Walsh LUT Evaluation on Lazy Bits for CKKS AES Transciphering

Rostin Shokri and Nektarios Georgios Tsoutsos

University of Delaware
{rostinsh, tsoutsos}@udel.edu

Abstract. In this work we propose a novel Boolean lookup-table evaluation methodology over binary CKKS when circuit XORs are kept lazy, i.e., evaluated as additions whose least significant bits remain correct. Our method represents a LUT in the Walsh basis, forms the required parity sums by lazy CKKS additions, and packs them into ciphertext slots. We then use CKKS binary bootstrapping as a refresh step: the StC stage maps the packed lazy parities to MSB-encoded bits, removing the overflow; CtS places the parity values in slots; and $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$ cleans the binary noise, leaving clean parities in the slot domain. The LUT is then evaluated by recombining these parities with plaintext Walsh coefficients. This decouples the LUT size from the multiplicative depth of the surrounding circuit: large LUTs can be handled by cleaning selected factored parity signs and spending only a small constant depth in recombination. We apply this framework to AES-CTR transciphering. The AES S-box is evaluated with a nibble-split Walsh decomposition, which supports more AES blocks at the cost of one additional multiplication depth. The AES state remains in full complex CKKS packing so real and imaginary lanes carry independent AES blocks. In CPU experiments, the Walsh S-box AES-CTR algorithm is 3.25× faster than the sparse-bootstrapping XBOOT variant at the same 1024-block batch size.

Keywords: Homomorphic Encryption · CKKS · AES Transciphering.

1 Introduction

Fully homomorphic encryption (FHE) allows a server to compute on encrypted data without seeing the plaintext [30]. This abstraction has moved beyond theory into public deployments and tooling: Microsoft Edge's Password Monitor uses homomorphic encryption in a private set-intersection protocol, Apple describes production use of homomorphic encryption for private lookups and Enhanced Visual Search in Photos, and HEIR provides an open compiler infrastructure for FHE programs [6, 8, 35].

The main cost is that native FHE ciphertexts are large and expensive for clients to create. Homomorphic transciphering, also called hybrid homomorphic encryption, addresses this bottleneck. The client encrypts data with a compact

symmetric cipher, such as AES [28], and sends the symmetric ciphertexts together with an FHE encryption of the symmetric key. The server then homomorphically evaluates the decryption circuit and obtains FHE ciphertexts that encrypt the original data, ready for the rest of the encrypted computation.

This hybrid approach reduces communication bandwidth and client-side work and has become an active research area [37]. One line of work homomorphically evaluates existing or standardized ciphers, including AES [13, 38, 40] and Ascon [25,29,39]. Another line designs new ciphers around the target FHE scheme, such as FRAST [21] for TFHE [20]. We focus on AES-CTR because AES is standardized, widely deployed, and has decades of cryptanalytic study. The challenge is that AES was not designed for homomorphic evaluation: each round contains many XORs and a nonlinear S-box.

CKKS is an FHE scheme that supports approximate arithmetic over packed real or complex slots [19]. In CKKS-AES, Boolean operations must therefore be represented inside an arithmetic scheme. Prior systems make different tradeoffs: they pack many AES blocks across the slot space, evaluate the S-box with lookup-table or functional-bootstrapping techniques, and either spend multiplicative depth for XOR or do lazy XOR using CKKS additions followed by binary bootstrapping to remove the accumulated overflow [2, 10, 22, 33, 36].

In this work, we take a different approach. Prior CKKS-AES systems either evaluate the AES S-box separately from the binary cleanup step [2,10,33,36], or use functional-bootstrapping-style LUT evaluation, which can be powerful but is expensive and requires high depth and a large ring dimension [22]. Our construction moves the binary cleanup into the Walsh representation of the S-box: the bootstrap refreshes the packed parity information needed for the Walsh expansion, and the S-box is then recovered by plaintext-coefficient recombination.

The key idea is to evaluate LUTs in the Walsh basis on lazy CKKS bits. We first form the parity information required by the Walsh representation using lazy additions. These parities are packed into one ciphertext, cleaned and refreshed together by CKKS binary bootstrapping, and then converted into the signs needed for Walsh recombination. Since a Walsh expansion writes a Boolean lookup table as a linear combination of parity characters, cleaning the required parity signs is enough to recover the LUT output.

Prior CKKS-AES designs also typically treat the imaginary lane as unavailable, move parts of the circuit to real-only packing [36], or convert between the standard CKKS ring and the conjugate-invariant CKKS ring (CI-CKKS) [10,32]. In contrast, our design keeps the computation in full complex CKKS ciphertexts and uses the real and imaginary components as independent AES lanes.

To improve the latency/throughput tradeoff, we introduce a nibble-split Walsh decomposition. Instead of cleaning every full-byte Walsh character directly, we clean only the low- and high-nibble parity information and spend one additional multiplication depth to finish the recombination. This supports many more AES blocks per packed cleanup step while targeting moderate batch sizes rather than only maximum full-slot throughput. Our contributions are as follows:

– We formulate a Walsh-basis method for evaluating general Boolean LUTs on lazy-XOR CKKS circuits. The method forms the required parity sums by CKKS additions, extracts the corresponding parity signs through binary bootstrapping, and then recovers the LUT output by plaintext-coefficient Walsh recombination.
– We give an exact nibble-split Walsh decomposition of the AES S-box that extracts a small packed set of parity signs and uses one additional multiplicative depth to support moderate-size AES transciphering block counts while sharing one parity-cleaning step across many S-box instances.
– Unlike previous CKKS-AES works that restrict parts of the AES circuit to real-only packing, use costly conversions between CI-CKKS and standard CKKS rings, or treat the imaginary lane as unavailable, our solution works with full complex-packed ciphertexts. Each active complex slot carries one AES block in the real lane and one in the imaginary lane, doubling the usable lanes and the number of AES blocks that can be transciphered.

Roadmap: Section 2 discusses notation and background needed for our algorithms, while Section 3 presents our Walsh-transform LUT evaluation on lazy bits, including packed parity cleaning and plaintext-coefficient recombination. In Section 4, we specialize the construction to the AES S-box, full-complex AES packing, and AES-CTR transciphering, while Section 5 evaluates our algorithms and compares with the XBOOT CPU baseline and recent GPU CKKS-AES systems. Section 6 discusses related works on CKKS-AES transciphering and shallow functional-bootstrapping, and Section 7 concludes the paper.

2 Background

2.1 CKKS Preliminaries

CKKS is an RLWE-based homomorphic encryption scheme for approximate arithmetic over packed real or complex values [19]. For ring dimension N, CKKS encrypts $N/2$ complex slots. To encode a slot vector, CKKS scales the values by a factor Δ and maps them into the polynomial ring $R_Q = \mathbb{Z}_Q[X]/(X^N+1)$ with an inverse FFT. Decryption recovers a noisy encoded polynomial and applies the corresponding forward FFT to obtain approximate slot values. Homomorphic addition and multiplication act slotwise on these encrypted slots.

