Latency-Aware, High-Throughput Homomorphic AES Evaluation with CKKS

Taeseong Kim
Department of Mathematics
Seoul National University
Seoul, Republic of Korea
kts1023@snu.ac.kr

Jonghoo Lee
CryptoLab, Inc.
Seoul, Republic of Korea
jonghoolee@cryptolab.co.kr

Taeyeong Noh
CryptoLab, Inc.
Seoul, Republic of Korea
tynoh0219@cryptolab.co.kr

Jung Hee Cheon
Department of Mathematics
Seoul National University
Seoul, Republic of Korea
CryptoLab, Inc.
Seoul, Republic of Korea
jungheecheon@cryptolab.co.kr

Guillaume Hanrot
CryptoLab, Inc.
Lyon, France
guillaume.hanrot@cryptolab.co.kr

Abstract

Homomorphic Advanced Encryption Standard (AES) evaluation refers to evaluating the AES circuit with a fully homomorphic encryption (FHE)-encrypted secret key. Applications include in particular Transciphering, which converts AES-encrypted data into FHE ciphertexts without exposing the secret key.

Existing homomorphic AES evaluations show a clear separation between latency-oriented solutions and throughput-oriented solutions. CKKS-based methods exploit massive SIMD parallelism and focus on throughput by processing many AES blocks in parallel. They are hardly suitable for latency-critical settings. In contrast, TFHE-based methods process a small number of blocks efficiently. They are preferable for low-latency settings, but provide very limited throughput.

In this work, we show that AES-CKKS evaluation can achieve both interactive latency and high throughput. Our first variant is optimized for latency and decrypts a single AES block in only 26ms on an NVIDIA RTX-5090. This is more than 6× faster than recent TFHE-based state-of-the-art approaches Our second variant is based on a new embedding of GF(16), the finite field with 16 elements, into CKKS message space. It is optimized for throughput and processes up to 2048 AES blocks at once, achieving over 238KB/s throughput (a 3.41× improvement over the state-of-the-art CKKS-based approaches), while maintaining latency comparable to TFHE-based methods. To the best of our knowledge, this is the first AES-FHE evaluation algorithm combining good latency and throughput properties, bringing homomorphic outsourcing with AES within reach of real-time applications on constrained devices.

Our main ingredients are redundant structures that maximize SIMD utilization, improved algorithms for the SubBytes step (one of them being based on inversion in GF(256) using CKKS), fusion of linear layers into bootstrapping, and carefully crafted FHE parameters.

Keywords

Fully homomorphic encryption, Transciphering, CKKS, AES

1 Introduction

The homomorphic evaluation of symmetric ciphers—evaluating encryption or decryption directly over FHE ciphertexts without exposing plaintext or key—is a fundamental primitive at the intersection of symmetric and homomorphic cryptography. Its most natural instantiation is the transciphering paradigm, which addresses the client-side bottleneck in FHE-based computation. Rather than requiring the client to perform expensive FHE encryption, transciphering delegates this task to the server: the client encrypts data under a lightweight symmetric cipher and sends an FHE encryption of the key, while the server evaluates the decryption circuit to convert the ciphertext into an FHE ciphertext. This separation of tasks enables practical deployment on resource-constrained devices such as sensors and IoT nodes, but relies critically on the efficient homomorphic evaluation of the underlying symmetric cipher.

In practice, the choice of symmetric cipher is rarely flexible. A line of recent work has proposed FHE-friendly ciphers—such as Pasta [21], Rubato [27], and Hera [18]—designed to minimize multiplicative depth and reduce homomorphic evaluation cost. While these constructions enable fast transciphering, they come with a critical limitation: unlike standardized ciphers, they lack extensive cryptanalytic scrutiny, and several early proposals have already been broken [26, 31, 35, 36]. In contrast, AES is not merely a design choice but a requirement in many real-world deployments, where compatibility with existing infrastructure is essential. Moreover, vast amounts of sensitive data are already stored in AES-encrypted form, making homomorphic AES evaluation the only viable path to enable privacy-preserving computation over such legacy data. Consequently, homomorphic AES transciphering is a prerequisite for the practical adoption of FHE in these settings.

While maximizing homomorphic AES evaluation throughput is natural in many settings, practical applications impose more complex performance requirements. In FHE-based threshold AES decryption, where an AES key is secret-shared among mutually distrusting parties and AES is evaluated homomorphically to jointly decrypt data, throughput can be important when multiple ciphertexts must be processed. At the same time, such protocols are often embedded in interactive MPC workflows, where latency accumulates

across rounds and directly impacts overall performance. Similarly, applications such as AES-based homomorphic key derivation function (KDF) are inherently latency-sensitive, since key derivation lies on the critical path of subsequent cryptographic operations.

