Revisiting homomorphic AES evaluation with Hybrid Byte-Bit Packing

Lu Zhang[1], Xuan Shen[2], Ruida Wang[2], Xianhui Lu[2], Yin Zhang[1], Chao Li[3], Shaojing Fu[1*]

[1] College of Computer Science and Technology, National University of Defense Technology, Changsha, China.
[2]State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS, Beijing, China.
[3] College of Science, National University of Defense Technology, Changsha, China.

*Corresponding author(s). E-mail(s): fushaojing@nudt.edu.cn;

Abstract

Homomorphic AES evaluation is an important benchmark for practical FHE systems. Existing BGV-based AES transciphering schemes typically employ byte-oriented representations, which are well suited for exploiting the algebraic structure of the multiplicative inversion in the SubBytes operation. However, evaluating the affine transformation and subsequent linear layers in this representation introduces additional automorphisms and key-switching operations. In this work, we propose a hybrid byte-bit packing strategy for efficient BGV-based AES transciphering. After performing the multiplicative inversion in the SubBytes operation, we introduce an unpacking procedure to convert the slot packing scheme from byte-oriented to bit-oriented. The affine transformation in SubBytes and subsequent AES linear transformations are unified and evaluated over GF(2) using homomorphic additions, avoiding the key-switching operations involved in the packed byte-oriented evaluation of these linear components. We further optimize the unpacking procedure using a hoisting technique, achieving a 2.5× speedup in unpacking latency compared with the non-hoisted implementation. We implement our approach in HElib and evaluate it under parameters satisfying at least 128-bit security. Experimental results demonstrate per-block latency speedups of 2.6× and 11.7× without bootstrapping, and 2.1× and 3.3× with bootstrapping, on a local machine and a 64-thread server, respectively, compared with the reproduced BGV-based AES baseline under the same software, hardware, and security settings.

Keywords: Fully homomorphic encryption, transciphering, AES, BGV

1 Introduction

Fully Homomorphic Encryption (FHE) [1] enables computation directly over encrypted data and provides a powerful framework for privacy-preserving computation. However, the large ciphertext expansion of FHE makes directly encrypting and transmitting large amounts of data inefficient. Transciphering addresses this problem by combining compact symmetric encryption with FHE. In a typical transciphering scenario, data are encrypted using a symmetric cipher, while only the corresponding secret key is encrypted under FHE. The server then homomorphically evaluates the decryption circuit of the symmetric cipher, transforming the symmetric ciphertext into an FHE ciphertext without revealing the secret key. AES is one of the most widely studied symmetric primitives in transciphering because of its widespread deployment and standardized design.

Existing homomorphic AES transciphering approaches mainly rely on BGV-based exact arithmetic [2], TFHE-based programmable bootstrapping [3, 4], or CKKS-based approximate arithmetic [5]. BGV naturally supports exact arithmetic over finite fields and is therefore well suited to the algebraic structure of AES. TFHE-based approaches evaluate Boolean operations efficiently through programmable bootstrapping, but their limited SIMD parallelism restricts efficient processing of large data batches. CKKS-based approaches exploit large SIMD packing and highly optimized bootstrapping to achieve high throughput. However, evaluating discrete AES computations under CKKS requires additional mechanisms to maintain Boolean semantics, with recent designs typically relying on round-wise bootstrapping for bit refresh.

Despite this natural compatibility, the performance of BGV-based AES evaluation [6, 7] has received comparatively less recent optimization. The computational bottleneck of classical BGV-based AES evaluation is not only limited to the nonlinear multiplicative inversion in the S-box. Evaluating the subsequent affine transformation, ShiftRows, and MixColumns in the byte-oriented representation introduces additional Frobenius and Galois automorphisms and the associated key-switching cost. A fully bit-sliced representation avoids much of this post-inversion overhead by expressing the linear layers over GF_2, but in turn makes the S-box expensive because the multiplicative inversion must be evaluated as a Boolean circuit. Thus, no single representation is simultaneously well suited to both the nonlinear inversion and the subsequent linear layers.

At the same time, the classical GHS15 implementation [7] remains the main reference baseline for BGV-based AES evaluation, although its reported performance was obtained under an earlier HElib environment, older hardware, and different parameter settings. Directly comparing a modern implementation against these historical timings therefore does not provide a fully fair assessment. In this work, we first reproduce and re-parameterize the GHS15 packed BGV implementation under the current HElib environment and parameter sets satisfying at least 128-bit security, and then redesign the ciphertext representation so that the multiplicative inversion is evaluated in a byte-oriented representation while the subsequent AES layers are evaluated in a bit-oriented representation. This allows us to assess the algorithmic benefit of the hybrid design against a consistent and updated baseline. Our main contributions are summarized as follows.

- **Hybrid byte–bit AES evaluation.** To address the mismatch between a single ciphertext representation and the different algebraic requirements of AES layers, we retain the byte-oriented representation for the multiplicative inversion in SubBytes, while converting the state to a bit-oriented representation for the subsequent affine transformation, ShiftRows, MixColumns, and AddRoundKey. These post-inversion operations are jointly evaluated over $\mathbb{F}_2$ using homomorphic additions only, avoiding the additional automorphism-induced key switching required by the conventional packed byte-oriented evaluation.
- **Low-overhead representation conversion.** To prevent the byte-to-bit conversion from becoming a new bottleneck, we optimize the unpacking procedure with hoisting. This reduces the cost of representation conversion while preserving the benefit of the hybrid evaluation. In our implementation, hoisted unpacking achieves a 2.5× speedup over the non-hoisted unpacking, while the hybrid design reduces the post-inversion key-switching count from 13 to 7 per ciphertext path and lowers the bit-capacity consumption.
- **Updated BGV-AES baseline and improved end-to-end evaluation.** To enable a fair assessment of the hybrid BGV-AES implementations against historical results, we reproduce and re-parameterize the classical GHS15 packed BGV implementation under the current HElib environment using parameter sets satisfying at least 128-bit security. Compared with the reproduced baseline, our implementation achieves amortized per-block speedups of 2.6× and 11.7× without bootstrapping, and 2.1× and 3.3× with bootstrapping, on a local machine and a 64-thread server, respectively.

The source code of our implementation is publicly available at https://github.com/LuZhang-coder/aes-transciphering-bgv.

1.1 Related work

BGV-based transciphering. Gentry, Halevi, and Smart presented the first complete implementation of homomorphic AES-128 evaluation based on the BGV scheme [6]. Their updated implementation [7] substantially improved the performance in HElib and considered both non-bootstrapping and bootstrapping settings. The updated implementation processes 120 AES blocks in about 245 seconds without bootstrapping and 180 blocks in about 1050 seconds with bootstrapping, corresponding to approximately 2 and 5.8 seconds per block, respectively [7].

TFHE-based transciphering. TFHE-based approaches exploit programmable bootstrapping to efficiently evaluate AES. Stracovsky et al. [8] reported an evaluation time of 252 seconds for a single AES block. Trama et al. [9] introduced a generic LUT-evaluation operator based on multi-value bootstrapping, requiring about 270 seconds with one thread and 54 seconds with 16 threads. Subsequent works further improved TFHE-based AES evaluation, reducing the single-thread AES-128 latency from 86 seconds in Fregata [10] to 46 seconds in Thunderbird [11], and further to 26.2 seconds with the redesigned circuit bootstrapping of Wang et al. [12]. Recent works explore different optimizations, including batched LUT evaluation [13], alternative plaintext encoding [14]. More recently, the GPU-based implementation [15] reduces

the AES transciphering latency to 0.17 seconds per block. Nevertheless, the limited SIMD parallelism of TFHE constrains throughput for large-scale AES evaluation.

CKKS-based transciphering. CKKS-based transciphering primarily leverages SIMD packing for high-throughput AES evaluation. Aharoni et al. [16] implemented AES-CTR using bit-sliced evaluation and BLEACH-based error cleanup [17], achieving an amortized evaluation time of 37 milliseconds per AES block with 64 threads. XBOOT [18] further reduces the cost of Boolean operations by replacing XOR gates with homomorphic additions and handling the resulting redundant overflow during bootstrapping, achieving approximately 14.4 milliseconds per block in 64 threads. Walsh LUT [19] combines binary bootstrapping with Walsh-basis techniques for LUT evaluation, achieving approximately 53.1 milliseconds per AES block with 64 threads. More recently, [20] proposed separate latency-oriented and high-throughput CKKS designs. On a GPU, the latency-oriented variant evaluates a single AES block in 26 milliseconds, while the high-throughput variant processes 2048 blocks in 137 milliseconds.

FHE-friendly symmetric ciphers. A line of research has proposed FHE-friendly symmetric ciphers, such as Pasta [21], Rubato [22], and HERA [23], designed to reduce multiplicative depth and homomorphic evaluation cost. However, some early FHE-friendly constructions have subsequently been shown to be vulnerable to cryptanalytic attacks [24–27]. These limitations motivate continued interest in homomorphic evaluation of standardized ciphers such as AES.

