i, \ldots, a_{N/2}^i)$ for $0 \leq i < n$. After the RS operation, with a level consumption of one, the batched round constant is added to the state. By applying XBOOT to each FHE ciphertext, the round function can be iterated until the keystream is output.

With XBOOT, XOR operations are transformed into homomorphic additions, resulting in a noised integer in each slot of the FHE ciphertext, bounded by $\tau = (n+1) \times 3$ (where the result of each S layer is bounded by three). After applying XBOOT, these integers are converted into Boolean values, allowing other operations within the round function to proceed efficiently. This approach significantly improves performance by leveraging the SIMD capabilities of the CKKS scheme, with a bootstrapping layer consumption of $r + 1$. Each bootstrapping layer contains $\lceil \frac{n}{2} \rceil$ calls to BatchXBOOT.

6 Experimental Results

We implemented XBOOT using the open-source homomorphic encryption library Lattigo V6 [lat24] and conducted experiments based on this framework. All experiments were performed on a server equipped with an Intel Xeon Platinum 8369B 64-Core 2.50 GHz machine with 128 GB RAM.

In the experiments, N denotes the ring degree of the bootstrapping parameter. The symbols h and $\tilde{h}$ represent the Hamming weights of the dense and sparse secret keys, respectively, as described in the sparse secret encapsulation from [BTH22]. $\log(QP)$ denotes the maximum modulus used in the key-switching process, *dnum* indicates the gadget rank of the gadget decomposition, and *depth* represents the multiplication levels available after bootstrapping. All parameters used in the experiments ensure 128-bit security, as referenced in [APS15].

6.1 FHE Parameters

Similar to [BCKS24], our XBOOT benefits from modulus savings due to the reduced gap between the base modulus q_0 and the scale Δ_0. Moreover, the novel proposal of its free XOR functionality further reduces the multiplication depths. As a result, we can decrease the ring degree N to 2^{15} with a total modulus $\log(QP) \leq 830$-bit, ensuring 128-bit security. And we switch between different Hamming weights for bootstrapping as suggested in [BTH22]. The detail patameters are listed in the following table.

Table 2: Concrete parameters for transciphering are as follows: The $\log(q)$ columns represent the consumption of primes, with Base, StC, Mult, EvalMod, and CtS indicating the prime size and the number of primes consumed in each step. The $\log(p)$ column denotes the bit-size and the number of temporary moduli used for key switching.

	N	$(h, \tilde{h})$	$\log(QP)$	*dnum*	*depth per round*
Param-AES-13*	2^{14}	256, 16	433	13	3
Param-AES-14	2^{15}	256, 32	829	4	3
Param-Rasta-14	2^{15}		830	3	2

	$\log(q)$					$\log(p)$
	Base	StC	Mult	EvalMod	CtS	
Param-AES-13	32	28	31×3	32×6	28×2	32
Param-AES-14	38	36×2	37×3	38×7	38×4	38×5
Param-Rasta-14	42	41×2	41×2	42×7	40×3	42×5

* The significantly smaller $\log(qp) \approx 64$-bit at the lowest level permits a lower Hamming weight of $\tilde{h} = 16$ while still achieving more than 128-bit security, according to [APS15].

6.2 AES Transciphering Results

We compare our implementation with the previous best results (directly taken from their papers) in Table 3.

As shown, the three CKKS-based methods achieve the least time per block in an amortized sense, which translates to the highest throughput. However, in terms of latency, several TFHE-based methods outperform the CKKS ones. This is because TFHE-based methods have a batch size of 1. In other words, they only have to deal with one AES block at a time, whereas CKKS-based methods leverage the SIMD feature and deal with multiple AES blocks at a time.

The performance of BGV-based method [GHS12] falls between that of CKKS ones and TFHE ones: it exhibits moderate latency and time cost per block. Note that [GHS12] would have similar throughput as [ADE+23] if we divide their time cost by 64. However, FHE is not only computationally intensive but also incurs a significant memory footprint, which makes full parallelization challenging. Indeed, our 64-threaded implementation can only achieve a 20-30 times performance increase compared to our own single-threaded implementation.