2.2 Binary Bootstrapping and Lazy XOR

CKKS bootstrapping refreshes a ciphertext by homomorphically moving between slot and coefficient representations, raising the modulus, and applying a periodic modular reduction [17]. In the binary setting we use the binary bootstrapping target of [11], which we denote by $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$. We distinguish this from the generic CKKS $\mathsf{EvalMod}$ used for approximate modular reduction: $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$ is the bit-cleaning $\mathsf{EvalMod}$ function whose output is close to the least significant bit of an integer message. If a homomorphic computation produces an integer $u =$

$b+2t$, then b is still the desired bit even though u may be outside $\{0,1\}$. The StC step in binary bootstrapping removes the even overflow, and the $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$ polynomial performs the noise reduction needed to clean the bit. Concretely, after the standard BinBoot normalization, $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$ approximates the periodic parity function

$$f_{\mathrm{BinBoot}}(z) = \frac{1-\cos(2\pi z)}{2}.$$

We use the same periodic function with the interval scaling parameter K, namely $f_{\mathrm{BinBoot},K}(z) = (1-\cos(2\pi K z))/2$, so the polynomial only needs to be accurate near the scaled integer and half-integer neighborhoods used by binary bootstrapping.

We exploit this observation for lazy XOR. Instead of evaluating $x \oplus y$ with a multiplication, we add $x+y$. The result may be $0, 1$, or 2, but its least significant bit is the XOR. XBOOT uses this idea to make XOR gates multiplication-free in CKKS and relies on binary bootstrapping to recover the bit values [36]. Our Walsh construction uses the same invariant, but it applies it directly to the parities needed by a Boolean lookup table.

2.3 AES-CTR Transciphering

AES operates on 128-bit blocks. For AES-128, the circuit has ten rounds. Each full round applies SubBytes, ShiftRows, MixColumns, and AddRoundKey; the final round omits MixColumns. In CTR mode, AES encrypts public counter blocks to produce a keystream, and the keystream is XORed with the user ciphertext. We recall this standard AES-128-CTR flow in Appendix A. CTR is convenient for homomorphic transciphering because the counter blocks are public while the AES key is encrypted under FHE.

In the standard bit-sliced CKKS layout, the AES state is represented by 128 ciphertexts, one for each bit position. Slot s of every ciphertext belongs to AES block s. The linear AES layers are XOR-linear, so in a lazy-XOR setting they become additions and permutations. The nonlinear bottleneck is the S-box.

2.4 Walsh Representation of Boolean Functions

Let $F : \{0,1\}^m \to \{0,1\}^r$ be a Boolean vector function, and let $F_j(x)$ be output bit j. It is useful to work with the sign version of an output bit:

$$g_j(x) = (-1)^{F_j(x)} = 1 - 2F_j(x).$$

Every Boolean sign function has a Walsh expansion [15]:

$$g_j(x) = 2^{-m} \sum_{a \in \{0,1\}^m} W_j[a]\chi_a(x), \qquad \chi_a(x) = (-1)^{a \cdot x}.$$

Here a is a binary mask and $a \cdot x$ is the mod-2 inner product, i.e., the parity of the selected input bits. Therefore

$$F_j(x) = \frac{1}{2} - 2^{-(m+1)} \sum_a W_j[a]\chi_a(x).$$

The Walsh basis separates LUT evaluation into parity-sign extraction and linear recombination. For a mask a, a lazy circuit can form

$$\tau_a = \sum_i a_i x_i$$

by CKKS additions. If $x_i = b_i + 2e_i$, then $\tau_a = p_a + 2t_a$, where

$$p_a = \langle a, b \rangle_{\mathbb{F}_2}$$

is the selected mod-2 parity. Thus lazy overflow changes only the even part of τ_a, not its least significant bit. The packed $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$ call extracts the clean parity bit p_a; applying $1 - 2p_a$ gives the Walsh sign χ_a.

2.5 Threat Model

We work in the standard honest-but-curious outsourced-computation setting. The client keeps the CKKS secret key, encrypts the AES key under CKKS, encrypts bulk data under AES-CTR, and sends the resulting ciphertexts together with the public counter values. The server holds the public CKKS material and evaluation keys, evaluates the AES-CTR decryption circuit homomorphically, and may then continue with downstream CKKS computation.

The server is assumed to follow the prescribed evaluation procedure, including bootstrapping, but it may inspect all ciphertexts, public keys, evaluation keys, and intermediate values it sees during the computation. The security guarantee is that these observations reveal neither the AES key nor the underlying plaintext blocks. CKKS provides IND-CPA security under the RLWE assumption [34], making chosen-message ciphertexts computationally indistinguishable.

3 LUT Evaluation on Lazy Bits with Walsh Transform

3.1 Overview

We consider Boolean circuits in which XOR gates are evaluated lazily in CKKS. Rather than evaluating $x \oplus y$ with the Boolean polynomial $(x - y)^2$, which requires a ciphertext multiplication and consumes multiplicative depth, we add the ciphertexts. The sum may be an integer outside $\{0, 1\}$, but its least significant bit is still $x \oplus y$. For an m-input, r-output Boolean lookup table $F : \{0,1\}^m \to \{0,1\}^r$, our goal is to evaluate F directly on these lazy bit ciphertexts.

The Walsh basis turns LUT evaluation into parity extraction followed by linear recombination, following the workflow below:

lazy bit ciphertexts $\to$ packed parity sums $\to$ BinBoot
$\to$ Walsh signs $\to$ coefficient recombination.

The expensive cleaning step is therefore applied to the parity characters that the LUT needs, rather than to every intermediate bit in the surrounding circuit.

3.2 Lazy Walsh Parities

Let the encrypted inputs be lazy bits $x_i = b_i + 2e_i$, with $b_i \in \{0,1\}$. For a Walsh mask $a \in \{0,1\}^m$, form the parity sum

$$\tau_a = \sum_{i=0}^{m-1} a_i x_i = \left(\sum_{i=0}^{m-1} a_i b_i \right) + 2E_a.$$

Let

$$p_a = \left(\sum_{i=0}^{m-1} a_i b_i \right) \bmod 2 = \langle a, b \rangle_{\mathbb{F}_2}.$$

Then $\tau_a = p_a + 2t_a$ for some integer t_a. In other words, the CKKS additions may accumulate arbitrary even overflow, but the least significant bit of the packed sum is still the Walsh parity p_a. No ciphertext multiplication is needed to form this value; it is only a sum of selected input ciphertexts.

We use the same StC overflow-removal mechanism as XBOOT [36]: StC moves the lazy parity sum into the coefficient/MSB representation used by binary bootstrapping. The subsequent ModRaise, CtS, and binary $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$ polynomial convert this packed representation back into clean slot bits. This cleanup is necessary before Walsh recombination: the intermediate bootstrap representation after StC is not a usable CKKS slot value, and the Walsh LUT needs clean parity bits or signs in slot encoding so that it can be multiplied by plaintext coefficients.

3.3 Packed Walsh LUT Evaluation

Extracting every parity in a separate bootstrap is too expensive. We instead pack many parity sums into slot segments before calling BinBoot. Let $\mathcal{A}$ be the nonzero set of Walsh masks required by the chosen LUT representation, and let B be the width of one packed segment. Segment $\mathrm{Seg}(a)$ stores the lazy parity sum τ_a in slots $\mathrm{Seg}(a)B, \ldots, (\mathrm{Seg}(a)+1)B - 1$.