2 Preliminaries

2.1 Notation

Let $\mathbb{Z}_t = \mathbb{Z}/t\mathbb{Z}$ denote the ring of integers modulo t. Let $\Phi_m(X)$ denote the m-th cyclotomic polynomial of degree $\phi(m)$, where ϕ denotes Euler's totient function. We define the cyclotomic ring $R = \mathbb{Z}[X]/(\Phi_m(X))$, and for an integer modulus t, we write $R_t = R/tR \cong \mathbb{Z}_t[X]/(\Phi_m(X))$. Throughout this paper, p and q denote the plaintext and ciphertext moduli, respectively. We assume that p is prime and $\gcd(p, m) = 1$. Accordingly, plaintexts are represented in R_p, whereas ciphertext components lie in R_q. Let $d = \mathrm{ord}_m(p)$ denote the multiplicative order of p modulo m, i.e., the smallest positive integer d such that $p^d \equiv 1 \pmod{m}$.

We use $\mathbb{F}_p$ to denote the finite field with p elements, and $\mathbb{F}_{p^d}$ to denote its degree-d extension field with p^d elements. The number of SIMD slots induced by the CRT decomposition of R_p is $\ell = \phi(m)/d$. We use $\mu, \nu \in R_p$ to denote plaintexts and bold lowercase letters $\mathbf{c}, \mathbf{c}'$ to denote ciphertexts. Encryption and decryption under secret key s are denoted by $\mathrm{Enc}_s(\mu)$ and $\mathrm{Dec}_s(\mathbf{c})$, respectively. Let $\mathbb{Z}_m^*$ denote the multiplicative group of units modulo m. For $k \in \mathbb{Z}_m^*$, we denote the corresponding Galois automorphism by $\tau_k(f(X)) = f(X^k), f(X) \in R_p$. In particular, for $0 \le i < d$, the Frobenius automorphism acting within each SIMD slot is denoted by $\sigma_i(x) = x^{p^i}$.

2.2 BGV Scheme

The Brakerski–Gentry–Vaikuntanathan (BGV) scheme [2] is a leveled fully homomorphic encryption scheme that supports exact arithmetic over finite rings. It enables homomorphic additions and multiplications on encrypted data, while bootstrapping can be employed to refresh ciphertexts.

- **Encrypt.** Given a plaintext $\mu \in R_p$ and a secret key s, the encryption algorithm outputs a ciphertext $\mathbf{c} = \mathrm{Enc}_s(\mu) = (c_0, c_1) \in R_q^2$ satisfying $c_0 + c_1 s = \mu + pe \pmod q$. The error polynomial e hides the plaintext while remaining sufficiently small to ensure correct decryption.
- **Decrypt.** Given a ciphertext $\mathbf{c} = (c_0, c_1)$, decryption first computes $v = c_0 + c_1 s \pmod q$, and then reduces the result modulo p: $\mathrm{Dec}_s(\mathbf{c}) = v \bmod p = \mu$.
- **HomAdd.** Given two ciphertexts $\mathbf{c} = (c_0, c_1)$ and $\mathbf{c}' = (c_0', c_1')$ encrypting μ and μ', respectively, homomorphic addition is performed component-wise: $\mathbf{c}_{\mathrm{add}} = \mathbf{c} + \mathbf{c}' = (c_0 + c_0',\ c_1 + c_1')$. Upon decryption, the resulting ciphertext yields $\mu + \mu' \in R_p$.
- **HomMult.** Homomorphic multiplication initially produces a degree-two ciphertext $\mathbf{c}_{\mathrm{mult}} = \big(c_0c_0',\ c_0c_1' + c_1c_0',\ c_1c_1'\big)$. Indeed, $c_0c_0' + (c_0c_1' + c_1c_0')s + c_1c_1's^2 = \mu\mu' + p(\mu e' + \mu' e) + p^2 ee' \pmod q$. Therefore, reduction modulo p yields $\mu\mu' \in R_p$. Since the product ciphertext is decryptable with respect to the extended secret-key basis $(1, s, s^2)$, relinearization is subsequently applied to restore the standard degree-one ciphertext form.
- **Key switching and relinearization.** Key switching transforms a ciphertext under one secret key into an equivalent ciphertext under another, without changing the underlying plaintext. Abstractly, $\mathrm{KS}_{s' \to s} : \mathrm{Enc}_{s'}(\mu) \longrightarrow \mathrm{Enc}_s(\mu)$. The transformation is performed using an evaluation key generated in advance. Relinearization can be viewed as a special case of key switching.
- **Modulus switching.** Let $q_H > q_{H-1} > \cdots > q_0$ be a modulus chain. Modulus switching maps a ciphertext at level h as $\mathrm{MS}_{q_h \to q_{h-1}} : R_{q_h}^2 \longrightarrow R_{q_{h-1}}^2, \mathbf{c}^{(h)} \mapsto \mathbf{c}^{(h-1)}$, while preserving the underlying plaintext: $\mathrm{Dec}_s\big(\mathbf{c}^{(h-1)}\big) \equiv \mathrm{Dec}_s\big(\mathbf{c}^{(h)}\big) \equiv \mu \pmod p$.
- **SIMD packing.** BGV supports single-instruction, multiple-data (SIMD) computation by packing multiple data into a single plaintext via the Chinese remainder theorem. Over $\mathbb{F}_p$, the cyclotomic polynomial factors as $\Phi_m(X) = \prod_{i=1}^{\ell} f_i(X),\ \ell = \varphi(m)/d$, where the $f_i(X)$ are distinct irreducible polynomials of degree d. Consequently, $R_p = \mathbb{F}_p[X]/(\Phi_m(X)) \cong \prod_{i=1}^{\ell} \mathbb{F}_p[X]/(f_i(X)) \cong \prod_{i=1}^{\ell} \mathbb{F}_{p^d}$. The ℓ components in this CRT decomposition are referred to as SIMD slots. Under the CRT isomorphism, a single homomorphic operation applies the same arithmetic operation simultaneously to all packed slots.
- **Galois automorphisms.** For each $k \in \mathbb{Z}_m^*$, define $\tau_k(f(X)) = f(X^k), f(X) \in R_p$, as a ring automorphism. If $k \in \langle p \rangle$, write $k \equiv p^i \pmod m$. The induced operation within each SIMD slot $\mathbb{F}_{p^d}$ is the Frobenius automorphism $\sigma_i(x) = x^{p^i}$. If $k \notin \langle p \rangle$, the automorphism τ_k induces a permutation among different SIMD slots.

2.3 BGV Bootstrapping

BGV bootstrapping [28–36] refreshes a ciphertext by homomorphically evaluating the decryption function, thereby reducing the accumulated noise and restoring the available computation capacity. Given a ciphertext $\mathbf{c} \in R_q^2$, the bootstrapping procedure consists of the following steps.

- **Modulus switching.** The input ciphertext is first modulus-switched from q to a suitable modulus $\widetilde{q}$: $\mathbf{c} \longrightarrow \mathbf{c}^{(0)} = (c_0', c_1') \in R_{\widetilde{q}}^2$. This transformation preserves the underlying plaintext while bringing the ciphertext to the modulus required by the recryption procedure.
- **Multiply by encrypted key.** Let $\widetilde{\mathbf{c}}$ denote an encryption of the secret key s under ciphertext modulus Q and an enlarged plaintext modulus. For $\mathbf{c}^{(0)} = (c_0', c_1')$, the homomorphic decryption step evaluates $u = c_0' + c_1' s$. Using the encrypted secret key, this is computed as $\widetilde{\mathbf{c}}_0 = c_1' \widetilde{\mathbf{c}} + (c_0', 0)$, where $\widetilde{\mathbf{c}}_0$ encrypts the intermediate value u in the enlarged plaintext space.
- **Coefficient-to-slot transformation (CoeffToSlot).** The coefficient representation of the intermediate plaintext is transformed into the slot representation: $\mathrm{CoeffToSlot}(u) = \big(L_0(u), \ldots, L_{d-1}(u)\big)$, where $L_i(u) = \sum_{j=0}^{d-1} \kappa_{i,j} \sigma^j(u)$. This transformation exposes the base-field components of the intermediate plaintext so that the subsequent digit-extraction procedure can be evaluated slot-wise.
- **Digit extraction.** After CoeffToSlot, digit extraction operates on the base-p expansion $x = \sum_{i=0}^{d-1} x_i p^i$ of each intermediate slot value. The digit-extraction circuit homomorphically isolates the required digits x_i, which are then combined to recover the plaintext modulo the original plaintext modulus.
- **Slot-to-coefficient transformation (SlotToCoeff).** Finally, SlotToCoeff maps the extracted slot representation back to the coefficient representation. The output is a refreshed ciphertext encrypting the same plaintext as the original ciphertext but with reduced noise.