Among the three CKKS-based methods, **Param-AES-13** exhibits lower latency but a higher amortized cost compared to **Param-AES-14** since its smaller parameter can accommodate only half as many AES blocks. But **Param-AES-14** achieves a 5× reduction in latency and a 2.5× reduction in amortized time cost than [ADE+23] under smaller parameters. This demonstrates the effectiveness of XBOOT.

Table 3: Comparison of AES transciphering efficiency. Following [ADE+23], we report the latency (the time required to transcipher a smallest batch of symmetric cipher) and the amortized time cost per block (calculated as latency/batch size). Each AES block contains 128 bits.

Scheme	Work	Latency	Time per block	Batch size
BGV	[GHS12]	240 s	2 s	120 blocks
TFHE	[SMK22]	252 s	252 s	1 block
	[TCBS23]	270 s	270 s	
	[BPR24]	211 s	211 s	
	[WWL+23]	86 s	86 s	
	[WLW+24]	110 s	110 s	
	[WLW+24]	46 s	46 s	
	[SAP24]	61.9 s	61.9 s	
	[SAP24]*	0.95 s	0.95 s	
CKKS	[ADE+23]*	1200 s	37 ms	2^{15} blocks
	Param-AES-13*	**153 s**	**18.7 ms**	2^{13} blocks
	Param-AES-14*	**236 s**	**14.4 ms**	2^{14} blocks

* 64-threading is applied. The multi-threaded version of [SAP24] uses 16 threads while we use 64 threads, so we improve their performance by 4× for comparison.
Works without star symbol report their results using single thread.

6.3 Rasta Transciphering Results

Transciphering for Rasta have been studied in several previous papers. We choose the Rasta-6 cipher and compare our implementation with their results in Table 4.

As shown, the BGV and TFHE methods require tens to thousands of seconds to process one block of Rasta cipher, significantly slower than our result of 18.5 milliseconds. Even if

Table 4: Comparison of Rasta-6 transciphering efficiency. Each block contains 351 bits.

Scheme	Work	Latency	Time per block	Batch size
BGV	[DEG+18]	815 s	815 s	1 block
	[DGH+23]	207 s	207 s	1 block
	[CHK+21]	11980 s	16.6 s	720 blocks
TFHE	[DGH+23]	3275 s	3275 s	1 block
CKKS	BLEACH*	> 16124 s	> 0.49 s	2^{15} blocks
	Param-Rasta-14*	**303 s**	**18.5 ms**	2^{14} blocks

* 64-threading is applied.
Works without star symbol report their results using single thread.

we grant them the advantage of perfect parallelization and divide their time cost by 64, the best-performing [CHK+21] is still at least 14 times slower than ours when evaluated by amortized time cost per block.

Since Rasta-CKKS transciphering using BLEACH without free XOR is extremely time-consuming, we chose not to implement it and instead estimated its cost to highlight the importance of free XOR. The number $16124 \approx 351 \times 350 \times 7 \times 1.2/64$ is explained as follows: Each evaluation of a batched random matrix requires 351×350 XOR operations. For the seven random matrices in Rasta-6, this translates to at least $351 \times 350 \times 7$ multiply-then-relinearlize operations when using BLEACH. We only consider the cost of the KS (Key Switching) step in relinearization, which takes approximately 1.2 second according to [MBTPH20, Table 5]. Then we divide the cost by 64 to simulate multi-threading. The amortized time cost block is at least $16124/2^{15} \approx 0.49$s. This is at least $26\times$ slower than ours because we treat the XORs as cheap additions.

Other FHE-friendly ciphers. Dasta is not compatible with CKKS due to its randomized bit permutation in each block. The efficiency of CKKS stems from its support for SIMD operations, which enables the simultaneous execution of the same operations on multiple data chunks. However, Dasta's use of different bit permutations for each block makes it incompatible with these SIMD operations.

LowMC employs a randomized matrix for its linear layer, which requires more rounds and consequently additional bootstrapping. While XBOOT could be beneficial for LowMC, it would result in higher transciphering latency compared to Rasta. In contrast, Fasta, with its larger state size, requires significantly larger FHE ciphertexts to fully encapsulate the state, leading to considerably more RAM consumption during homomorphic evaluation.