After the packed binary bootstrap, each segment contains a clean parity bit $p_a = \langle a, b \rangle_{\mathbb{F}_2}$. We convert it into the Walsh sign $\chi_a(b) = (-1)^{\langle a,b \rangle_{\mathbb{F}_2}} = 1 - 2p_a$. With the constant character $\chi_0 = 1$, the Walsh coefficients $W_j[a]$ of the sign function $1 - 2F_j$,

$$W_j[a] = \sum_{u \in \{0,1\}^m} (1 - 2F_j(u))\chi_a(u),$$

determine output bit j:

$$F_j(b) = \frac{1}{2} - 2^{-(m+1)} \sum_{a \in \mathcal{A} \cup \{0\}} W_j[a]\chi_a(b).$$

The recombination is a plaintext-coefficient linear combination once the needed signs are available.

3.4 Reducing Cleaned Parities by Walsh Factorization

The goal of the factorization step is to reduce how many Walsh parity signs must be cleaned by BinBoot. The direct Walsh expansion can be dense: an m-bit LUT may require up to $2^m - 1$ nonconstant characters. Cleaning all of these characters directly would require many parity-cleaning calls or a very large packed parity ciphertext. The core observation behind our construction is that Walsh characters multiply across an input split: $\chi_{a_1 \| a_2}(x_1 \| x_2) = \chi_{a_1}(x_1)\chi_{a_2}(x_2)$.

This lets us extract signs for smaller input groups and synthesize the full characters with ciphertext multiplications. The tradeoff is explicit: fewer BinBoot-cleaned parity signs in exchange for a small, controlled multiplication depth during recombination.

In Algorithm 1, BinBoot denotes the full binary bootstrap applied to the packed parity sums: StC, ModRaise, CtS, and the binary $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$ polynomial. ComplexBinBoot is the same operation on a full-complex packed ciphertext; after CtS it separates the real and imaginary lanes and applies binary $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$ to both. $\mathrm{PackSeg}(u, s, B)$ is implemented by CKKS rotations and additions: rotate the ciphertext u by sB slots so that its active B-slot range lands in segment s, then add it into the packed accumulator P. $\mathrm{UnpackSeg}(P, s, B)$ applies the inverse rotation by $-sB$ to bring segment s back to the active slot range. In the implementation, the many unpack rotations are evaluated with hoisted key-switching [31].

Algorithm 1. Packed Walsh LUT evaluation on lazy bits

Input: Lazy bit ciphertexts $x_0, \ldots, x_{m-1}$; packing width B; nonzero mask set $\mathcal{A}$; Walsh tables W
Output: WalshSigns returns signs χ_a; WalshLut returns output bits $y_0, \ldots, y_{r-1}$

procedure WalshSigns($\mathcal{X} = (x_0, \ldots, x_{m-1}), B, \mathcal{A}$)
  $P \leftarrow 0$
  for $a \in \mathcal{A}$ **do**
    $\tau_a \leftarrow \sum_{i=0}^{m-1} a_i x_i$
    $P \leftarrow P + \mathrm{PackSeg}(\tau_a, \mathrm{Seg}(a), B)$
  $P \leftarrow \mathrm{BinBoot}(P)$ ▷ *BinBoot [11]*
  $\chi_0 \leftarrow 1$
  for $a \in \mathcal{A}$ **do**
    $\chi_a \leftarrow 1 - 2\,\mathrm{UnpackSeg}(P, \mathrm{Seg}(a), B)$
  return $\{\chi_a\}_{a \in \mathcal{A} \cup \{0\}}$
procedure WalshLut($\mathcal{X} = (x_0, \ldots, x_{m-1}), B, \mathcal{A}, W$)
  $\{\chi_a\}_{a \in \mathcal{A} \cup \{0\}} \leftarrow \mathrm{WalshSigns}(\mathcal{X}, B, \mathcal{A})$
  for $j = 0, \ldots, r-1$ **do**
    $y_j \leftarrow 1/2 - 2^{-(m+1)} \sum_{a \in \mathcal{A} \cup \{0\}} W_j[a]\chi_a$
  return $y_0, \ldots, y_{r-1}$

Algorithm 1 describes the generic packed Walsh-cleaning primitive. The reusable subroutine is WalshSigns; WalshLut shows the direct recombination form. Direct or sparse LUTs use the cleaned characters in the displayed linear sum, while factored decompositions reuse WalshSigns and replace that final recombination with products of smaller cleaned character groups. The correctness

invariant is the same in all cases: lazy CKKS additions compute parity sums whose least significant bits are the Walsh characters needed by the LUT.

4 Application to AES Transciphering

We instantiate the lazy-bit Walsh LUT method for AES-CTR transciphering. We propose a novel Walsh S-box decomposition, evaluate the AES circuit in a full-complex packing layout, and evaluate the linear AES layers as lazy CKKS additions in complex form.

4.1 Nibble-Split AES Walsh S-box

Write an AES byte as $x = (x_L, x_H)$, where x_L and x_H are the low and high nibbles. We generate the 15 nonconstant Walsh characters for the low nibble and the 15 nonconstant characters for the high nibble:

$$L_\ell = \chi_\ell(x_L), \quad H_h = \chi_h(x_H), \qquad \ell, h \in \{1, \ldots, 15\}.$$

The zero-mask characters $L_0 = H_0 = 1$ are constants and need no BinBoot cleaning. The direct Walsh formula computes each S-box output bit as

$$\mathsf{S}_j(x) = \frac{1}{2} - 2^{-9} \sum_{\ell=0}^{15} \sum_{h=0}^{15} W_j[\ell, h] L_\ell H_h.$$

For AES, all Walsh coefficients in this table are divisible by four. We therefore store the scaled table $\widetilde{W}_j[\ell, h] = W_j[\ell, h]/4$ and evaluate the equivalent formula

$$\mathsf{S}_j(x) = \frac{1}{2} - 2^{-7} \sum_{\ell=0}^{15} \sum_{h=0}^{15} \widetilde{W}_j[\ell, h] L_\ell H_h.$$

This changes the final normalization from $1/512$ to $1/128$, without changing the exact S-box value.

The distinction matters for CKKS parameter selection. The Walsh recombination first forms an integer numerator from the sign products and plaintext coefficients, then normalizes it back to a bit value. A factor 2^{-t} consumes t bits of scale: to keep the output bit at the normal CKKS scale after the fixed-point normalization, the ciphertext modulus at this stage must contain those t extra bits of headroom for the scaling and rounding noise. Using the unscaled table would require 9 extra bits for $1/512 = 2^{-9}$. Since we use $W/4$, the denominator is only $1/128 = 2^{-7}$, so the Walsh recombination needs 7 extra modulus bits instead of 9.