A dominating assumption in the community is that the latency-throughput tradeoff in FHE evaluation is largely governed by the scheme choice: TFHE targets low-latency, small integers operations with efficient boolean circuits, while CKKS, BFV and BGV achieve high-throughput through SIMD batching at the cost of latency. Homomorphic AES evaluation is no exception to this rule. Existing CKKS-based designs [6, 37] are strongly throughput-oriented, processing $\geq 2^{14}$ blocks at once, to process blocks fully in parallel and avoid expensive slot manipulation. The counterpart of this design choice is poor latency performance, of the order of several seconds using GPU, an order of magnitude which is not acceptable for the applications previously mentioned. Meanwhile, state-of-the-art TFHE designs [41] achieve good latency (of the order of 0.2-0.4s on GPU) but process only a few blocks at once, thereby obtaining poor throughput performance.

We challenge this assumption and demonstrate that CKKS can simultaneously improve both state-of-the-art latency and throughput, disproving the perceived tradeoff in this setting and providing a strong practical solution to the homomorphic AES evaluation problem.

1.1 Contributions

We provide two new homomorphic designs for the AES evaluation with the CKKS scheme, designed to address the latency bottleneck of existing solutions. Our second circuit simultaneously improves on state-of-the-art throughput for homomorphic AES evaluation.

Our $\textbf{Latency}_1$ circuit is optimized purely for latency and achieves an evaluation time of 26ms on a commodity GPU to process a single AES block. We use two techniques: (1) optimized use of the redundancy provided by the structure of sub-ring CKKS ciphertexts for SubBytes and linear algebra steps; (2) fusion of the full AES circuit – except the SubBytes step – directly into the CKKS bootstrapping operation. A variant of this solution, called $\textbf{Latency}_4$, processes 4 blocks at once in 29ms.

Our **GF(16)** circuit targets a practical compromise, yielding higher throughput without sacrificing the latency objective. It relies on a novel finite field encoding and achieves various latency/throughput tradeoffs, from 40 ms latency for 103.5KB/s throughput to 137ms latency for 238.7 KB/s throughput.

1.2 Related Work

The homomorphic AES evaluation framework was first introduced in the BGV scheme by Gentry et al. [24, 25], where AES-BGV transciphering was proposed. They reported a latency of 252s (2s per block) for non-bootstrappable parameters and 1050s (5.8s per block) for bootstrappable parameters.

Following this pioneering work, numerous studies have been conducted to improve the performance of AES-FHE evaluation [1, 6, 9, 37, 39, 40]. Broadly, existing approaches can be categorized into two directions: throughput-oriented scenarios leverage SIMD-aware schemes such as CKKS over discrete data [1, 6, 37], whereas latency-oriented scenarios typically adopt CGGI-based implementations [9, 39–41].

Throughput: AES-CKKS Evaluation. Following the introduction of techniques for handling discrete data over CKKS in [7, 8, 22], several works have explored improving the throughput of homomorphic AES evaluation by leveraging the SIMD capability of CKKS and its fast bootstrapping [1, 6, 37].

Early work by IBM [1] demonstrated high throughput AES transciphering, and subsequent works such as XBOOT [37] and [6] further improved efficiency through optimized circuit design with the improved CKKS algorithms [8, 33] and the use of binary bootstrapping [7]. In particular, [6] achieves > 1000 blocks per second throughput on modern GPUs.

Despite these improvements in throughput, CKKS-based approaches still incur relatively high latency due to the design choice of processing a large number of ciphertexts during evaluation in order to avoid expensive slot management operations.

Latency: AES-TFHE Evaluation. TFHE schemes support fast functional bootstrapping, which enables the evaluation of the AES circuit by replacing Boolean gates and non-linear S-box operations with programmable bootstrapping procedures. Leveraging this advantage, several TFHE-based AES evaluations [9, 39, 40] achieve low latency for a small number of AES blocks, either by optimizing functional bootstrapping or by adopting improved leveled homomorphic encryption techniques [17]. More recently, GPU acceleration has reduced the latency of AES transciphering from several seconds to as low as 170 ms [41]. However, despite these improvements in latency, TFHE-based approaches inherently suffer from low throughput due to the lack of SIMD parallelism.

Existing works show a clear separation between throughput-oriented CKKS-based approaches and latency-oriented TFHE-based approaches. We demonstrate that CKKS-based AES evaluation can achieve both high throughput and low latency, bridging the gap between previous CKKS- and TFHE-based solutions. Notably, our 1-block, 26 ms latency-optimized solution on a commodity GPU significantly outperforms the 170ms latency of the 8-block state-of-the-art TFHE GPU implementation [41] while matching its throughput. Our 256-block / 512-block / 2048-block GF(16) circuits simultaneously satisfy the demands of throughput-heavy workloads while preserving a better latency than the TFHE-based solution.