2.4 AES-128

AES is a standardized block cipher that operates on 128-bit plaintext blocks. In AES-128, the 128-bit state is represented as a 4×4 matrix of bytes,

$$A = \begin{pmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{pmatrix}, \qquad \alpha_{ij} \in \mathbb{F}_{2^8}.$$

AES-128 uses a 128-bit secret key and consists of an initial AddRoundKey operation, followed by nine full rounds and one final round. Each full round consists of

$$\mathsf{SubBytes} \rightarrow \mathsf{ShiftRows} \rightarrow \mathsf{MixColumns} \rightarrow \mathsf{AddRoundKey},$$

whereas the final round omits the MixColumns operation.

- **SubBytes.** The SubBytes transformation applies the AES S-box independently to each byte of the state. For an input byte $a \in \mathbb{F}_{2^8}$, the S-box first computes the multiplicative inverse $a^{-1} \in \mathbb{F}_{2^8}$, with 0 mapped to 0. Let $\mathbf{x} = (x_0, x_1, \ldots, x_7)^T \in \mathbb{F}_2^8$ denote the bit representation of a^{-1}. The affine transformation then produces the output vector $\mathbf{y} = (y_0, y_1, \ldots, y_7)^T \in \mathbb{F}_2^8$, where for $0 \leq i < 8$, $y_i = x_i \oplus x_{(i+4) \bmod 8} \oplus x_{(i+5) \bmod 8} \oplus x_{(i+6) \bmod 8} \oplus x_{(i+7) \bmod 8} \oplus c_i$ and $(c_7 c_6 \cdots c_0)_2 = \{63\}_{16}$. Thus, the AES S-box consists of a multiplicative inversion over $\mathbb{F}_{2^8}$ followed by a fixed affine transformation over $\mathbb{F}_2$. Applying the S-box to all 16 bytes produces the transformed AES state, which is then processed by ShiftRows.
- **ShiftRows.** The ShiftRows operation cyclically shifts the bytes in each row of the state. Row r is cyclically shifted to the left by r positions. Thus,

$$\mathsf{ShiftRows}(A) = \begin{pmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{10} \\ \alpha_{22} & \alpha_{23} & \alpha_{20} & \alpha_{21} \\ \alpha_{33} & \alpha_{30} & \alpha_{31} & \alpha_{32} \end{pmatrix}.$$

- **MixColumns.** The MixColumns operation independently transforms each column of the AES state using a fixed linear transformation over $\mathbb{F}_{2^8}$. For the j-th column,

$$\begin{pmatrix} \beta_{0j} \\ \beta_{1j} \\ \beta_{2j} \\ \beta_{3j} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} \alpha_{0j} \\ \alpha_{1j} \\ \alpha_{2j} \\ \alpha_{3j} \end{pmatrix},$$

  where all entries and arithmetic are defined over $\mathbb{F}_{2^8}$.
- **AddRoundKey.** The AddRoundKey operation combines the current AES state with the corresponding round key using bitwise XOR. Let A denote the state and $K^{(r)}$ the round key of round r. Then $A \leftarrow A \oplus K^{(r)}$. The round keys are derived from the original 128-bit AES key using the AES-128 key schedule. After the initial AddRoundKey, nine full rounds are executed, followed by a final round consisting of SubBytes, ShiftRows, and AddRoundKey.

3 Hybrid Byte–Bit Homomorphic AES Evaluation

In this section, we first revisit the main ciphertext representations used in existing BGV-based homomorphic AES evaluations and analyze their computational trade-offs. Based on this analysis, we introduce a hybrid byte–bit representation that evaluates the multiplicative inversion of SubBytes in a byte-oriented representation and the subsequent affine and linear transformations in a bit-oriented representation.

3.1 Existing Homomorphic AES Representations

Existing BGV-based homomorphic AES implementations mainly differ in how the AES state is distributed across ciphertexts. The choice of representation directly affects the cost of the S-box evaluation and the linear transformations in each AES round. We briefly review three representative layouts below.

Packed representation of the AES state. Assume $8 \mid d$, so that $\mathbb{F}_{2^8} \subseteq \mathbb{F}_{2^d}$. Each plaintext slot can then store one AES byte. If a ciphertext contains ℓ plaintext slots, up to $\lfloor \frac{\ell}{16} \rfloor$ AES states can be packed into a single ciphertext. The 16 bytes of one state are represented as

$$\mathbf{a} = (\alpha_{00}, \alpha_{10}, \alpha_{20}, \alpha_{30}, \alpha_{01}, \alpha_{11}, \alpha_{21}, \alpha_{31}, \alpha_{02}, \alpha_{12}, \alpha_{22}, \alpha_{32}, \alpha_{03}, \alpha_{13}, \alpha_{23}, \alpha_{33}).$$

In this representation, the same homomorphic operation is applied simultaneously to all packed AES states. AddRoundKey is implemented by ciphertext addition. The AES S-box is a nonlinear substitution function defined over the finite field $\mathbb{F}_{2^8}$. For each byte $\alpha \in \mathbb{F}_{2^8}$, it is constructed by first computing the multiplicative inverse and then applying a fixed affine transformation T over $\mathbb{F}_{2^8}$: $S(x) = T(\alpha^{-1})$.

Since $\alpha^{-1} = x^{254}$ for every nonzero $\alpha \in \mathbb{F}_{2^8}$, the nonlinear part of the S-box can be written as

$$\begin{aligned}
&\alpha^2 = \mathrm{Frob}_1(\alpha),\ \alpha^3 = \alpha \cdot \alpha^2,\\
&\alpha^{12} = \mathrm{Frob}_2(\alpha^3),\ \alpha^{14} = \alpha^{12} \cdot \alpha^2,\\
&\alpha^{15} = \alpha^{12} \cdot \alpha^3,\ \alpha^{240} = \mathrm{Frob}_4(\alpha^{15}),\ \alpha^{254} = \alpha^{240} \cdot \alpha^{14},
\end{aligned}$$

which costs four ciphertext multiplications and three nontrivial Frobenius automorphisms. The affine part of the AES S-box can also be expressed using Frobenius conjugates. Specifically, there exist constants $\gamma_0, \ldots, \gamma_7, \delta \in \mathbb{F}_{2^8}$ such that $T(y) = \delta + \sum_{i=0}^{7} \gamma_i y^{2^i}$.

ShiftRows corresponds to a fixed permutation of the packed byte positions and is therefore implemented using Galois automorphisms that permute the SIMD slots. MixColumns is subsequently evaluated by combining slot permutations, ciphertext additions, and multiplications by plaintext constants.

In the packed implementation of Gentry, Halevi, and Smart, one AES round has multiplicative depth approximately four and requires 20 key-switching operations along one ciphertext path: seven associated with the multiplicative inversion, seven with the affine transformation, and six with the combined ShiftRows/MixColumns computation [6, 7].

Byte-sliced representation. An alternative is to represent the 16 byte positions of the AES state by 16 separate ciphertexts $\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_{15}$, where each ciphertext packs the same byte position from ℓ independent AES states.

AddRoundKey and SubBytes are evaluated independently on each of the 16 ciphertexts in the same way as the packed representation. The main difference appears in ShiftRows and MixColumns. Since different byte positions are already stored in separate ciphertexts, ShiftRows directly reorders these ciphertexts and therefore requires no slot automorphism. MixColumns can be realized through linear combinations of the corresponding byte-position ciphertexts and multiplications by fixed plaintext constants.

The byte-sliced implementation retains the same approximately four-level round depth, but reduces the number of key-switching operations from 20 to 14, corresponding to the seven operations used in the inversion and the seven Frobenius automorphisms used in the affine transformation [7]. The trade-off is that 16 ciphertexts must be processed instead of one, which increases the wall-clock work and memory consumption.

Bit-sliced representation. A more fine-grained representation stores each of the 128 bits of the AES state in a separate ciphertext,

$$\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_{127}.$$

Each ciphertext packs the same bit position from ℓ independent AES states, so that one collection of 128 ciphertexts represents ℓ AES blocks in parallel.

Under this representation, the S-box is implemented using the Boyar–Peralta binary circuit [37]. AddRoundKey is implemented using ciphertext additions, while ShiftRows is merely a permutation of the 128 ciphertext indices. MixColumns is an $\mathbb{F}_2$-linear transformation and can likewise be evaluated using ciphertext additions only. Therefore, the only nonlinear component is SubBytes.

From the viewpoint of multiplicative depth, a bit-sliced AES round can be evaluated in approximately four levels. However, this representation requires 128 ciphertexts per AES state position, and the large number of additions in the Boolean S-box circuit also contributes substantial noise and computational overhead. In the original implementation, the effective level consumption was observed to be approximately 5–10 levels per round, despite the smaller theoretical multiplicative depth [6].

Comparison and motivation. Table 1 summarizes the main differences among the three representations. The comparison shows that no single representation is well suited to all AES operations. Byte-oriented representations efficiently support the extension-field arithmetic required by the multiplicative inversion, whereas a bit-oriented representation is better suited to the affine transformation and the subsequent AES linear layers.