After BinBoot cleans the packed parity bits, WalshSigns converts them to Walsh signs. We then evaluate the S-box by plaintext-coefficient recombination. For one lane λ, define

$$A^{\lambda}_{j,h} = \sum_{\ell=0}^{15} \widetilde{W}_j[\ell, h] L^{\lambda}_\ell.$$

Then the output bit is

$$b_j^\lambda = \frac{1}{2} - 2^{-7}\left(A_{j,0}^\lambda + \sum_{h=1}^{15} A_{j,h}^\lambda H_h^\lambda\right).$$

The low forms $A_{j,h}^\lambda$ are only additions and plaintext multiplications. Multiplying these forms by the nonzero high signs uses one ciphertext-ciphertext multiplication layer. The terms with $h = 0$, and in particular the constant and low-only parts, are added without multiplication. The construction computes the four bit pairs $(b_0, b_4), \ldots, (b_3, b_7)$ as complex pairs and then writes the eight bits back into the AES bit layout.

4.2 Full-Complex Packing and Masking

The default AES layout keeps two AES block batches in each CKKS complex slot. Let A be the number of active complex slots supplied by the input and let B be the Walsh packing width, with $A \le B$. In the full-width parameter sets of Section 5, $A = B$. For $0 \le s < A$, bit ciphertext x_i has the form

$$x_i[s] = x_i^{(0)}[s] + i\, x_i^{(1)}[s],$$

where the real lane carries logical AES block s, and the imaginary lane carries logical AES block $A + s$. Thus A active complex slots represent $2A$ logical AES blocks. Slots $A, \ldots, B-1$ are initially zero padding inside the Walsh packing window; during evaluation they are treated as ignored padding and may contain garbage. Slots $s \ge B$ are inactive and are masked away.

Before every Walsh SubBytes call, we apply an active-slot mask

$$M_B[s] = \begin{cases} 1, & 0 \le s < B, \\ 0, & \text{otherwise.} \end{cases}$$

Multiplying each AES state ciphertext by this plaintext mask removes stale values outside the Walsh packing window. The mask is required because padding and inactive slots may contain garbage, and the packing step uses rotations and additions. Without the mask, inactive slots could rotate into packed parity segments and pollute the values sent to BinBoot.

For one byte and the nibble-split Walsh decomposition, the AES S-box uses 30 packed segments:

$$\underbrace{1, \ldots, 15}_{\text{low masks}}, \underbrace{1, \ldots, 15}_{\text{high masks}},$$

each segment having width B complex slots. The packed ciphertext therefore occupies only $30B$ complex slots, but each slot contains both the real and imaginary AES lanes. With ring dimension $N = 2^{15}$, CKKS has 16384 complex slots, or 32768 effective real/imag lanes. The AESWalsh15 parameter set uses $A = B = 512$ and represents 1024 logical AES blocks. We reserve 32 segments, leaving two spare segments for regular packing.

Conceptually, packing places segment m into slots $mB, \ldots, (m+1)B-1$. For a low mask ℓ, this segment contains $\sum_{i=0}^{3} \ell_i x_i$; for a high mask h, it contains $\sum_{i=0}^{3} h_i x_{i+4}$. These are lazy XOR sums, so they may be larger than one, but their least significant bits are the desired parities. After the single ComplexBinBoot call, unpacking rotates each segment back to the active slot range and converts the cleaned bit b into the Walsh sign $1-2b$. A direct Walsh S-box would use the same principle with the full 8-bit mask set, namely 255 nonconstant parity segments plus the constant zero-mask character. The nibble-split Walsh decomposition is what reduces the packed cleanup from that full set to 30 nonconstant low/high-nibble parities. With packing width B, one ComplexBinBoot call cleans $30B$ parity values per temporary lane in the nibble-split construction, and the complex packing carries two such lanes through the same linear StC/CtS work.

Algorithm 2. Packed Walsh S-box for one AES byte

Input: Masked lazy bit ciphertexts $x_0, \ldots, x_7$; packing width B; scaled tables $\widetilde{W}$
Output: Output bits $y_0, \ldots, y_7$ in the same layout

procedure RecombineSigns$(L, H, \widetilde{W})$
  for $p = 0, \ldots, 3$ **do**
    $A^{\rho}_{p,h} \leftarrow \sum_{\ell=0}^{15} \widetilde{W}_{p+4\rho}[\ell, h] L_\ell$, for $\rho \in \{0,1\}$, $h = 0, \ldots, 15$
    $R^{\rho}_{p} \leftarrow A^{\rho}_{p,0} + \sum_{h=1}^{15} A^{\rho}_{p,h} H_h$, for $\rho \in \{0,1\}$
    $o_p \leftarrow (1/2 - 2^{-7} R^0_p) + i(1/2 - 2^{-7} R^1_p)$
  return $o_0, \ldots, o_3$
procedure WalshSbox$(x_0, \ldots, x_7, B, \widetilde{W})$
  $\{L^\lambda, H^\lambda\}_{\lambda=0,1} \leftarrow$ WalshSigns$((x_0, \ldots, x_7), B, \mathcal{A}_{\mathrm{LH}})$ ▷ *Alg. 1*
  $O^\lambda \leftarrow$ RecombineSigns$(L^\lambda, H^\lambda, \widetilde{W})$, for $\lambda \in \{0,1\}$
  return MergeRI(O^0, O^1)

Algorithm 2 presents the complete AES S-box dataflow for one byte in the complex-block layout. We use $\mathrm{LowSeg}(m) = m - 1$ and $\mathrm{HighSeg}(m) = 15 + m - 1$ for $m = 1, \ldots, 15$. Let $a^L_m = (m_0, \ldots, m_3, 0, \ldots, 0)$ and $a^H_m = (0, \ldots, 0, m_0, \ldots, m_3)$, and let $\mathcal{A}_{\mathrm{LH}} = \{a^L_m, a^H_m : m = 1, \ldots, 15\}$, with $\mathrm{Seg}(a^L_m) = \mathrm{LowSeg}(m)$ and $\mathrm{Seg}(a^H_m) = \mathrm{HighSeg}(m)$. The call to WalshSigns is Algorithm 1 specialized to these 30 low/high nibble masks: it builds one packed parity ciphertext, runs the shared binary bootstrap, and returns cleaned sign arrays for the real and imaginary AES-block lanes. We write $L^\lambda_m = \chi^\lambda_{a^L_m}$ and $H^\lambda_m = \chi^\lambda_{a^H_m}$ for $\lambda \in \{0,1\}$, with $L^\lambda_0 = H^\lambda_0 = 1$. Here λ indexes the real/imaginary AES-block lane, while ρ in RecombineSigns indexes the two output bit halves. Thus o_p stores bit p in its real component and bit $p+4$ in its imaginary component. Finally, MergeRI splits the four paired outputs from each lane and reassembles them into eight full-complex AES bit ciphertexts in the original layout.

4.3 AES-CTR Transciphering Schedule

We apply the Walsh S-box construction to AES-CTR transciphering with encrypted expanded AES round-key bits and public counter blocks. Round-key bits are encrypted in the same layout, with the same key bit placed in both real and

imaginary lanes. For a batch with A active complex slots, $\text{ENCODING}(\cdot, A, B)$ is the usual CKKS encoding into the full-complex bit-sliced layout: the first A blocks occupy real lanes, the next A blocks occupy imaginary lanes, and slots $A, \ldots, B-1$ are zero padding. The initial state is the encrypted round key XORed with the public counter, where XOR is implemented as addition.