The main idea used in the design of our GF(16) algorithm is an encoding from GF(16) into CKKS message space allowing to homomorphically compute with elements of GF(16) using CKKS. A similar idea was published and explored in [38] which appeared online during the final stages of preparing this manuscript. The two works offer complementary perspectives on this common idea: [38] focuses on generality and develops the idea towards implementation of finite fields in general, while our work targets usability and optimization in the context of GF(16) and GF(256) for AES transciphering. Specifically, we address the parameterization required in order to manage the encoding's numerical stability issues and optimize the evaluation of the AES circuit.

1.3 Technical overview

Our implementation of the AES circuit using CKKS starts from the XBOOT [37] design, which substitutes integer computations for modulo 2 arithmetic, performing modular reduction opportunistically during bootstrapping. This strategy requires bootstrapping between each round. This design induces many bootstrappings (10 in the case of AES-128), but produces very shallow circuits, allowing various homomorphic optimizations along the way. In return, reducing the depth of the AES circuit makes the bootstrapping performance better by reducing the total computational modulus. Efficient circuits thus need to achieve a search for tradeoffs between total depth (impacting mainly the bootstrapping costs) and number of operations (impacting the total circuit cost outside bootstrapping).

We adopt two different strategies to optimize latency, then to achieve a latency/throughput tradeoff.

1.3.1 Latency optimized solution. Our latency-optimized solution evaluates AES on a single block, represented in a CKKS ciphertext in a redundant way. We use this redundancy and the SIMD capability of CKKS in order to optimize the various steps of the AES evaluation.

Homomorphic SubBytes. The SubBytes step is implemented using polynomial interpolation. We evaluate the interpolation polynomial in the *Lagrange basis*, as a linear combination of indicator functions $\mathbf{1}_{\mathbf{y}}(\mathbf{x})$ for $\mathbf{y} \in \{0,1\}^8$. We take advantage of the massive redundancy to design a solution requiring only 7 rotations and 7 ciphertext-ciphertext multiplications to obtain the elements of the Lagrange basis evaluated at the input ciphertext. The remaining linear combination of the basis elements is treated as a linear algebra step.

Linear algebra steps. Previous approaches [6, 37] implement ShiftRows and MixColumns by linear combinations of ciphertexts, by using a bit-based packing where each of the 128 ciphertexts corresponds to one bit position over all the blocks. In contrast, our latency-oriented packing structure forces us to resort to a linear algebra formulation, and view these steps as more costly plaintext-ciphertext matrix-vector multiplications. The same is true for the final stage of the SubBytes computation in the "Latency" circuit.

In order to reduce the depth of the circuit, we fuse these various linear steps as a unique step, and fuse it together with the StC map which starts bootstrapping. This fusion produces a matrix which is as costly to evaluate as the StC map alone.

1.3.2 The GF(16) *circuit.* We adapt an idea originating in [23] in order to optimize for throughput. As the SubBytes step is derived from the computation of an inverse in GF(256), we realize finite field arithmetic within CKKS. We notice that the ring $\mathcal{R}' = \mathbb{Z}[X]/(X^4 + X^3 + X^2 + X + 1)$ embeds into $\mathbb{C}^2$ through its canonical embedding, so that $N/4$ elements of $\mathcal{R}'$ can be mapped to a CKKS message in a way compatible with the arithmetic of $\mathcal{R}'$. As $\mathcal{R}'/2\mathcal{R}' = \mathrm{GF}(16)$, by reducing modulo 2 we can similarly map $N/4$ copies of GF(16) in a CKKS message, in a way compatible with the arithmetic of GF(16).

Elements of GF(256) are then realized as pairs of elements of GF(16), and we use linearity of the map $x \mapsto x^{2^k}$ in these fields to obtain an algorithm for inverse computation with cost dominated by 4 ciphertext-ciphertext multiplications, using 3 multiplicative levels. One weakness of this finite-field encoding is that the values represented in the slots may quickly grow out of control, leading to numerical instability as soon as the number of multiplicative levels grows. We demonstrate that by carefully handling a number of choices (polynomial $x^4 + x^3 + x^2 + x + 1$, liftings of linear maps from GF(2) to $\mathbb{Z}$), this encoding can be kept numerically stable while achieving the required number of multiplicative levels.

Contrary to [23] where the complexity of the bootstrapping process significantly increases compared to CKKS bootstrapping, we extend the classical CKKS bootstrapping process to this encoding. In order to perform reduction modulo 2 of elements of $\mathcal{R}'$ during bootstrapping, we need to evaluate one more linear map at the beginning and at the end of the bootstrapping process, to move the coefficients of the elements of $\mathcal{R}'$ within the CKKS slots.

As in the Latency case, the remaining steps (including the end of SubBytes) are all fused as a single linear step, with AddRoundKey performed at the time of bootstrapping.

Implementation and applications. Implementing our results using the HEaaN2 library over an NVIDIA RTX-5090 GPU, we execute the Latency circuit in 26ms, obtaining a 6.95× improvement over the best SOTA latency [41]).