The above trade-off motivates a hybrid representation rather than using a single layout throughout the entire AES round. In particular, the multiplicative inversion in SubBytes is naturally evaluated over $\mathbb{F}_{2^8}$, whereas the affine transformation and the subsequent AES linear layers are fixed transformations over $\mathbb{F}_2$.

Our approach therefore retains the byte-oriented representation for the multiplicative inversion and converts the state into a bit-oriented representation immediately afterwards. The affine transformation, ShiftRows, MixColumns, and AddRoundKey can then be jointly evaluated over $\mathbb{F}_2$ before the state is converted back to the byte-oriented representation for the next-round inversion.

3.2 Design Overview

Our hybrid evaluation follows the standard structure of AES-128 while switching ciphertext representations within each round. After the initial AddRoundKey, the first

Table 1 Comparison of ciphertext representations for BGV-based homomorphic AES evaluation.

Representation	**S-box inversion**	**Post-inversion linear layers**	**Main costs**
Packed	$\mathbb{F}_{2^8}$	Field-level operations with automorphisms	20 key switches
Byte-sliced	$\mathbb{F}_{2^8}$	Field-level operations with automorphisms	14 key switches
Bit-sliced	Boolean circuit	$\mathbb{F}_2$ additions	Expensive Boolean S-box
Hybrid (ours)	$\mathbb{F}_{2^8}$	$\mathbb{F}_2$ additions	7 key switches + representation conversion

nine rounds are evaluated using the same hybrid round procedure, while the final round uses the same procedure with MixColumns omitted. Figure 2 summarizes the organization of the complete ten-round AES evaluation.

Figure 1 illustrates the resulting per-round evaluation flow used in each of the first nine AES rounds. Each round starts from 16 byte-oriented ciphertexts, in which the multiplicative inversion of SubBytes is evaluated slot-wise over $\mathbb{F}_{2^8}$. The resulting ciphertexts are then converted into 128 bit-oriented ciphertexts through Unpack. In the bit-oriented representation, the affine transformation of SubBytes, ShiftRows, MixColumns, and AddRoundKey are combined into a single binary affine transformation and evaluated using homomorphic additions over $\mathbb{F}_2$. Finally, Pack converts the 128 bit-oriented ciphertexts back into 16 byte-oriented ciphertexts for the multiplicative inversion of the next round.

Since Unpack requires several Frobenius automorphisms applied to the same input ciphertext, we further reduce the representation-conversion overhead using hoisting. The common ciphertext decomposition required by key switching is performed once and reused across the automorphisms involved in the unpacking procedure. We now formalize the two ciphertext representations used in the hybrid evaluation. Let an AES state be represented as $A = (\alpha_{i,j})_{0 \le i,j < 4},\ \alpha_{i,j} \in \mathbb{F}_{2^8}$. In the byte-oriented representation, the 16 byte positions of the AES state are stored in 16 byte-oriented ciphertexts,

$$\mathbf{c}^{\text{byte}} = \left(\mathbf{c}_0^{\text{byte}}, \ldots, \mathbf{c}_{15}^{\text{byte}}\right),$$

where each ciphertext packs the same byte position from multiple AES states across the SIMD slots. For the bit-oriented representation, each byte $\beta \in \mathbb{F}_{2^8}$ is expanded with respect to a fixed basis $\{1, \zeta, \ldots, \zeta^7\}$ as $\beta = \sum_{k=0}^{7} b_k \zeta^k,\ b_k \in \mathbb{F}_2$. Accordingly, the 16 byte-oriented ciphertexts are transformed into 128 bit-oriented ciphertexts,

$$\mathbf{c}^{\text{bit}} = \left(\mathbf{c}_0^{\text{bit}}, \ldots, \mathbf{c}_{127}^{\text{bit}}\right),$$

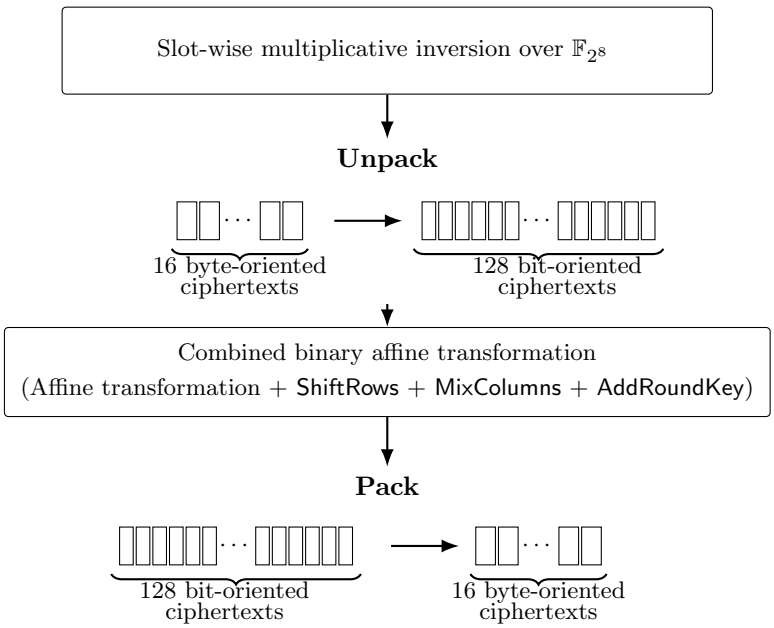

Fig. 1 Representation flow used in each of the first nine rounds of the proposed hybrid byte–bit homomorphic AES evaluation.

where each ciphertext stores one fixed bit position across the packed AES states. The conversion from the byte-oriented representation to the bit-oriented representation is denoted by

$$\mathsf{Unpack} : (\mathbb{F}_{2^8})^{16} \longrightarrow (\mathbb{F}_2)^{128}.$$

Once the state is converted into the bit-oriented representation, the remaining operations of a non-final AES round can be composed into a single linear transformation over $\mathbb{F}_2$. Let $\mathbf{x} \in \mathbb{F}_2^{128}$ denote the bit-oriented state. For round r, we write this transformation as

$$\mathcal{T}_r(\mathbf{x}) = M_r\mathbf{x} \oplus \mathbf{c}_r,$$

where $M_r \in \mathbb{F}_2^{128\times 128}$ captures the linear dependencies introduced by the affine part of SubBytes, ShiftRows, and MixColumns, and $\mathbf{c}_r \in \mathbb{F}_2^{128}$ incorporates the corresponding constant terms and the round key. The composed mapping is evaluated directly on the 128 bit-oriented ciphertexts. After this transformation, the state is converted back to the byte-oriented representation using

$$\mathsf{Pack} : (\mathbb{F}_2)^{128} \longrightarrow (\mathbb{F}_{2^8})^{16},$$

so that the multiplicative inversion of the next AES round can again be evaluated over $\mathbb{F}_{2^8}$.

Accordingly, Rounds 1–9 follow the same hybrid evaluation pattern: multiplicative inversion in the byte-oriented representation, followed by *Unpack*, the combined bit-oriented transformation $\mathcal{T}_r$, and *Pack*. The final round follows the same structure,

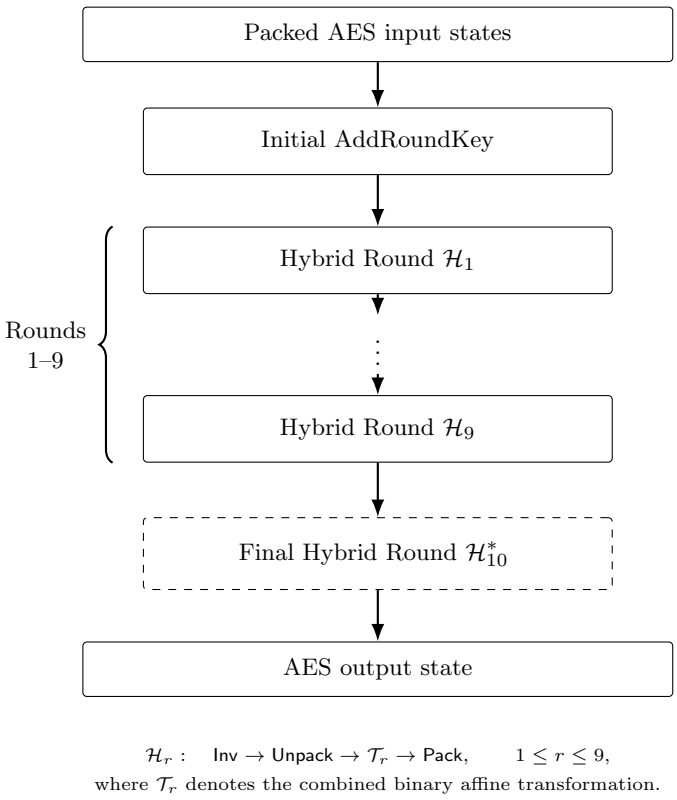

Fig. 2 Overall evaluation structure of the proposed hybrid byte–bit AES. Rounds 1–9 use the same hybrid round structure described in Fig. 1, whereas the final round omits MixColumns.

except that $\mathcal{T}_r$ excludes the MixColumns operation. Figure 2 summarizes how this hybrid evaluation pattern is organized across the complete AES computation.