WalshSubBytes(S,B) applies Algorithm 2 independently to each of the 16 AES bytes. MixColumnsLazy(S) is the XBOOT-style lazy MixColumns layer: AES XORs and the fixed multiplications by {02} and {03} are XOR-linear bit maps, so they remain lazy additions and bit permutations in the full-complex packing (see XBOOT [36] for details). FinalBinaryRefresh(T,A) removes the even overflow left by the final AES-CTR XOR by packing the 128 lazy bit ciphertexts into complex BinBoot calls. The general procedure is shown in Algorithm 5 in Appendix B.

Algorithm 3. AES-CTR transciphering with Walsh S-boxes

Input: $K_0, \ldots, K_{10}$: encrypted expanded AES round-key bits
Input: C: public counter blocks, with active slots A, packing width B, and $A \leq B$
Input: Y: AES-CTR ciphertext blocks
Output: 128 clean bit ciphertexts; valid plaintext blocks occupy real/imag slots $0, \ldots, A-1$

procedure $\text{WalshAesCtr}(K_0, \ldots, K_{10}, C, Y, A, B)$
  $\widehat{C} \leftarrow \text{ENCODING}(C, A, B)$
  $\widehat{Y} \leftarrow \text{ENCODING}(Y, A, B)$
  $S \leftarrow \widehat{C} + K_0$ ▷ *lazy AddRoundKey*
  for $r = 1, \ldots, 9$ **do**
    $S \leftarrow \text{MaskInactiveSlots}(S, B)$
    $S \leftarrow \text{WalshSubBytes}(S, B)$ ▷ *Alg. 2*
    $S \leftarrow \text{ShiftRows}(S)$
    $S \leftarrow \text{MixColumnsLazy}(S)$
    $S \leftarrow S + K_r$ ▷ *lazy AddRoundKey*
  $S \leftarrow \text{MaskInactiveSlots}(S, B)$
  $S \leftarrow \text{WalshSubBytes}(S, B)$
  $S \leftarrow \text{ShiftRows}(S)$
  $S \leftarrow S + K_{10}$ ▷ *final round has no MixColumns*
  $T \leftarrow S + \widehat{Y}$ ▷ *AES-CTR output XOR*
  $T \leftarrow \text{FinalBinaryRefresh}(T, A)$
  return T

Algorithm 3 presents the complete procedure. The returned 128 ciphertexts contain plaintext block s in real slot s and plaintext block $A + s$ in imaginary slot s, for $0 \leq s < A$; all other slots are padding or masked.

5 Experimental Evaluation

In this section we compare the proposed AES-CTR Walsh algorithm with current CPU and GPU baselines. We develop both the XBOOT-style baseline and our Walsh AES algorithm in a custom C++ implementation of CKKS. To show

Table 1. Parameter sets and effective packing used in the comparison. All rows use binary-bootstrap failure target 2^{-16}.

Parameter	$\log N$	$\hat{h}$	h	blocks	mult. depth	q_0	mult.	StC	$\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$	CtS	$\log P$
AESWalsh15	15	256	32	1024	2	36	42×2	61	38×7	50×2	61×4
AESXBoot15	15	256	32	16384	3	38	37×3	36×2	38×7	38×4	38×5

that the baseline is competitive with Lattigo [1], a state-of-the-art library implementation, Appendix C reports measured latency results against the Lattigo XBOOT implementation. We conduct all experiments on an Ubuntu workstation equipped with an AMD Ryzen Threadripper 7970X CPU with 32 physical cores, 64 hardware threads, and 160 GB of RAM. All reported CPU measurements use 64 software threads.

5.1 Parameter Sets

Table 1 lists the CKKS parameters used for the comparison. The Walsh parameters reserve two multiplication levels for the packed Walsh recombination. The XBOOT parameters use the depth-three AES LUT round circuit from the XBOOT-style baseline. The AESXBoot15 row follows the AES-param14 setting from XBOOT [36]. For AESWalsh15, the underlying ring has 16384 complex CKKS slots, and using both real and imaginary lanes gives an effective full-complex lane capacity of 32768. The block count in Table 1 uses both real and imaginary lanes for AESWalsh15. For XBOOT, the number of AES blocks equals the CKKS complex slot count $N/2$. The Q-chain columns list the base limb q_0, followed by the multiplication, StC, $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$, and CtS limbs separately. The columns $\hat{h}$ and h report the dense CKKS secret Hamming weight and the sparse bootstrapping-secret Hamming weight, respectively.

The Walsh packed segment count is 32, with at least 30 required nonzero low/high nibble sign segments; this yields 1024 AES blocks for AESWalsh15 when both real and imaginary lanes are active. The folded Walsh StC mask uses scale logs $18, 20, 23$, grouped into one 61-bit StC limb. The XBOOT parameter set keeps the folded Walsh StC mask disabled.

5.2 Security Considerations

All parameter sets target 128-bit RLWE security [34]. The CKKS ciphertexts and evaluation keys use a dense fixed-Hamming-weight ternary secret with $\hat{h} = 256$. The sparse bootstrapping secret used in the $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$ key-switch path has $h = 32$ for both parameter sets at $N = 2^{15}$.

Summing the active Q-chain columns in Table 1 gives $\log Q = 547$ for AESWalsh15 and $\log Q = 639$ for AESXBoot15. The $\log P$ column records auxiliary special primes used during key switching, giving $\log P = 244$ for AESWalsh15 and $\log P = 190$ for AESXBoot15. Thus the extended-modulus sizes are $\log(PQ) = 791$ and $\log(PQ) = 829$, respectively. We estimated the

Table 2. CPU comparison with the AESXBoot15 baseline. The star in XBOOT* denotes sparse bootstrapping.

Method	Param.	Blocks	Time (s)	Speedup	Error (max/avg)	Rot.
XBOOT	AESXBoot15	16384	210.939	3.88×	$1.73\times10^{-5}/1.36\times10^{-7}$	84
XBOOT*	AESXBoot15	1024	176.838	3.25×	$1.35\times10^{-5}/1.34\times10^{-7}$	43
Walsh	AESWalsh15	1024	54.378	1.00×	$1.41\times10^{-5}/2.75\times10^{-7}$	65

security with the lattice estimator [3], using the dense secret $\hat{h} = 256$ and the sparse secret $h = 32$ for $\log N = 15$.[1] Both parameter sets satisfy the 128-bit target. These totals are below the 853-bit maximum allowed $\log_2(PQ)$ estimated for fixed-$h = 256$ ternary secrets at $N = 2^{15}$, and below the 868-bit uniform-ternary guideline bound reported in recent HE security guidance [14]. For both parameters, the failure probability of the $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$ is 2^{-16}.

5.3 Comparison with XBOOT

We compare our Walsh schedule against the XBOOT-style CKKS-AES baseline. In each full AES round, our construction uses 16 complex bootstraps, one for each AES byte, while XBOOT uses 64 complex bootstraps to refresh the full bit state after the linear layer. The remaining difference is the LUT evaluation itself: XBOOT evaluates the AES S-box as a depth-three LUT after refresh, while our method uses the bootstrap outputs as packed Walsh parity signs and recombines them with one extra post-$\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$ multiplication layer.