6.4 Comparison for Transciphering Accuracy

Although the modulus chains in the experiments are of reduced size, the transciphering accuracy of the Free-XOR framework is significantly enhanced due to the customized bootstrapping for a limited message domain. In [ADE+23], it is claimed that the average noise is $\mathrm{AVG}(|\mathsf{pt} - \mathrm{Dec}_{\mathsf{sk}}(\mathsf{ct})|) \approx 1.16 \times 2^{-10}$, indicating a bias in the binary values from the original set $\{0,1\}^{N/2}$.

In this work, we integrate the optimizations from [BCKS24], exploit the limited Boolean message domain of transciphering, carefully adapting the bootstrapping process to enhance performance while maintaining high accuracy. As evidence, **Param-AES-14**, which offers the best throughput, exhibits an average noise of $\mathrm{AVG}(|\mathsf{pt} - \mathrm{Dec}_{\mathsf{sk}}(\mathsf{ct})|) \approx 1.1 \times 2^{-19}$.

The configuration of **Param-AES-13** has an average noise of $\mathrm{AVG}(|\mathsf{pt} - \mathrm{Dec}_{\mathsf{sk}}(\mathsf{ct})|) \approx 1.2 \times 2^{-13}$.

7 Conclusion

In this paper, we study the problem of homomorphic computation on Boolean circuits using CKKS. We propose a novel method XBOOT that consumes zero multiplication depth per XOR gate instead of one. By achieving a shallower multiplication depth, we are able to enhance the AES transciphering efficiency to 256KB per 236 seconds, which is a 5× latency improvement and 2.5× throughput improvement over the state-of-the-art method. We also adopt our method to Rasta cipher to show the generalability of XBOOT. As a result, we can better leverage the SIMD feature of CKKS homomorphic encryption and improve the throughput of Rasta transciphering by more than one order of magnitude.

References

[ADE+23] Ehud Aharoni, Nir Drucker, Gilad Ezov, Eyal Kushnir, Hayim Shaul, and Omri Soceanu. E2e near-standard and practical authenticated transciphering. Cryptology ePrint Archive, Paper 2023/1040, 2023.

[AKP24] Andreea Alexandru, Andrey Kim, and Yuriy Polyakov. General functional bootstrapping using CKKS. *IACR Cryptol. ePrint Arch.*, page 1623, 2024.

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

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *Advances in Cryptology - EUROCRYPT 2015*, volume 9056 of *Lecture Notes in Computer Science*, pages 430–454. Springer, 2015.

[BCC+22] Youngjin Bae, Jung Hee Cheon, Wonhee Cho, Jaehyung Kim, and Taekyung Kim. META-BTS: bootstrapping precision beyond the limit. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 223–234. ACM, 2022.

[BCKS24] Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, and Damien Stehlé. Bootstrapping bits with CKKS. In *Advances in Cryptology - EUROCRYPT 2024*, volume 14652 of *Lecture Notes in Computer Science*, pages 94–123. Springer, 2024.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3):13:1–13:36, 2014.

[BKSS24] Youngjin Bae, Jaehyung Kim, Damien Stehlé, and Elias Suvanto. Bootstrapping small integers with CKKS. In Kai-Min Chung and Yu Sasaki, editors, *Advances in Cryptology - ASIACRYPT 2024*, volume 15484 of *Lecture Notes in Computer Science*, pages 330–360. Springer, 2024.

[BMTH21] Jean-Philippe Bossuat, Christian Mouchet, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In Anne Canteaut and François-Xavier

Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021*, volume 12696 of *Lecture Notes in Computer Science*, pages 587–617. Springer, 2021.

[BPR24] Nicolas Bon, David Pointcheval, and Matthieu Rivain. Optimized homomorphic evaluation of boolean functions. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(3):302–341, 2024.

[BTH22] Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation. In Giuseppe Ateniese and Daniele Venturi, editors, *Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings*, volume 13269 of *Lecture Notes in Computer Science*, pages 521–541. Springer, 2022.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, 2011.

[BV14] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM J. Comput.*, 43(2):831–871, 2014.

[CCF$^+$16] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. In Thomas Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*, pages 313–333. Springer, 2016.

[CCK$^+$13] Jung Hee Cheon, Jean-Sébastien Coron, Jinsu Kim, Moon Sung Lee, Tancrède Lepoint, Mehdi Tibouchi, and Aaram Yun. Batch fully homomorphic encryption over the integers. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 315–335. Springer, 2013.

[CCS19] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019*, volume 11477 of *Lecture Notes in Computer Science*, pages 34–54. Springer, 2019.