3.3 Byte-Oriented Multiplicative Inversion

We first evaluate the multiplicative inversion of SubBytes while retaining the byte-oriented SIMD representation. For each AES byte position $t \in \{0, \ldots, 15\}$, one ciphertext

$$\mathbf{c}_t = (\alpha_{t,0}, \alpha_{t,1}, \ldots, \alpha_{t,\ell-1}), \qquad \alpha_{t,s} \in \mathbb{F}_{2^8},$$

packs the same byte position from ℓ AES states across the SIMD slots. Thus, the ℓ packed AES states are represented by 16 byte-oriented ciphertexts.

The multiplicative inversion is applied independently to every SIMD slot of each ciphertext. For each nonzero byte $\alpha \in \mathbb{F}_{2^8}$, $\alpha^{-1} = \alpha^{254}$. As described in the packed representation of Section 3.1, this exponentiation can be evaluated by combining homomorphic multiplications with Frobenius automorphisms. The same slot-wise inversion procedure is applied to each of the 16 byte-position ciphertexts, while all ℓ packed AES states are processed in parallel across the SIMD slots.

Retaining the byte-oriented representation beyond the inversion would require the subsequent affine transformation and AES linear layers to be evaluated using additional field-level automorphisms and the associated key-switching operations. Hence, the multiplicative inversion is the last operation evaluated in the byte-oriented $\mathbb{F}_{2^8}$ domain before the computation switches to the bit-oriented representation.

3.4 Unpacking with Hoisting

After the multiplicative inversion, each SIMD slot contains an element $\beta = \sum_{i=0}^{7} b_i \zeta^i \in \mathbb{F}_{2^8}$, $b_i \in \mathbb{F}_2$. Our goal is to convert this byte-oriented ciphertext into eight bit-oriented ciphertexts $\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_7$, such that the slot corresponding to β in $\mathbf{c}_i$ contains b_i. Hence, the 16 byte-oriented ciphertexts representing the packed AES states are transformed into 128 bit-oriented ciphertexts. Following the unpacking procedure [38], for every $0 \leq i < 8$ we define the linear coefficient-extraction map

$$L_i : \mathbb{F}_{2^8} \longrightarrow \mathbb{F}_2, \qquad L_i(\beta) = b_i.$$

Each L_i can be expressed as a linearized polynomial

$$L_i(\beta) = \sum_{j=0}^{7} \kappa_{i,j} \beta^{2^j}, \qquad \kappa_{i,j} \in \mathbb{F}_{2^8}.$$

Therefore, for a byte-oriented ciphertext $\mathbf{c}$, the i-th bit-oriented output ciphertext is computed as

$$\mathbf{c}_i = \sum_{j=0}^{7} \kappa_{i,j} \sigma_j(\mathbf{c}), \qquad 0 \leq i < 8.$$

The result is a collection of eight packed ciphertexts representing the eight bit positions of every encrypted AES byte. Applying the same procedure to all 16 byte-position ciphertexts produces the 128 bit-oriented ciphertexts required by the subsequent combined transformation $\mathcal{T}_r$.

This transformation changes only the representation of the encrypted state. Semantically, the information contained in the original $\mathbb{F}_{2^8}$ slots is preserved, but it is redistributed into separate $\mathbb{F}_2$-oriented ciphertexts. Subsequent operations can therefore be expressed directly as binary linear transformations.

A direct implementation of the above unpacking procedure requires several Frobenius automorphisms of the same input ciphertext. Although applying the ring automorphism itself is relatively inexpensive, each automorphism is normally followed by a key-switching operation. Repeating the complete key-switching procedure independently for all Frobenius maps introduces substantial redundant work.

To reduce this conversion cost, we integrate hoisting [39] into the AES-specific unpacking procedure, which reorders these operations by decomposing the ciphertext component once and reusing the decomposition across multiple automorphisms. Let the input ciphertext be $\mathbf{c} = (c_0, c_1)$ and decompose $c_1 = \sum_k q_k \Delta_k$. For each required Frobenius automorphism σ_j, the automorphism is applied directly to the decomposed

digits,

$$q_k^{(j)} = \sigma_j(q_k).$$

Since σ_j fixes the decomposition basis, we have

$$\sigma_j(c_1) = \sigma_j\left(\sum_k q_k \Delta_k\right) = \sum_k \sigma_j(q_k)\Delta_k.$$

Hence, the same ciphertext decomposition can be reused across the Frobenius automorphisms required to construct the eight bit-oriented outputs associated with one AES byte. The direct and hoisted implementations can be summarized as

$$\begin{aligned} \text{Direct:} \quad & \{\text{Frob}_j \to \text{Decomp} \to \text{KeySwitch}_j\}_j, \\ \text{Hoisted:} \quad & \text{Decomp} \to \{\text{Frob}_j \to \text{KeySwitch}_j\}_j. \end{aligned}$$

Hoisting does not change the unpacking transformation or the number of required Frobenius automorphisms. Instead, it amortizes the expensive ciphertext decomposition across the automorphisms derived from the same byte-position ciphertext. The effectiveness of this optimization is evaluated separately in Section 4.3.

3.5 Fused Binary Affine Transformation

After the multiplicative inversion and byte-to-bit unpacking, the AES state is represented by 128 bit-oriented ciphertexts. At the plaintext level, these ciphertexts correspond to a binary state vector $\mathbf{x} = (x_0, \ldots, x_{127})^T \in \mathbb{F}_2^{128}$, where each coordinate x_i represents one fixed bit position of the AES state, and the corresponding bit-oriented ciphertext packs that position from multiple AES states across the SIMD slots.

In this bit-oriented representation, the remaining operations of a full AES round—the affine part of SubBytes, ShiftRows, MixColumns, and AddRoundKey— can be expressed jointly as an affine transformation over $\mathbb{F}_2$. Let $A_{\text{AES}} \in \mathbb{F}_2^{8\times 8}$ denote the linear part of the affine transformation applied to one AES byte. Since the same transformation is applied independently to all 16 byte positions, its action on the 128-bit state is represented by $A_{\text{SB}} = \text{diag}\,(A_{\text{AES}}, \ldots, A_{\text{AES}}) \in \mathbb{F}_2^{128\times 128}$, where the block diagonal contains 16 identical 8×8 blocks.

Let $P_{\text{SR}} \in \mathbb{F}_2^{128\times 128}$ denote the permutation matrix induced by ShiftRows, i.e., it only reorders the 128 bit positions of the AES state without changing their values. Let $M_{\text{MC}} \in \mathbb{F}_2^{128\times 128}$ denote the binary linear representation of MixColumns. Although MixColumns is conventionally defined over $\mathbb{F}_{2^8}$ on each 4-byte column, multiplication by the fixed field constants can be expanded into fixed linear maps over $\mathbb{F}_2$, yielding a 128-bit binary linear transformation. If $\mathbf{a} \in \mathbb{F}_2^{128}$ is the affine constant of SubBytes and $\mathbf{k}_r \in \mathbb{F}_2^{128}$ is the r-th round key, then a full round after inversion can be written as

$$\begin{aligned} \mathbf{y} &= M_{\text{MC}} P_{\text{SR}} \big(A_{\text{SB}}\mathbf{x} + \mathbf{a}\big) + \mathbf{k}_r \\ &= M_r \mathbf{x} + \mathbf{b}_r, \end{aligned}$$

where $M_r = M_{\mathrm{MC}} P_{\mathrm{SR}} A_{\mathrm{SB}}$ and $\mathbf{b}_r = M_{\mathrm{MC}} P_{\mathrm{SR}} \mathbf{a} + \mathbf{k}_r$. Since the linear part is identical for the first nine AES rounds, we simply write $M = M_{\mathrm{MC}} P_{\mathrm{SR}} A_{\mathrm{SB}}$, and precompute it once. Only the additive vector $\mathbf{b}_r$ depends on the round key. For the final round, where MixColumns is omitted, the corresponding affine map is $\mathbf{y} = M_{\mathrm{final}}\mathbf{x} + \mathbf{b}_{\mathrm{final}}$, with $M_{\mathrm{final}} = P_{\mathrm{SR}} A_{\mathrm{SB}}$ and $\mathbf{b}_{\mathrm{final}} = P_{\mathrm{SR}}\mathbf{a} + \mathbf{k}_{10}$. For each output bit position $i \in \{0, \ldots, 127\}$, the i-th row of M specifies which input bit positions contribute to that output. Accordingly,

$$y_i = b_{r,i} + \sum_{j:M_{ij}=1} x_j, \qquad 0 \le i < 128,$$

where $j \in \{0, \ldots, 127\}$ indexes the input bit positions and $M_{ij} = 1$ indicates that the input bit x_j contributes to the i-th output bit.