The Walsh row in Table 2 counts both full-complex lanes: AESWalsh15 evaluates 1024 AES blocks when both lanes are active. The Walsh row incurs one packed parity refresh for the 30 low/high-nibble Walsh signs of each AES byte, followed by exact nibble-split Walsh decomposition recombination. The XBOOT rows incur the standard LUT S-box evaluation together with the separate round-end XBOOT refresh that cleans the lazy AES state.

Sparse bootstrapping evaluates the bootstrap DFT on only 1024 logical slots inside the same $N = 2^{15}$ ring, using a trace/repack step before CtS. This saves some linear-transform work and rotation keys, but it leaves the dominant XBOOT cost unchanged: the circuit still refreshes 64 packed bit pairs per full round and still pays the same $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$ and key-switching structure.

The measured result is therefore a limited improvement, from 210.939 s to 176.838 s, about 1.19×. In contrast, Walsh reduces the number of round-refresh bootstraps from 64 to 16 by cleaning byte-level packed parity groups, giving 54.378 s and a 3.88× speedup over XBOOT (3.25× over sparse bootstrapping). Thus the main speedup comes from changing the AES evaluation schedule and cutting the bootstrap count by a factor of four, rather than only applying sparse bootstrapping to a smaller transciphering batch.

[1] We utilized the public lattice-estimator code at `https://github.com/malb/lattice-estimator/commit/3e48ef421ec256afddb3e7d2249a77eab6e9ba12`.

Table 3. Contextual comparison of 1024-block AES transciphering with GPU-accelerated CKKS-AES works. Our row uses AESWalsh15. The Bae et al. [10] row uses their Param A setting, and the Kim et al. [33] row uses their GF(16) setting; full parameter details are given in the respective papers. The Kim et al. time is scaled from their reported 512-block GF(16) RTX 4090 throughput. The depth column reports multiplicative depth between consecutive bootstrap operations.

Method	Blocks	Ring N	Hardware	Inter-bootstrap mult. depth	Time
Ours	1024	2^{15}	CPU	2	54.378 s
Bae et al. [10]	1024	2^{16}	RTX 4090	17	3.20 s
Kim et al. [33]	1024	2^{16}	RTX 4090	5	0.160 s

5.4 Comparison with GPU CKKS-AES Works

The second experiment compares against the two recent GPU accelerated CKKS-AES works. The work from [10] focuses on amortized throughput through CI-CKKS conversion, batch-bit bootstrapping (BB-BTS) [12], and lazy SubByte evaluation. The work from [33] shows that CKKS can also be competitive for low-latency AES by using a GF(16) representation to evaluate the S-box more efficiently. A hardware-normalized latency comparison between CPU and GPU implementations is not meaningful without matching backends. However, the multiplicative-depth comparison in Table 3 shows the algorithmic advantage of the Walsh approach: AESWalsh15 uses depth 2 at $N = 2^{15}$, while the related GPU rows use larger depth and larger rings. This indicates that the Walsh schedule is flexible and scalable.

Table 3 compares the algorithmic structure of the three AES-CKKS schedules; the timings are included for context. The GPU rows benefit from mature state-of-the-art HEaaN-family GPU backends: Bae et al. use HEaaN [23], while Kim et al. use HEaaN2 [24]. We therefore use latency as context and focus the comparison on the AES schedule and inter-bootstrap depth. Both GPU works also use orthogonal implementation and CKKS-backend optimizations. Bae et al. [10] use lazy relinearization/rescaling for the SubByte computation. The newer Kim et al. work [33] further improves efficiency by using Grafting [16] and PC-MM-based fused linear maps with shared-a batching [9], which help explain its much lower latency by accelerating repeated StC and CtS-type linear transforms. These techniques optimize the execution of CKKS operations around the circuit, while the Walsh construction changes the LUT representation itself and reduces the inter-bootstrap S-box depth. They are orthogonal to the Walsh S-box method and can in principle be layered with it to further improve execution time.

6 Related Works

Early CKKS-AES Transciphering and CKKS LUTs. The work from [2] presented the first practical SIMD AES transciphering system using CKKS. Their

bit-sliced AES-CTR/GCM keeps one AES bit position per ciphertext, so an AES state uses 128 ciphertexts and no rotations, and delayed BLEACH-style cleanup [26] is applied once per AES round. Chung, Kim, Kim, and Lee [22] follow a different AES transciphering approach: they evaluate large LUTs by rewriting them as multivariate polynomials and amortizing the resulting computation. This is general, but for AES it still incurs S-box cost through polynomial evaluation, so the multiplicative depth, modulus budget, and latency remain high. Newer functional-bootstrapping LUT methods are faster and more flexible [4, 5, 12, 27], but they keep the same pressure point for AES: high multiplicative depth, high latency, and large parameters, often requiring ring dimension $\log N = 16$. Our work keeps the practical CKKS-SIMD setting, but uses lazy XOR parities: the S-box table appears only as plaintext Walsh coefficients, while the encrypted work is parity generation, one packed $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$, and fixed-depth recombination.

XBOOT. XBOOT [36] is the state-of-the-art CKKS-AES follow-up to E2E [2] and the amortized polynomial-LUT route of Chung et al. [22]. It observes that CKKS XOR gates can be replaced by additions when only the least significant bit is needed, and that binary bootstrapping can remove the lazy overflow. For AES-CKKS transciphering, this reduces the AES round depth from the BLEACH-style depth-nine round to a depth-three S-box plus one binary bootstrap per round. Its strength is a clean free-XOR abstraction and a strong full-batch throughput result. Its limitation for our target is that the refresh remains a separate round-level operation and the S-box remains a depth-three LUT, with the implementation optimized for full-slot transciphering. Our Walsh design uses the same lazy-XOR invariant, but it utilizes $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$ to produce the Walsh parity signs needed by the S-box itself. We therefore place the binary refresh differently, which better matches low-latency moderate batches.

High-throughput AES Transciphering Using CKKS. Bae, Cheon, Kang, and Kim [10] optimize CKKS-AES for high-throughput GPU evaluation. They convert between standard CKKS and CI-CKKS [32], combine this with real-context packing, batch-bit bootstrapping (BB-BTS) [12], and lazy relinearization/rescaling for SubByte, reaching sub-millisecond amortized time per block at large batch sizes. The cost is that the LUT computation does not inherit lazy-XOR additions: those operations still contribute to multiplicative depth, leading to larger parameters, with ring dimension 2^{16} in their main throughput setting. The design also pays for conversions between standard CKKS and CI-CKKS. Our approach avoids both bottlenecks: Walsh parities are formed by CKKS additions from the lazy AES state, cleaned together by one packed $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$, and recombined with nibble-split Walsh decomposition using one extra multiplication layer, while the state remains in standard full-complex CKKS throughout. Their GPU, packing, and lazy relinearization/rescaling techniques are orthogonal backend improvements, while our contribution changes the S-box/LUT schedule.