[CGGI17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017*, volume 10624 of *Lecture Notes in Computer Science*, pages 377–408. Springer, 2017.

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

[CHK$^+$18] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology -*

EUROCRYPT 2018, volume 10820 of *Lecture Notes in Computer Science*, pages 360–384. Springer, 2018.

[CHK$^+$21] Jihoon Cho, Jincheol Ha, Seongkwang Kim, ByeongHak Lee, Joohee Lee, Jooyoung Lee, Dukjae Moon, and Hyojin Yoon. Transciphering framework for approximate homomorphic encryption. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2021*, volume 13092 of *Lecture Notes in Computer Science*, pages 640–669. Springer, 2021.

[CIR22] Carlos Cid, John Petter Indrøy, and Håvard Raddum. FASTA - A stream cipher for fast FHE evaluation. In Steven D. Galbraith, editor, *Topics in Cryptology - CT-RSA 2022 - Cryptographers' Track at the RSA Conference 2022, Virtual Event, March 1-2, 2022, Proceedings*, volume 13161 of *Lecture Notes in Computer Science*, pages 451–483. Springer, 2022.

[CKK20] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. Efficient homomorphic comparison methods with optimal complexity. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020*, volume 12492 of *Lecture Notes in Computer Science*, pages 221–256. Springer, 2020.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.

[CLW$^+$24] Chunling Chen, Xianhui Lu, Ruida Wang, Zhihao Li, Xuan Shen, and Benqiang Wei. Free-xor gate bootstrapping. *IACR Cryptol. ePrint Arch.*, page 1703, 2024.

[DEG$^+$18] Christoph Dobraunig, Maria Eichlseder, Lorenzo Grassi, Virginie Lallemand, Gregor Leander, Eik List, Florian Mendel, and Christian Rechberger. Rasta: A cipher with low anddepth and few ands per bit. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018*, volume 10991 of *Lecture Notes in Computer Science*, pages 662–692. Springer, 2018.

[DEM15] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. Higher-order cryptanalysis of lowmc. In Soonhak Kwon and Aaram Yun, editors, *Information Security and Cryptology - ICISC 2015*, volume 9558 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2015.

[DGH$^+$23] Christoph Dobraunig, Lorenzo Grassi, Lukas Helminger, Christian Rechberger, Markus Schofnegger, and Roman Walch. Pasta: A case for hybrid homomorphic encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3):30–73, 2023.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015*, volume 9056 of *Lecture Notes in Computer Science*, pages 617–640. Springer, 2015.

[DMPS24] Nir Drucker, Guy Moshkowich, Tomer Pelleg, and Hayim Shaul. BLEACH: cleaning errors in discrete computations over CKKS. *J. Cryptol.*, 37(1):3, 2024.

[FRL19] Hayato Fujii, Félix Carvalho Rodrigues, and Julio López. Fast AES implementation using armv8 ASIMD without cryptography extension. In Jae Hong Seo, editor, *Information Security and Cryptology - ICISC 2019*, volume 11975 of *Lecture Notes in Computer Science*, pages 84–101. Springer, 2019.

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

[GAH$^+$23] Lorenzo Grassi, Irati Manterola Ayala, Martha Norberg Hovd, Morten Øygarden, Håvard Raddum, and Qingju Wang. Cryptanalysis of symmetric primitives over rings and a key recovery attack on rubato. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology - CRYPTO 2023*, volume 14083 of *Lecture Notes in Computer Science*, pages 305–339. Springer, 2023.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178. ACM, 2009.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867. Springer, 2012.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2013.

[HK20] Kyoohyung Han and Dohyeong Ki. Better bootstrapping for approximate homomorphic encryption. In Stanislaw Jarecki, editor, *Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings*, volume 12006 of *Lecture Notes in Computer Science*, pages 364–390. Springer, 2020.

[HL20] Phil Hebborn and Gregor Leander. Dasta - alternative linear layer for rasta. *IACR Trans. Symmetric Cryptol.*, 2020(3):46–86, 2020.

[JM22] Charanjit S. Jutla and Nathan Manohar. Sine series approximation of the mod function for bootstrapping of approximate HE. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology - EUROCRYPT 2022*, volume 13275 of *Lecture Notes in Computer Science*, pages 491–520. Springer, 2022.