At the ciphertext level, let $\mathbf{c}_j$ denote the bit-oriented ciphertext corresponding to input bit position j. The ciphertext corresponding to output bit position i is then evaluated as

$$\mathbf{c}'_i = \mathsf{b}_{r,i} + \sum_{j:M_{ij}=1} \mathbf{c}_j, \qquad 0 \le i < 128.$$

Since all entries of M lie in $\mathbb{F}_2$, the fused transformation is implemented using homomorphic additions of the selected input ciphertexts only. The effects of the S-box affine transformation, ShiftRows, and MixColumns are already absorbed into the fixed matrix M, while the affine constant and the AddRoundKey contribution are incorporated into $\mathbf{b}_r$. Consequently, once the state has been converted to the bit-oriented representation, these operations require no ciphertext–ciphertext multiplication, Frobenius automorphism, or key switching.

The final round is evaluated in the same manner using M_{final} and $\mathbf{b}_{\mathrm{final}}$, with the MixColumns component omitted. Figure 3 illustrates the ciphertext-level realization of this fused affine transformation.

3.6 Key-Switching Overhead Analysis

Although byte-sliced and bit-sliced variants were explored in [6, 7], only the packed variant was re-implemented in the updated HElib-based work [7]. We therefore use the packed implementation as our primary quantitative baseline. Since the two schemes use different ciphertext layouts, we compare key-switching overhead on a per-ciphertext basis rather than the aggregate cost over all 16 ciphertexts.

In the packed GHS15 implementation [7], processing one ciphertext through an AES round requires 20 key-switching operations. Seven are associated with the multiplicative inversion in SubBytes, while the remaining 13 occur after inversion: seven in the affine transformation and six in ShiftRows and MixColumns.

Our design retains the same byte-oriented inversion. For each byte-oriented ciphertext, Unpack requires seven nontrivial Frobenius automorphisms, each followed by key switching. After Unpack, the affine transformation, ShiftRows, MixColumns, and AddRoundKey are evaluated in the bit-oriented representation using homomorphic additions only and require no further automorphism-induced key switching. Therefore, on a per-ciphertext basis, the post-inversion key-switching count is reduced from 13 to

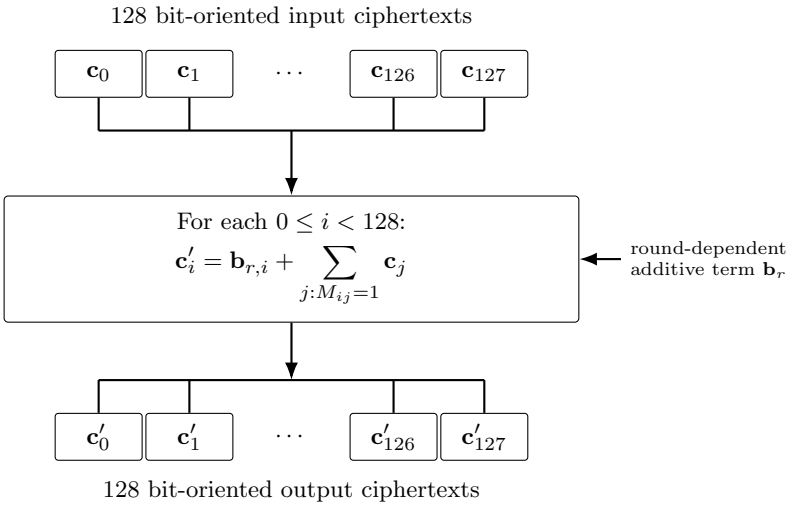

Fig. 3 Ciphertext-level realization of the fused binary affine transformation. Each output ciphertext is obtained by homomorphically summing the selected input ciphertexts and adding the round-dependent term $\mathbf{b}_{r,i}$.

7, and the corresponding total count along one ciphertext path is reduced from 20 to 14. The seven post-inversion key switches in our design arise from Frobenius automorphisms applied to the same input ciphertext during Unpack. The expensive ciphertext decomposition can therefore be shared through hoisting. The resulting per-ciphertext key-switching costs are summarized in Table 2.

Table 2 Per-round per-ciphertext key-switching cost of the implemented packed GHS15 baseline and our hybrid design.

Method	**Inversion**	**Combined binary affine transformation**	**Total/ciphertext**	**Hoisting**
GHS15 packed [7]	7	13	20	No
Ours	7	7	14	Yes

The reduction in key-switching operations also changes the noise-growth profile along each ciphertext path. This reduced noise growth is reflected in the non-bootstrapping parameter selection in Section 4.2: our scheme completes all ten AES rounds with $m = 53261$, while the reproduced GHS15 baseline requires the larger $m = 65281$ to maintain a sufficient noise budget.

4 Implementation and Experimental Evaluation

4.1 Experimental Setup

We implement both the BGV-based packed AES transciphering scheme [7] and our hybrid byte–bit scheme in HElib under the same software and hardware environments. To ensure a fair comparison, we also reproduce the scheme in [7] under the current HElib implementation. Since the original parameter sets do not simultaneously satisfy the required security level and provide sufficient noise budget for the complete ten-round AES evaluation, we re-select the parameters for the reproduced GHS15 baseline accordingly.

The local experiments are performed on a laptop equipped with an Intel Core i7-1260P processor and 16 GB of memory running Windows 11. Unless otherwise specified, the local experiments use a single execution thread.

The server experiments are performed on an AMD EPYC 9T34 processor with 256 GB of memory running Ubuntu 22.04.5 LTS. Up to 64 threads are used in the parallel experiments.

4.2 Parameter Selection

For the GHS15 baseline, we increase the cyclotomic parameter from $m = 53261$ to $m = 65281$ in the non-bootstrapping setting and from $m = 28679$ to $m = 53261$ in the bootstrapping setting. All parameter sets used in our experiments provide at least 128-bit estimated security. In contrast, our hybrid implementation exhibits a smaller accumulated bit-capacity loss during the AES evaluation, allowing the non-bootstrapping configuration to complete all ten rounds with $m = 53261$.

The two schemes have different SIMD batch sizes due to their ciphertext layouts and parameter sets. We therefore use the fully packed configuration of each implementation and report amortized latency per AES block. In the non-bootstrapping setting, our scheme processes 1920 blocks, compared with 84 blocks for the GHS15 baseline; with bootstrapping, the corresponding batch sizes are 1920 and 360 blocks, as shown in Table 3.

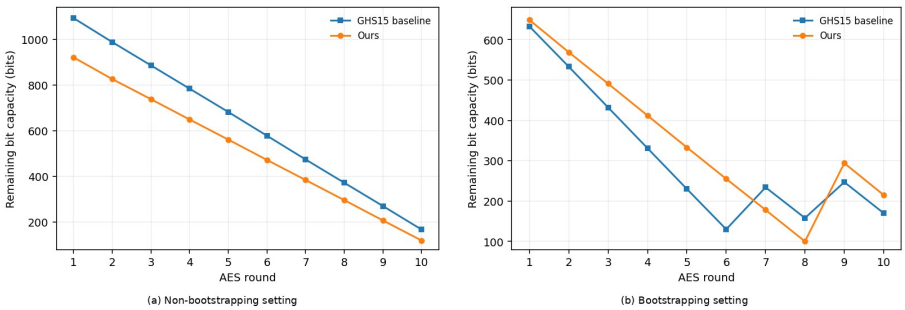

Fig. 4 Remaining bit capacity throughout the AES evaluation for the reproduced GHS15 baseline and our hybrid scheme. Since the parameter sets are independently selected to satisfy the same security target, the comparison focuses on the rate of bit-capacity consumption rather than the absolute capacity values.

Table 3 Parameter sets used for the reproduced GHS15 baseline and our implementation. No-BS and BS denote the non-bootstrapping and bootstrapping settings, respectively. Here, m is the cyclotomic parameter, $\phi(m)$ is Euler's totient function, and $\log_2 Q$ denotes the bit length of the ciphertext modulus Q immediately before the first AES round begins. Blocks denotes the number of AES blocks processed simultaneously. All parameter sets satisfy the 128-bit security requirement.

Setting	**Method**	m	$\phi(m)$	$\log_2 Q$	**Blocks**
No-BS	GHS15 [7]	65281	32256	1093	84
No-BS	Ours	53261	46080	921	1920
BS	GHS15 [7]	53261	46080	633	360
BS	Ours	53261	46080	649	1920

Figure 4 illustrates the consumption of the remaining bit capacity throughout the AES evaluation, which reflects the noise growth behavior of the corresponding scheme. In the non-bootstrapping setting, the GHS15 baseline consumes approximately 102.9 bits of capacity per round on average, whereas our hybrid design consumes approximately 89.1 bits per round. Since the two parameter sets are independently selected to satisfy the same security target, the absolute bit-capacity values are not directly compared. Instead, the slower round-wise decrease indicates lower noise consumption in our design.