The GF(16) circuit runs in 51 ms (512-block version) or 137 ms (2048-block version) on RTX-5090. Comparing 2048-block version to SOTA for optimal throughput [6] on RTX-5090, we observe a 3.41× throughput improvement. Comparing the GF(16) circuit (2048-block version) to SOTA for optimal latency (TFHE-based), both on RTX-4090, we observe about 21% better latency together with an approximately 316× gain in throughput, making it a strong solution in all regards.

2 Preliminaries

2.1 Notations

Let $N > 1$ be a power-of-two integer, and define the ring $\mathcal{R}_N = \mathbb{Z}[X]/(X^N + 1)$ and its quotient $\mathcal{R}_{N,Q} = \mathcal{R}_N/Q\mathcal{R}_N$ for $Q \in \mathbb{Z}_{>0}$. When N is clear from the context, we omit the subscript and simply write $\mathcal{R}$ and $\mathcal{R}_Q$. For a positive integer ℓ, $[\ell] = \{0, 1, \ldots, \ell - 1\}$. For $a \in \mathbb{Z}$, let $[a]_q$ denote the signed modular reduction in $(-q/2, q/2]$. We define the rounding function $\lfloor x \rceil = \lfloor x + 1/2 \rfloor$. In order to avoid confusion with indices i, we use I for a complex square root of -1.

We use boldface letters (e.g., $\mathbf{a}$) to denote vectors. Given two vectors $\mathbf{a}, \mathbf{b}$, $\mathbf{a} + \mathbf{b}$ and $\mathbf{a} \odot \mathbf{b}$ denote component-wise addition and multiplication, respectively. For two polynomials $x, y \in \mathcal{R}$, we let $x + y$ and $x \cdot y$ denote polynomial addition and multiplication, respectively. For a vector $\mathbf{v}$, we let $\mathrm{Rot}_k(\mathbf{v})$ be the vector obtained by rotating cyclically the coordinates of $\mathbf{v}$ by k positions. For an integer n, we let $\mathbf{I}_n$ be the identity matrix of order n. Unless otherwise specified, log denotes $\log_2$.

We use the notation $\mathbf{a}^{\times k}$ to denote the k-fold concatenation of a vector $\mathbf{a}$, i.e. $\mathbf{a}^{\times k} = \underbrace{(\mathbf{a}, \mathbf{a}, \ldots, \mathbf{a})}_{k \text{ times}}$.

For matrices $\mathbf{A} = (a_{i,j}) \in \mathbb{C}^{m\times n}$ and $\mathbf{B} = (b_{k,\ell}) \in \mathbb{C}^{p\times q}$, we denote by $\mathbf{A} \otimes \mathbf{B}$ their Kronecker (tensor) product , defined as

$$\mathbf{A} \otimes \mathbf{B} = \begin{pmatrix} a_{1,1}\mathbf{B} & \cdots & a_{1,n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m,1}\mathbf{B} & \cdots & a_{m,n}\mathbf{B} \end{pmatrix} \in \mathbb{R}^{mp\times nq}.$$

With these notations, we have $\mathbf{a}^{\times k} = (1)^{\times k} \otimes \mathbf{a}$.

The mixed-product property $(\mathbf{A} \otimes \mathbf{B}) \cdot (\mathbf{C} \otimes \mathbf{D}) = (\mathbf{AC}) \otimes (\mathbf{BD})$ implies that $\mathbf{A} \otimes \mathbf{I}_n$ and $\mathbf{I}_m \otimes \mathbf{B}$ commute, where $\mathbf{A}$ and $\mathbf{B}$ are square matrices of size m and n, respectively, and $\mathbf{I}_m$ and $\mathbf{I}_n$ denote the identity matrices of size m and n. That is:

$$(\mathbf{A} \otimes \mathbf{I}_p) \cdot (\mathbf{I}_m \otimes \mathbf{B}) = \mathbf{A} \otimes \mathbf{B} = (\mathbf{I}_m \otimes \mathbf{B}) \cdot (\mathbf{A} \otimes \mathbf{I}_p).$$

2.2 Cheon-Kim-Kim-Song (CKKS) Scheme

The Cheon-Kim-Kim-Song (CKKS) [16] is a 4th generation FHE scheme, which can handle real and complex numbers as input and perform approximate arithmetic on these numbers. Given a power of two integer N, its message space is $\mathbb{C}^{N/2}$, and its plaintext space $\mathcal{R}_N$.