Latency-Aware CKKS-AES. The very recent work of Kim et al. [33] represents AES data in a GF(16) encoding and uses this representation to evaluate the S-box more efficiently, reducing the cost of the LUT computation and improving the latency/throughput tradeoff. A key distinction is that the GF(16) construction is AES-specific: it exploits the finite-field structure of the AES S-box, whereas the Walsh method evaluates general Boolean LUTs on lazy bits. The finite-field encoding can also cause CKKS slot values to grow during lazy arithmetic, creating numerical-stability constraints that require carefully chosen embeddings and signed lifts. They also use Grafting-optimized parameters [16] and PC-MM-based fused linear maps [9] to accelerate the surrounding CKKS bootstrapping and linear-transform work. These optimizations are orthogonal to our Walsh S-box schedule and can in principle be combined with it.

SWIFT. A very recent line of work, represented by SWIFT [18], has a similar goal of separating LUT size from online multiplicative depth. SWIFT exploits the exponential map produced by CKKS functional bootstrapping: for $h(x) = e^{2\pi i x}$, one has $h(\ell x) = h(x)^{\ell}$. By packing the multiples ℓx in SIMD slots before ExpBTS, SWIFT obtains the powers needed for a trigonometric LUT polynomial directly from bootstrapping, then combines them through tensor reconstruction and PCMV. This is similar at a high level to our use of packing to avoid degree-dependent LUT evaluation, but it targets clean discrete CKKS inputs. After a lazy XOR, higher bits may contain overflow rather than zeros, while SWIFT requires combining cleaned bits into an integer before bootstrapping. This representation therefore does not directly apply to lazy XOR bits with overflow. SWIFT also has a fixed-depth reconstruction idea, but its coefficient-combination stage uses PCMV and inner products whose plaintext and ciphertext multiplications consume levels. Our construction instead forms Walsh parities by CKKS additions on lazy bits and cleans those parity signs with one packed $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$. The Walsh table appears only as integer plaintext coefficients, so the plaintext recombination itself consumes no level; in the nibble-split AES case, only the low/high sign products add the fixed ciphertext multiplication layer. This makes our LUT evaluation method specialized to lazy bits in arbitrary circuits.

7 Concluding Remarks

We presented a Walsh-basis method for evaluating LUTs on lazy CKKS bits and applied it to AES-CTR transciphering. The central idea is to move the binary refresh into the LUT evaluation itself: after the lazy parity sums are formed, $\mathsf{EvalMod}_{f_{\mathrm{BinBoot}}}$ cleans the packed Walsh signs, and a fixed-depth recombination recovers the S-box output. For AES, the nibble-split Walsh decomposition extracts only 30 nonconstant low/high-nibble signs per byte and uses one additional multiplication layer for exact recombination.

In our experiments, AESWalsh15 transciphers 1024 AES blocks in 54.378 s, which is 3.25× faster than a sparse-bootstrapping XBOOT variant at the same

1024-block batch size. Future work can combine the depth-2 Walsh schedule with orthogonal CKKS backend and algorithmic techniques, such as Grafting [16], PC-MM-based fused linear transforms [9], lazy relinearization and rescaling, and GPU acceleration.

Acknowledgment

The authors would like to acknowledge the support of the National Science Foundation (Award 2239334).

References

1. Lattigo v6. Online: `https://github.com/tuneinsight/lattigo` (Aug 2024), ePFL-LDS, Tune Insight SA
2. Aharoni, E., Drucker, N., Ezov, G., Kushnir, E., Shaul, H., Soceanu, O.: E2E near-standard and practical authenticated transciphering. Cryptology ePrint Archive (2023)
3. Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. J. Math. Cryptol. **9**(3), 169–203 (2015)
4. Alexandru, A., Kim, A., Polyakov, Y.: General functional bootstrapping using CKKS. In: Advances in Cryptology – CRYPTO 2025. pp. 304–337. Springer (2025)
5. Alexandru, A., Kim, A., Polyakov, Y., Zheng, H.: Sparse hermite interpolation method for discrete-CKKS functional bootstrapping. Cryptology ePrint Archive (2026)
6. Ali, A., Choi, J., Gipson, B., Gorantala, S., Kun, J., Legiest, W., Lim, L., Viand, A., Demissie, M.Z., Zheng, H.: HEIR: A universal compiler for homomorphic encryption (2025), `https://arxiv.org/abs/2508.11095`
7. AntCPLab: XBOOT: Free-xor gates for ckks with applications to transciphering. Online: `https://github.com/AntCPLab/xboot-ckks-transciphering` (2025)
8. Apple Machine Learning Research: Combining Machine Learning and Homomorphic Encryption in the Apple Ecosystem. `https://machinelearning.apple.com/research/homomorphic-encryption` (2024)
9. Bae, Y., Cheon, J.H., Hanrot, G., Park, J.H., Stehlé, D.: Plaintext-ciphertext matrix multiplication and FHE bootstrapping: Fast and fused. In: Advances in Cryptology – CRYPTO 2024. pp. 387–421. Springer (2024)
10. Bae, Y., Cheon, J.H., Kang, M., Kim, T.: High-throughput AES transciphering using CKKS: Less than 1ms. In: Proceedings of the 13th Workshop on Encrypted Computing & Applied Homomorphic Computing. pp. 1–12 (2025)
11. Bae, Y., Cheon, J.H., Kim, J., Stehlé, D.: Bootstrapping bits with CKKS. In: Advances in Cryptology – EUROCRYPT 2024. pp. 94–123. Springer (2024)
12. Bae, Y., Kim, J., Stehlé, D., Suvanto, E.: Bootstrapping small integers with CKKS. In: Advances in Cryptology – ASIACRYPT 2024. pp. 330–360. Springer (2024)
13. Belaïd, S., Bon, N., Boudguiga, A., Sirdey, R., Trama, D., Ye, N.: Further improvements in AES execution over TFHE. IACR Communications in Cryptology **2**(1) (2025)
14. Bossuat, J.P., Cammarota, R., Chillotti, I., Curtis, B.R., Dai, W., Gong, H., Hales, E., Kim, D., Kumara, B., Lee, C., et al.: Security guidelines for implementing homomorphic encryption. IACR Communications in Cryptology **1**(4) (2025)