A similar behavior can be observed in the bootstrapping setting. Although the bit capacity is refreshed during the evaluation, our implementation exhibits a slower capacity decrease between refresh points. In addition to the reduced noise growth of the hybrid byte–bit evaluation, our bootstrapping implementation incorporates the optimization of GIKV23 [29], whereas the reproduced GHS15 baseline follows the original evaluation strategy.

4.3 Performance of Hoisted Unpacking

The byte-to-bit conversion requires a family of Frobenius automorphisms applied to the same input ciphertext. In a direct implementation, the ciphertext decomposition required by key switching is repeated independently for these automorphisms. The hoisting technique described in Section 3.4 instead performs the common decomposition once and reuses it across the required automorphisms.

To evaluate the effectiveness of this optimization, we compare the standard and hoisted unpacking implementations under the bootstrapping parameter set in Table 3 on the local machine described in Section 4.1. Both implementations execute the same complete ten-round AES computation, while differing only in whether the unpacking procedure uses hoisting. We report the accumulated running time spent in the byte-to-bit unpacking stages over all ten AES rounds. Table 4 reports the corresponding latency. Hoisting reduces the unpacking time from 267.9 s to 109.1 s, corresponding to a 2.5× speedup.

Table 4 Accumulated byte-to-bit unpacking time over the complete ten-round AES evaluation under the bootstrapping parameter set on the local machine.

Method	Unpacking time	Speedup
Standard unpacking	267.9 s	1.0×
Hoisted unpacking	109.1 s	2.5×

4.4 End-to-End AES Performance and Analysis

We next evaluate the complete ten-round AES computation. The baseline and our implementation are executed on the same local machine and the same 64-thread server using the parameter sets from Table 3. For a fair comparison, the GHS15 baseline is re-evaluated in the current HElib environment using the updated parameter sets in Table 3, rather than using the historical running times reported in GHS15 [7]. Table 5 reports both the number of AES blocks processed in one evaluation and the amortized latency per block.

Table 5 End-to-end AES evaluation performance of the reproduced GHS15 baseline and our implementation. Local and 64-thread server results report the amortized latency per AES block. Speedup is measured relative to the reproduced GHS15 baseline under the same setting.

Setting	Method	Blocks	Local	Speedup	64-thread Server	Speedup
No-BS	GHS15 [7]	84	1.17 s	1.0×	528 ms	1.0×
	Ours	1920	452 ms	2.6×	45.2 ms	11.7×
BS	GHS15 [7]	360	2.72 s	1.0×	386 ms	1.0×
	Ours	1920	1.27 s	2.1×	117.4 ms	3.3×

Without bootstrapping, our implementation reduces the amortized latency per AES block from 1.17 s to 452 ms on the local machine and from 528 ms to 45.2 ms on the 64-thread server, corresponding to speedups of 2.6× and 11.7×, respectively. In this setting, the end-to-end computation consists of the ten-round homomorphic AES evaluation, so the reduction in AES evaluation cost directly translates into the overall performance improvement. With bootstrapping, the corresponding speedups are 2.1× and 3.3×. The relative improvement is smaller because bootstrapping introduces a substantial additional cost that is not eliminated by the hybrid representation, thereby reducing the fraction of the total execution time affected by our AES-specific optimization.

The performance improvement primarily results from reducing automorphism-induced key switching in the post-inversion AES layers. By evaluating these operations in the bit-oriented representation, our scheme lowers both key-switching overhead and noise consumption, allowing the complete ten-round non-bootstrapping evaluation to

use $m = 53261$, compared with $m = 65281$ for the re-parameterized GHS15 baseline. The representation-conversion overhead is further reduced by the hoisted unpacking optimization evaluated in Section 4.3. In addition, our larger batch size provides substantially more parallel work for multi-core execution, which explains why the speedup increases from 2.6× on the local machine to 11.7× on the 64-thread server.

4.5 Comparison with Prior Transciphering Approaches

Table 6 summarizes BGV-, TFHE-, and CKKS-based AES transciphering implementations discussed in Section 1.1. They differ substantially in arithmetic model, hardware platform, batching strategy, and bootstrapping mechanism. In particular, our goal is to improve the efficiency of exact BGV-based AES transciphering by addressing the representation and key-switching overhead of the classical byte-oriented design.

Table 6 Reported results of representative homomorphic AES transciphering works, collected from the respective papers. The entries differ in arithmetic model, hardware platform, security parameters, batch size, parallelism, and whether bootstrapping is included. In the Parallelism column, numerical entries denote the number of CPU threads, whereas GPU entries report the GPU model. "Time/block" denotes single-block latency for unbatched implementations and amortized time per block for batched ones. The table is intended to place our work in the design space rather than to provide a normalized performance ranking.

Scheme	Work	Parallelism	Batch	Time / block
BGV	GHS15 (No-BS) [7]	1	84	1.17 s
		64	84	528 ms
	Ours (No-BS)	1	1920	452 ms
		64	1920	45.2 ms
TFHE	SMK22 [8]	1	1	252 s
	TCBS23 [9]	1	1	270 s
		16	1	54 s
	WWL$^+$23 [10]	1	1	86 s
	WLW$^+$24 [11]	1	1	46 s
	WWL$^+$24 [12]	1	1	26.2 s
	WHZ$^+$26 [15]	H100	1	0.17 s
CKKS	ADE$^+$23 [16]	64	2^{15}	37 ms
	NHY$^+$25 (Param-AES-14) [18]	64	2^{14}	14.4 ms
	ST26 [19]	64	1024	53.1 ms
	KLN$^+$26 [20]	RTX 5090	1	26 ms
		RTX 5090	2048	0.067 ms

These approaches represent different design points. The main limitation of the classical packed BGV design is not only the nonlinear inversion itself, but the automorphism and key-switching overhead introduced by evaluating the subsequent linear layers. Under identical software, hardware, and security settings, our implementation reduces the amortized per-block time of the reproduced GHS15 baseline while preserving exact arithmetic throughout the AES evaluation.

TFHE-based approaches provide a natural mechanism for evaluating nonlinear functions through programmable bootstrapping and LUT evaluation. Their single-block latency has been substantially reduced in recent work, particularly with GPU acceleration. Nevertheless, the limited SIMD parallelism of TFHE leads to a different performance trade-off from ring-based batching approaches.

Recent CKKS-based implementations report lower amortized per-block times, particularly on GPUs. For example, [20] reports a latency-oriented implementation evaluating one AES block in 26 ms on an NVIDIA RTX 5090, while its throughput-oriented variant processes up to 2048 blocks in parallel. This performance advantage, however, comes from a different arithmetic model: CKKS natively operates on approximate values, and AES evaluation therefore requires additional mechanisms to recover or preserve Boolean semantics during the computation. In contrast, our work addresses a different question: how far the performance of BGV-based AES transciphering can be improved while retaining exact finite-field arithmetic. The two approaches therefore optimize different points in the design space rather than representing directly interchangeable implementations.

5 Conclusion

In this work, we revisited BGV-based homomorphic AES evaluation from the perspective of plaintext representation. We introduced a hybrid byte–bit strategy that retains the byte-oriented representation for the multiplicative inversion in SubBytes, while evaluating the subsequent affine and linear operations in a bit-oriented representation over $\mathbb{F}_2$. This design reduces the automorphism and key-switching overhead associated with the classical byte-oriented evaluation. We further applied hoisting to the byte-to-bit unpacking procedure to reduce the cost of repeated representation conversion.

Our implementation in HElib shows that hoisted unpacking reduces the accumulated unpacking time by approximately 2.5×. Compared with the reproduced GHS15 baseline under updated parameters satisfying at least 128-bit security, the complete AES evaluation achieves speedups of 2.6× and 11.7× without bootstrapping, and 2.1× and 3.3× with bootstrapping, on the local machine and the 64-thread server, respectively. These results demonstrate that part of the overhead in classical BGV-based AES evaluation can be reduced through a representation better matched to the algebraic structure of different AES components, while retaining exact finite-field arithmetic.

Funding

This work was supported by the National Natural Science Foundation of China (Nos. 62472431, 12631019, U25A20429) and the Hunan Provincial Science Fund for Distinguished Young Scholars (No. 2025JJ20066).

Conflicts of interest

The authors declare that they have no conflicts of interest.