Encoding and decoding. Let $m(x) \in \mathcal{R}_N$ be a plaintext and let $\Delta \in \mathbb{R}_{>0}$ be a parameter called the scaling factor. We let $\zeta = \exp(I\pi/N) \in \mathbb{C}$ and define the discrete Fourier transform (DFT) as the 1-1 map $\mathsf{DFT} : \mathbb{R}[X]/(X^N + 1) \to \mathbb{C}^{N/2}$ given by

$$\mathsf{DFT}(p(x)) = \left(p(\zeta_i)\right)_{0\le i<N/2},$$

where $\zeta_i = \zeta^{5^i}$. We let $\mathsf{iDFT} : \mathbb{C}^{N/2} \to \mathbb{R}[X]/(X^N + 1)$ be the inverse mapping.

The slot-encoding map $\mathsf{Ecd} : \mathbb{C}^{N/2} \to \mathcal{R}$ is defined, for a message vector $\mathbf{m} \in \mathbb{C}^{N/2}$, as $\mathsf{Ecd}(\mathbf{m}, \Delta) = \lfloor \Delta \cdot \mathsf{iDFT}(\mathbf{m}) \rceil$, while the decoding map $\mathsf{Dcd} : \mathcal{R} \to \mathbb{C}^{N/2}$ is defined as $\mathsf{Dcd}(m, \Delta) = \Delta^{-1}\mathsf{DFT}(m)$. Those two maps are approximately inverse to one another, and establish an approximate ring isomorphism between $(\mathbb{C}^{N/2}, +, \cdot)$ and $(\mathcal{R}_N, +, \cdot)$, enabling SIMD-style parallel computation in the CKKS scheme. We refer to this representation as *slot encoding*. We also make use of the *conjugate-invariant* slot-encoding [33], which maps $\mathbb{R}^N$ into the conjugate-invariant ring $\mathbb{Z}[X+X^{-1}]/(X^N+X^{-N})$, and gives access to coordinate-wise arithmetic on $\mathbb{R}^N$.

We also consider an alternative representation, called *coefficient encoding*. Given $\mathbf{c} = (c_i)_{0\le i<N} \in \mathbb{R}^N$, the coefficient encoding map $\mathsf{CoeffEcd} : \mathbb{R}^N \to \mathcal{R}$ is defined as $\mathsf{CoeffEcd}(\mathbf{c}, \Delta) = \sum_{i=0}^{N-1} \lfloor \Delta \cdot c_i \rceil X^i$. When operating on a coefficient-encoded ciphertext, CKKS inherits the arithmetic of the plaintext space; addition still corresponds to componentwise addition.

Encryption and decryption. CKKS ciphertexts are elements of $\mathcal{R}_q^2$; the integer q is the modulus of the ciphertext. Given the underlying secret key $s \in \mathcal{R}$, the decryption of a ciphertext $\mathsf{ct} = (a, b) \in \mathcal{R}_q^2$ to a plaintext is $\mathsf{Dec}_s(a, b) = b + a \cdot s$.

Fewer Slot Encoding. We consider a *fewer-slot encoding* [15] that arises from the subring structure of the CKKS plaintext space. Let $n \mid N$ and define $Y = X^{N/n}$. A plaintext polynomial $m(Y) \in \mathbb{Z}[Y]/(Y^n + 1) = \mathcal{R}_n$ can be embedded into $\mathcal{R}_N$ via the map $Y \mapsto X^{N/n}$. Under the decoding map in $\mathcal{R}_N$, this embedding induces a structured repetition in the slot representation. Specifically, let

$$\mathbf{w} = (w_0, \ldots, w_{n/2-1}) \in \mathbb{C}^{n/2},$$

be a message in the subring $\mathcal{R}_n$. The corresponding embedded polynomial in $\mathcal{R}_N$ is then decoded as

$$\mathbf{z} = \mathbf{w}^{\times N/n} = (\mathbf{w}, \mathbf{w}, \ldots, \mathbf{w}) \in \mathbb{C}^{N/2}.$$

Homomorphic Operations. CKKS implements slot-wise addition (Add), plaintext–ciphertext multiplication (PMult) by direct extension of the corresponding operation in $\mathcal{R}_{N,q}^2$.

Other homomorphic primitives require the *key-switching operation*. For two secret keys s and s', the operation $\mathsf{KS}_{s'\to s}$ takes as input a switching key from s' to s and a ciphertext ct' and returns a ciphertext ct such that $\mathsf{Dec}_s(\mathsf{ct}) \approx \mathsf{Dec}_{s'}(\mathsf{ct}')$.

In particular, Rot and Conj are realized by a Galois automorphism σ applied to the coordinates of the 2-dimensional ciphertext (a, b), followed by key-switching ($\mathsf{KS}_{\sigma(s)\to s}$).

Ciphertext-ciphertext multiplication (Mult) starts with the tensor operation $\mathsf{Tensor}((a, b), (a', b')) = (a \cdot a', a \cdot b' + a' \cdot b, b \cdot b')$, and proceeds with the Relin operation defined by $\mathsf{Relin}(u, v, w) = (u, v) + \mathsf{KS}_{s^2\to s}(w, 0)$. This produces a ciphertext encrypting the plaintext $\mathsf{Dec}_s((a, b)) \cdot \mathsf{Dec}_s((a', b'))$.