15. Carlet, C.: Boolean functions for cryptography and coding theory. Cambridge University Press (2021)
16. Cheon, J.H., Choe, H., Kang, M., Kim, J., Kim, S., Mono, J., Noh, T.: Grafting: Decoupled scale factors and modulus in RNS-CKKS. In: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security. pp. 1098–1112 (2025)
17. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: Bootstrapping for approximate homomorphic encryption. In: Advances in Cryptology – EUROCRYPT 2018. pp. 360–384. Springer (2018)
18. Cheon, J.H., Jang, D., Kang, J., Rhee, H.: SWIFT: Shallow and SIMD-aware CKKS functional bootstrapping for low-latency. Cryptology ePrint Archive (2026)
19. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Advances in Cryptology – ASIACRYPT 2017. pp. 409–437. Springer (2017)
20. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In: Advances in Cryptology – ASIACRYPT 2016. pp. 3–33. Springer (2016)
21. Cho, M., Chung, W., Ha, J., Lee, J., Oh, E.G., Son, M.: FRAST: TFHE-friendly cipher based on random S-boxes. IACR Transactions on Symmetric Cryptology (2024)
22. Chung, H., Kim, H., Kim, Y.S., Lee, Y.: Amortized large look-up table evaluation with multivariate polynomials for homomorphic encryption. Cryptology ePrint Archive (2024)
23. CryptoLab: HEaaN library. Online: `https://www.cryptolab.co.kr/en/products-en/heaan-he/` (2022)
24. CryptoLab: HEaaN2 library. Online: `https://www.cryptolab.co.kr/en/products-en/heaan-he/` (2026)
25. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1.2 submission to the CAESAR competition. Submission to the CAESAR Competition (2016)
26. Drucker, N., Moshkowich, G., Pelleg, T., Shaul, H.: BLEACH: Cleaning errors in discrete computations over CKKS. Journal of Cryptology **37**(1), 3 (2024)
27. Dumezy, J., Alexandru, A., Polyakov, Y., Clet, P.E., Chakraborty, O., Boudguiga, A.: Evaluating larger lookup tables using CKKS. Cryptology ePrint Archive (2025)
28. Dworkin, M.J., Barker, E., Nechvatal, J., Foti, J., Bassham, L.E., Roback, E., Dray, J.F.J.: Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, National Institute of Standards and Technology (2001). `https://doi.org/10.6028/NIST.FIPS.197`, `https://doi.org/10.6028/NIST.FIPS.197`
29. Folkerts, L.W., Tsoutsos, N.G.: Proteus: A fully homomorphic authenticated transciphering protocol. Cryptology ePrint Archive (2024)
30. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing. pp. 169–178 (2009)
31. Halevi, S., Shoup, V.: Algorithms in HElib. In: Advances in Cryptology – CRYPTO 2014. pp. 554–571. Springer (2014)
32. Kim, D., Song, Y.: Approximate homomorphic encryption over the conjugate-invariant ring. In: Information Security and Cryptology – ICISC 2018. pp. 85–102. Springer (2018)
33. Kim, T., Lee, J., Noh, T., Cheon, J.H., Hanrot, G.: Latency-aware, high-throughput homomorphic AES evaluation with CKKS. Cryptology ePrint Archive, Paper 2026/1209 (2026), `https://eprint.iacr.org/2026/1209`

34. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Advances in Cryptology – EUROCRYPT 2010. pp. 1–23. Springer (2010)
35. Microsoft Research: Password Monitor: Safeguarding Passwords in Microsoft Edge. https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/ (2021)
36. Niu, C., Huang, Z., Yang, Z., Chen, Y., Kong, L., Hong, C., Wei, T.: XBOOT: Free-XOR gates for CKKS with applications to transciphering. IACR Transactions on Cryptographic Hardware and Embedded Systems **2025**(4), 118–144 (2025)
37. Niu, C., Wei, B., Huang, Z., Yang, Z., Hong, C., Wang, M., Wei, T.: SoK: FHE-friendly symmetric ciphers and transciphering. IACR Transactions on Cryptographic Hardware and Embedded Systems **2025**(3), 583–613 (2025)
38. Shokri, R., Tsoutsos, N.G.: Seamless switching between PBS and WoPBS for scalable TFHE. Cryptology ePrint Archive (2025)
39. Turan, M.S., McKay, K., Kang, J., Kelsey, J., Chang, D.: Ascon-based lightweight cryptography standards for constrained devices: Authenticated encryption, hash, and extendable output functions. NIST Special Publication 800-232, National Institute of Standards and Technology (2025), https://csrc.nist.gov/pubs/sp/800/232/final
40. Wang, R., Ha, J., Shen, X., Lu, X., Chen, C., Wang, K., Lee, J.: Refined TFHE leveled homomorphic evaluation and its application. In: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security. pp. 2309–2323 (2025)

A Reference AES-128-CTR Algorithm

Algorithm 4 presents the conventional AES-128-CTR transciphering computation used as the baseline functionality in this paper. The counter blocks are public, while the AES key is encrypted under the FHE key in the transciphering setting.

Algorithm 4. Standard AES-128-CTR transciphering

Input: AES-128 key K; initial counter block C_0; input stream blocks $Y_0, \ldots, Y_{\ell-1}$
Output: Output blocks $P_0, \ldots, P_{\ell-1}$

procedure AES128Ctr($K, C_0, Y_0, \ldots, Y_{\ell-1}$)
  $(K_0, \ldots, K_{10}) \leftarrow$ KeyExpand128(K)
  for $i = 0, \ldots, \ell - 1$ **do**
    $C_i \leftarrow$ IncrementCounter(C_0, i)
    $S \leftarrow C_i \oplus K_0$
    for $r = 1, \ldots, 9$ **do**
      $S \leftarrow$ SubBytes(S)
      $S \leftarrow$ ShiftRows(S)
      $S \leftarrow$ MixColumns(S)
      $S \leftarrow S \oplus K_r$
    $S \leftarrow$ SubBytes(S)
    $S \leftarrow$ ShiftRows(S)
    $S \leftarrow S \oplus K_{10}$
    $P_i \leftarrow Y_i \oplus S$
  return $P_0, \ldots, P_{\ell-1}$

B Final Binary Refresh Packing

Algorithm 5 gives the generalized final cleanup used after the AES-CTR output XOR. The 128 partially packed lazy bit ciphertexts are packed into complex ciphertexts, refreshed with ComplexBinBoot, and unpacked back to 128 clean output bit ciphertexts. Each refresh packs up to w bit positions; the last packed ciphertext may be only partially filled unless w divides 128. Here ComplexBinBoot cleans the real and imaginary lanes of one packed ciphertext and returns the two cleaned lanes.

Algorithm 5. Final binary refresh packing

Input: Lazy output bit ciphertexts $T_0, \ldots, T_{127}$; active complex slots A; total complex slots n
Output: Clean output bit ciphertexts $P_0, \ldots, P_{127}$

procedure FinalBinaryRefresh(T, A)
  $w \leftarrow \lfloor n/A \rfloor$ ▷ *bit positions packed per refresh*
  $T \leftarrow \text{MaskInactiveSlots}(T, A)$
  for $j = 0, \ldots, \lceil 128/w \rceil - 1$ **do**
    $U \leftarrow 0$
    for $\ell = 0, \ldots, w-1$ such that $jw + \ell < 128$ **do**
      $U \leftarrow U + \text{Rot}(T_{jw+\ell}, \ell A)$
    $(R^0, R^1) \leftarrow \text{ComplexBinBoot}(U)$
    $R \leftarrow R^0 + iR^1$
    for $\ell = 0, \ldots, w-1$ such that $jw + \ell < 128$ **do**
      $P_{jw+\ell} \leftarrow \text{Rot}(R, -\ell A)$
  $P \leftarrow \text{MaskInactiveSlots}(P, A)$
  return $P_0, \ldots, P_{127}$

C Lattigo XBOOT Latency Comparison

Table 4 records AES-CTR latency and maximum output error for our custom scalar C++ CKKS implementation and the Lattigo XBOOT AES-param14 benchmark [1].[2]

Table 4. AES-CTR latency and error comparison with Lattigo XBOOT.

Implementation	Latency	Max error
Ours	210.939 s	1.73×10^{-5}
Lattigo	221.319 s	1.40×10^{-5}

[2] We evaluated the AESParam14 AES-CTR benchmark from the public XBOOT GitHub repository [7], using `examples/transcipher/main.go` with `-mode=benchmark`.