[JPK$^+$24] Jae Hyung Ju, Jaiyoung Park, Jongmin Kim, Minsik Kang, Donghwan Kim, Jung Hee Cheon, and Jung Ho Ahn. Neujeans: Private neural network inference with joint optimization of convolution and FHE bootstrapping. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pages 4361–4375. ACM, 2024.

[KDE$^+$24] Andrey Kim, Maxim Deryabin, Jieun Eom, Rakyong Choi, Yongwoo Lee, Whan Ghang, and Donghoon Yoo. General bootstrapping approach for rlwe-based homomorphic encryption. *IEEE Trans. Computers*, 73(1):86–96, 2024.

[KN24] Jaehyung Kim and Taeyeong Noh. Modular reduction in CKKS. *IACR Cryptol. ePrint Arch.*, page 1638, 2024.

[KSS12] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 285–300. USENIX Association, 2012.

[lat24] Lattigo v6. Online: https://github.com/tuneinsight/lattigo, August 2024. EPFL-LDS, Tune Insight SA.

[LKSM24] Fukang Liu, Abul Kalam, Santanu Sarkar, and Willi Meier. Algebraic attack on fhe-friendly cipher HERA using multiple collisions. *IACR Trans. Symmetric Cryptol.*, 2024(1):214–233, 2024.

[LLK+22] Yongwoo Lee, Joon-Woo Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and HyungChul Kang. High-precision bootstrapping for approximate homomorphic encryption by error variance minimization. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology - EUROCRYPT 2022*, volume 13275 of *Lecture Notes in Computer Science*, pages 551–580. Springer, 2022.

[LLKN23] Joon-Woo Lee, Eunsang Lee, Young-Sik Kim, and Jong-Seon No. Rotation key reduction for client-server systems of deep neural network on fully homomorphic encryption. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology - ASIACRYPT 2023*, volume 14443 of *Lecture Notes in Computer Science*, pages 36–68. Springer, 2023.

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

[LLL+22] Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 12403–12422. PMLR, 2022.

[MBTPH20] Christian Vincent Mouchet, Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. Lattigo: A multiparty homomorphic encryption library in go. In *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, pages 64–70, 2020.

[PF15] NIST FIPS PUB and PUB FIPS. 202: Sha-3 standard: Permutation-based hash and extendable-output functions. *Federal Information Processing Standards Publication*, 202, 2015.

[SAP24] Leonard Schild, Aysajan Abidin, and Bart Preneel. Fast transciphering via batched and reconfigurable LUT evaluation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(4):205–230, 2024.

[SMK22] Roy Stracovsky, Rasoul Akhavan Mahdavi, and Florian Kerschbaum. Faster evaluation of aes using tfhe. Poster Session, FHE.Org - 2022, 2022. Available at https://rasoulam.github.io/data/poster-aes-tfhe.pdf.

[TCBS23] Daphné Trama, Pierre-Emmanuel Clet, Aymen Boudguiga, and Renaud Sirdey. At last! A homomorphic AES evaluation in less than 30 seconds by means of TFHE. *IACR Cryptol. ePrint Arch.*, page 1020, 2023.

[WLW$^+$24] Benqiang Wei, Xianhui Lu, Ruida Wang, Kun Liu, Zhihao Li, and Kunpeng Wang. Thunderbird: Efficient homomorphic evaluation of symmetric ciphers in 3gpp by combining two modes of TFHE. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(3):530–573, 2024.

[WWL$^+$23] Benqiang Wei, Ruida Wang, Zhihao Li, Qinju Liu, and Xianhui Lu. Fregata: Faster homomorphic evaluation of AES via TFHE. In Elias Athanasopoulos and Bart Mennink, editors, *Information Security - 26th International Conference, ISC 2023, Groningen, The Netherlands, November 15-17, 2023, Proceedings*, volume 14411 of *Lecture Notes in Computer Science*, pages 392–412. Springer, 2023.

[WWL$^+$24] Ruida Wang, Yundi Wen, Zhihao Li, Xianhui Lu, Benqiang Wei, Kun Liu, and Kunpeng Wang. Circuit bootstrapping: Faster and smaller. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology - EUROCRYPT 2024*, volume 14652 of *Lecture Notes in Computer Science*, pages 342–372. Springer, 2024.