However, multiplying two inputs (plaintext or ciphertext) encoded using a scaling factor Δ yields a result encoded with scaling factor Δ^2. Let $\mathsf{ct} = (a, b) \in \mathcal{R}_{N,Q}$ be a ciphertext with scaling factor Δ^2, and assume that $q \approx \Delta$ is a divisor of Q. Then, the ciphertext $\mathsf{Rescale}(a, b) = (\lfloor a/q \rceil, \lfloor b/q \rceil) \in \mathcal{R}_{N,Q/q}^2$ has scaling factor approximately equal to Δ. This Rescale step concludes the ciphertext-ciphertext multiplication procedure.

Both Relin and Rescale commute with addition; this allows to evaluate them lazily in computations of the type $\sum_{i=0}^{k} \mathsf{Mult}(\mathsf{ct}_i, \mathsf{ct}'_i)$.

We note that the modulus of ciphertexts reduces during multiplicative steps, leading to the notion of *level*. A ciphertext at level ℓ lies in $\mathcal{R}_{Q_\ell}^2$, where $Q_\ell = \prod_{i=0}^{\ell} q_i$ for pairwise distinct primes q_i is the remaining ciphertext modulus and each modulus prime satisfies $q_\ell \approx \Delta$ for the ciphertext scaling factor Δ. Multiplication, then rescaling thus reduces the modulus from Q_ℓ to $Q_{\ell-1} \approx Q_\ell/\Delta$, so each multiplication consumes one level of the modulus chain.

All homomorphic primitives reduce to arithmetic in the ring $\mathcal{R}_{N,Q}$, which have quasi-linear cost in N and $\log Q$. Reducing Q, hence the consumed depth, is a central goal to optimize practical CKKS algorithms.

Bootstrapping. When a ciphertext can no longer support further multiplications, a bootstrapping procedure (BTS) is required. The StC-first CKKS BTS [4] consists of the following four steps.

SlotToCoeff (StC) starts with ct such that $\mathsf{Dec}(sk, \mathsf{ct}) = \mathsf{Ecd}(\mathbf{m}, \Delta)$ for $\mathbf{m} \in \mathbb{C}^{N/2}$ and outputs a ciphertext $\mathsf{ct}_{\mathsf{StC}}$ satisfying

$$\mathsf{Dec}(sk, \mathsf{ct}_{\mathsf{StC}}) = \mathsf{CoeffEcd}\left(\mathrm{Re}(\pi(\mathbf{m}))||\mathrm{Im}(\pi(\mathbf{m})), \Delta\right),$$

where π is the bit-reversal permutation. Up to slight adaptations, StC corresponds to the evaluation of the DFT map – it is a homomorphic linear algebra step.

ModRaise (MR) takes as input a ciphertext $\mathsf{ct}_{\mathsf{StC}} = (a, b) \in \mathcal{R}_{N,q_0}^2$ such that $\mathsf{Dec}(sk, \mathsf{ct}_{\mathsf{StC}}) = \mathsf{CoeffEcd}(\mathbf{m}, \Delta')$, and outputs a ciphertext $\mathsf{ct}_{\mathsf{MR}} \in \mathcal{R}_Q^2$ satisfying

$$\mathsf{Dec}(sk, \mathsf{ct}_{\mathsf{MR}}) = \mathsf{CoeffEcd}(\mathbf{m} + q_0\mathcal{I}, \Delta),$$

where $\mathcal{I} \in \mathbb{Z}^N$ is a vector with small integer entries.

CoeffToSlot (CtS) takes as input a ciphertext $\mathsf{ct}_{\mathsf{MR}}$ such that

$$\mathsf{Dec}(sk, \mathsf{ct}_{\mathsf{MR}}) = \mathsf{CoeffEcd}((\mathbf{m}_1 + q_0\mathcal{I}_1)||(\mathbf{m}_2 + q_0\mathcal{I}_2), \Delta),$$

and outputs a ciphertext $\mathsf{ct}_{\mathsf{CtS}}$ satisfying[1]

$$\mathsf{Dec}(sk, \mathsf{ct}_{\mathsf{CtS}}) = \mathsf{Ecd}(\pi^{-1}(\mathbf{m}_1) + I \cdot \pi^{-1}(\mathbf{m}_2) + q_0(\pi^{-1}(\mathcal{I}_1) + I \cdot \pi^{-1}(\mathcal{I}_2)), \Delta).$$

Finally, EvalMod (EM) takes as input a ciphertext $\mathsf{ct}_{\mathsf{CtS}}$ such that

$$\mathsf{Dec}(sk, \mathsf{ct}_{\mathsf{CtS}}) = \mathsf{Ecd}(\mathbf{m} + q_0\mathcal{I}, \Delta),$$

and restores the original ciphertext format by returning $\mathsf{ct}' \in \mathcal{R}^2_{N,q'}$, where $q_0 < q' < Q$, satisfying $\mathsf{Dec}(sk, \mathsf{EM}(\mathsf{ct})) = \mathsf{Ecd}(\mathbf{m}, \Delta)$.