References

[1] Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the 41st Annual ACM Symposium on Theory of Computing. STOC '09, pp. 169–178. ACM, New York (2009)

[2] Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping. In: Innovations in Theoretical Computer Science (ITCS'12) (2012). http://eprint.iacr.org/2011/277

[3] Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In: Advances in Cryptology – ASIACRYPT 2017. Lecture Notes in Computer Science, vol. 10624, pp. 377–408. Springer, Cham (2017)

[4] Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: Fast fully homomorphic encryption over the torus. Journal of Cryptology **33**(1), 34–91 (2020)

[5] Cheon, J.H., Kim, A., Kim, M., Song, Y.S.: Homomorphic encryption for arithmetic of approximate numbers. In: Advances in Cryptology – ASIACRYPT 2017. Lecture Notes in Computer Science, vol. 10624, pp. 409–437. Springer, Cham (2017)

[6] Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: Advances in Cryptology – CRYPTO 2012. Lecture Notes in Computer Science, vol. 7417, pp. 850–867. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_49

[7] Gentry, C., Halevi, S., Smart, N.P.: Homomorphic Evaluation of the AES Circuit (Updated Implementation). Cryptology ePrint Archive, Paper 2012/099. Updated version, January 3, 2015 (2015). https://eprint.iacr.org/archive/2012/099/20150103:190644

[8] Stracovsky, R., Mahdavi, R.A., Kerschbaum, F.: Faster Evaluation of AES using TFHE. Poster Session, FHE.org 2022 (2022). https://rasoulam.github.io/data/poster-aes-tfhe.pdf

[9] Trama, D., Clet, P.-E., Boudguiga, A., Sirdey, R.: At Last! A Homomorphic AES Evaluation in Less than 30 Seconds by Means of TFHE. Cryptology ePrint Archive, Paper 2023/1020 (2023). https://eprint.iacr.org/2023/1020

[10] Wei, B., Wang, R., Li, Z., Liu, Q., Lu, X.: Fregata: Faster homomorphic evaluation of AES via TFHE. In: Information Security – 26th International Conference, ISC 2023. Lecture Notes in Computer Science, vol. 14411, pp. 392–412. Springer, Cham (2023)

[11] Wei, B., Lu, X., Wang, R., Liu, K., Li, Z., Wang, K.: Thunderbird: Efficient homomorphic evaluation of symmetric ciphers in 3GPP by combining two modes of TFHE. IACR Transactions on Cryptographic Hardware and Embedded Systems **2024**(3), 530–573 (2024)

[12] Wang, R., Wen, Y., Li, Z., Lu, X., Wei, B., Liu, K., Wang, K.: Circuit bootstrapping: Faster and smaller. In: Advances in Cryptology – EUROCRYPT 2024, pp. 342–372. Springer, Cham (2024)

[13] Schild, L., Abidin, A., Preneel, B.: Fast transciphering via batched and reconfigurable LUT evaluation. IACR Transactions on Cryptographic Hardware and Embedded Systems **2024**(4), 205–230 (2024)

[14] Belaïd, S., Bon, N., Boudguiga, A., Sirdey, R., Trama, D., Ye, N.: Further Improvements in AES Execution over TFHE: Towards Breaking the 1 sec Barrier. Cryptology ePrint Archive (2025)

[15] Wang, Z., He, H., Zhao, L., Niu, Q., Meng, D., Hou, R.: Maverick: Rethinking TFHE bootstrapping on GPUs via algorithm-hardware co-design. In: Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. ASPLOS '26, pp. 1168–1184. ACM, New York (2026). https://doi.org/10.1145/3779212.3790186

[16] Aharoni, E., Drucker, N., Ezov, G., Kushnir, E., Shaul, H., Soceanu, O.: E2E near-standard and practical authenticated transciphering. Cryptology ePrint Archive, Paper 2023/1040 (2023). https://eprint.iacr.org/2023/1040

[17] Drucker, N., Moshkowich, G., Pelleg, T., Shaul, H.: BLEACH: Cleaning errors in discrete computations over CKKS. Journal of Cryptology **37**(3) (2024) https://doi.org/10.1007/s00145-023-09483-1

[18] 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) https://doi.org/10.46586/tches.v2025.i4.118-144

[19] Shokri, R., Tsoutsos, N.G.: Walsh LUT Evaluation on Lazy Bits for CKKS. Cryptology ePrint Archive, Paper 2026/1385 (2026). https://eprint.iacr.org/2026/

1385

[20] 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. https://eprint.iacr.org/2026/1209 (2026)

[21] Dobraunig, C., Grassi, L., Helminger, L., Rechberger, C., Schofnegger, M., Walch, R.: Pasta: A case for hybrid homomorphic encryption. IACR Transactions on Cryptographic Hardware and Embedded Systems **2023**(3), 30–73 (2023) https://doi.org/10.46586/tches.v2023.i3.30-73

[22] Ha, J., Kim, S., Lee, B., Lee, J., Son, M.: Rubato: Noisy ciphers for approximate homomorphic encryption. In: Advances in Cryptology – EUROCRYPT 2022, pp. 581–610. Springer, Cham (2022)

[23] Cho, J., Ha, J., Kim, S., Lee, B., Lee, J., Lee, J., Moon, D., Yoon, H.: Transciphering framework for approximate homomorphic encryption. In: Advances in Cryptology – ASIACRYPT 2021, pp. 640–669. Springer, Cham (2021)

[24] Grassi, L., Ayala, I.M., Hovd, M.N., Øygarden, M., Raddum, H., Wang, Q.: Cryptanalysis of symmetric primitives over rings and a key recovery attack on rubato. In: Advances in Cryptology – CRYPTO 2023 (2023)

[25] Jiao, L., Li, Y., Hao, Y., Gong, X.: Differential fault attacks on privacy protocols friendly symmetric-key primitives: Rain and hera. IET Information Security (2024)

[26] Liu, F., Kalam, A., Sarkar, S., Meier, W.: Algebraic attack on fhe-friendly cipher hera using multiple collisions. IACR Transactions on Symmetric Cryptology (2024)

[27] Liu, F., Wang, T., Ni, J., Yamamoto, D., Sato, G.T.S., Wang, H., Meier, W.: Chosen-iv algebraic attack on randomized ciphers fasta and hera. IACR Transactions on Symmetric Cryptology (2025)

[28] Halevi, S., Shoup, V.: Bootstrapping for HElib. Journal of Cryptology **34**(1), 1–44 (2021)

[29] Geelen, R., Iliashenko, I., Kang, J., Vercauteren, F.: On polynomial functions modulo p^e and faster bootstrapping for homomorphic encryption. In: Advances in Cryptology – EUROCRYPT 2023. Lecture Notes in Computer Science, vol. 14006, pp. 257–286. Springer, Cham (2023)

[30] Alperin-Sheriff, J., Peikert, C.: Practical bootstrapping in quasilinear time. In: Advances in Cryptology – CRYPTO 2013. Lecture Notes in Computer Science, vol. 8042, pp. 1–20. Springer, Berlin, Heidelberg (2013)

[31] Geelen, R., Vercauteren, F.: Bootstrapping for BGV and BFV revisited. Journal of Cryptology **36**(2), 12 (2023) https://doi.org/10.1007/s00145-023-09454-6

[32] Gentry, C., Halevi, S., Smart, N.P.: Better bootstrapping in fully homomorphic encryption. In: Public Key Cryptography – PKC 2012. Lecture Notes in Computer Science, vol. 7293, pp. 1–16. Springer, Berlin, Heidelberg (2012)

[33] Liu, Z., Wang, Y.: Relaxed Functional Bootstrapping: A New Perspective on BGV/BFV Bootstrapping. Cryptology ePrint Archive, Paper 2024/172. https://eprint.iacr.org/2024/172 (2024)

[34] Ma, S., Huang, T., Wang, A., Wang, X.: Accelerating BGV bootstrapping for large p using null polynomials over $\mathbb{Z}_{p^e}$. In: Advances in Cryptology – EUROCRYPT 2024. Lecture Notes in Computer Science, vol. 14652, pp. 403–432. Springer, Cham (2024)

[35] Ma, S., Huang, T., Wang, A., Wang, X.: Faster BGV bootstrapping for power-of-two cyclotomics through homomorphic NTT. In: Advances in Cryptology – ASIACRYPT 2024. Lecture Notes in Computer Science, vol. 15484, pp. 143–175. Springer, Cham (2024). https://doi.org/10.1007/978-981-96-0875-1_5

[36] Geelen, R., Vercauteren, F.: Fully homomorphic encryption for cyclotomic prime moduli. In: Advances in Cryptology – EUROCRYPT 2025. Lecture Notes in Computer Science, vol. 15603, pp. 366–397. Springer, Cham (2025). https://doi.org/10.1007/978-3-031-91131-6_13

[37] Boyar, J., Peralta, R.: A depth-16 circuit for the AES s-box. Technical Report 2011/332, Cryptology ePrint Archive (2011). https://eprint.iacr.org/2011/332

[38] Halevi, S., Shoup, V.: Bootstrapping for helib. In: Advances in Cryptology – EUROCRYPT 2015, pp. 641–670. Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46800-5_25

[39] Halevi, S., Shoup, V.: Faster homomorphic linear transformations in helib. In: Advances in Cryptology – CRYPTO 2018, pp. 93–120. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96884-1_4