We make use of binary bootstrapping [7], a circuit specifically designed to handle messages in $\{0,1\}^{N/2}$. By setting $q_0 \approx 2\Delta$, this circuit provides free reduction modulo 2 of the underlying message before ModRaise.

When a ciphertext encrypts a message using only $n|N$ slots, with an underlying plaintext $\mathcal{R}_n$, both CtS and StC correspond to (i)$\mathbf{DFT}_n$ rather than (i)$\mathbf{DFT}_N$, reducing the overall cost of BTS. However, the $\mathcal{I}$ part introduced by the MR step creates an underlying plaintext which is no longer in $\mathcal{R}_n$. The structure is restored via the homomorphic evaluation of the trace map Tr_{R_N/R_n}, which requires $\log(N/n)$ automorphisms and additions.

Decomposition of StC. We denote by $\mathbf{DFT}$ the matrix associated to the DFT map. It admits a Cooley-Tukey style factorization into a product of sparse matrices, and this decomposition has been exploited to accelerate StC and CtS in CKKS Bootstrapping [12, 30]. Specifically, $\mathbf{DFT}_{N/2}$ can be decomposed into $\log N/2$ sparse matrices.

$$\mathbf{DFT}_{N/2} = D_{N/2} \cdot D_{N/4} \cdots D_2, \tag{1}$$

where $D_k = I_{N/2k} \otimes \begin{pmatrix} I_{k/2} & W_{k/2} \\ I_{k/2} & -W_{k/2} \end{pmatrix}$ and $W_{k/2}$ are diagonal matrices with roots of unity. [2]

Homomorphic matrix vector multiplication. Given a matrix $A = (a_{i,j}) \in \mathbb{C}^{d_1 \times d_2}$ with $d_1|d_2$, for $0 \le \ell < d_1$, we define its ℓ-th diagonal vector as

$$\Delta_\ell(\boldsymbol{A}) := \left(a_{i \bmod d_1,\, (i+\ell) \bmod d_2}\right)_{0 \le i < d_2} \in \mathbb{C}^{d_2},$$

where the row index is taken modulo d_1 and the column index is taken modulo d_2. We make use of the diagonal method [28] and the rectangular variant described in Gazelle [32] to compute homomorphic matrix-vector products. Given an encrypted vector $\mathbf{v} \in \mathbb{C}^{d_2}$, we compute

$$\mathbf{w} = \sum_{\ell=0}^{d_1-1} \Delta_\ell(\boldsymbol{A}) \odot \mathsf{Rot}_\ell(\mathbf{v}) \in \mathbb{C}^{d_2} \tag{2}$$

and derive a vector containing d_2/d_1 copies of the desired $A \cdot \mathbf{v} \in \mathbb{C}^{d_1}$ by folding $\mathbf{w}$ over blocks of length d_1 by computing the vector $\sum_{j=0}^{\log(d_2/d_1)-1} \mathsf{Rot}_{d_1 j}(\mathbf{w}) \in \mathbb{C}^{d_2}$ which contains d_2/d_1 copies of $\boldsymbol{A} \cdot \mathbf{v}$.

Equation (2) can be evaluated by using the baby-step giant-step (BSGS) technique, which we outline in the sparse setting. Let S be the set of non-zero diagonals and assume that S_1, S_2 are two sets such that $S \subset \{s_1 + s_2 \bmod d_1, s_1 \in S_1, s_2 \in S_2\}$. By decomposing the summation index ℓ as $\ell_1 + \ell_2$, $\ell_1 \in S_1$, $\ell_2 \in S_2$, we obtain

$$\mathbf{w} = \sum_{\ell_1 \in S_1} \mathsf{Rot}_{\ell_1}\left(\sum_{\ell_2 \in S_2} \mathsf{Rot}_{-\ell_1}(\Delta_\ell) \odot \mathsf{Rot}_{\ell_2}(\mathbf{v})\right). \tag{3}$$

This formula uses $\approx |S_1| + |S_2|$ rotations of the vector. Here, the inner rotations $\mathsf{Rot}_{\ell_2}(\tilde{\mathbf{v}})$ correspond to the baby-step (BS) part. Since all rotations in the BS step are applied to the same input ciphertext, hoisting [29] can be used to share the common key-switching decomposition, further reducing the practical cost.

2.3 AES-FHE Evaluation

2.3.1 Advanced Encryption Standard (AES). The Advanced Encryption Standard (AES), in its 128-bit version, is a widely used, NIST-standardized symmetric block cipher that operates on 128-bit data blocks through 10 iterative rounds of substitution and permutation operations, combining non-linear and linear transformations.

We briefly review the main components of an AES round. Offline, a set of 10 round keys is derived from the 128-bit secret key; this key expansion step is described in Appendix F.

SubByte (SB) This nonlinear operation applies the AES S-box, which is constructed from inversion in GF(256) and a subsequent affine transformation. On bit-decomposed inputs, this can be expressed as the multivariate polynomial

$$\mathsf{SB}(\mathbf{x}) = \sum_{\mathbf{y} \in \{0,1\}^8} A(\mathbf{y}) \prod_{j=0}^{7} \left(x_j^{y_j}(1-x_j)^{1-y_j}\right), \tag{4}$$

where $A(\mathbf{y})$ denotes the corresponding entry of the S-box table.

ShiftRows (SR) This step cyclically shifts each row of the byte state according to a predefined pattern and can be seen as a permutation of a 128 bit vector representation of a block.

MixColumns (MC) This linear transformation applies a fixed matrix multiplication over GF(256) to each column of the AES state. When the state is represented as a 128-bit vector, as in our AES-CTR implementation, the operation becomes a $\mathbb{Z}_2$-linear map.

AddRoundKey (ARK) Input is XORed bit-wise with the corresponding round key derived from the secret key.

Both the SubByte and the MixColumns step make use of the field structure of the field GF(256). By using the explicit realization of GF(256) used in AES,

$$\mathrm{GF}(256)_{\mathsf{AES}} = \mathrm{GF}(2)[X]/(X^8 + X^4 + X^3 + X + 1).$$

and mapping a polynomial $\sum_{i=0}^{7} a_i X^i$ to the binary vector $(a_i)_{0 \le i < 8} \in \mathrm{GF}(2)^8$, these steps can be seen as purely linear or affine operations. The hexadecimal notation $0xUV$, where $U = \sum_{i=4}^{7} a_i 2^{i-4}$, $V = \sum_{i=0}^{3} a_i 2^i$ are hexadecimal digits, is also commonly used to represent elements of GF(256).

Our Latency algorithms make use of the purely linear view, while our GF(16)-based algorithm switches from the algebraic view for the SB operations and the linear view for other steps.

2.3.2 Homomorphic AES-FHE Evaluation: short state of the art. Among the AES components, SB is the only non-linear operation and is usually computed through (4) with various strategies [1, 6, 37].

[1] Note that $\pi^{-1} = \pi$.

[2] The actual decomposition includes the bit-reversal matrix, i.e. $\mathbf{DFT}_{N/2} = D_{N/2} \cdot D_{N/4} \cdots D_2 \cdot P$, where P is a bit-reversal matrix. However, for efficiency, StC and CtS omit P and P^{-1}. For simplicity, we henceforth omit the bit-reversal matrix.

Since AES contains many bit permutations, CKKS-based homomorphic AES commonly uses bit-sliced packing, where each N-slot ciphertext stores one bit position across N AES instances. In this representation, SR is a mere permutation of ciphertexts with almost no homomorphic cost, and MC, ARK are linear combinations of ciphertexts with constant coefficients modulo 2. These packing choices minimize slot manipulation and provide a strong throughput behaviour. The requirement that N blocks are processed at a time leads however to poor latency behaviour.

A straightforward evaluation follows the original AES structure, resulting in multiplicative depth 7 (3 for SB and 4 for XORs, in view of $\mathrm{XOR}(x,y) = (x-y)^2$), plus additional depth for homomorphic cleaning [1, 6]. XBOOT [37] instead replaces XORs with homomorphic additions and restores correctness by modular reduction during discrete bootstrapping [7, 34], reducing the SB circuit depth to 3. We adopt the same idea for XOR to reduce latency.

3 Latency-oriented AES Algorithms for CKKS

Our Latency-aware approach ignores the GF(256) structure and sees AES purely as a sequence of bit operations. Our building blocks are latency-aware algorithms for the main AES round operations — SB, SR, MC —that are designed for efficient homomorphic evaluation. In this section, we focus on algorithmic aspects, with the HE computational model in mind; fine HE implementation aspects (in particular parameters) will be discussed in Section 5.

We focus on the case of a single AES block, which we write as $(b_i^{(j)})_{0\le i<8, 0\le j<16}$, where $b_i^{(j)}$ denotes the i-th bit of the j-th byte. One AES block $(b_i^{(j)})_{0\le i<8,0\le j<16}$ is naturally seen as a matrix. We turn this matrix with dimension order (bit, byte) into a vector using *row-major* flattening and obtain

$$\left(b_0^{(0)},\dots,b_0^{(15)},\dots,b_1^{(0)},\dots,b_1^{(15)},\dots,b_7^{(0)},\dots,b_7^{(15)}\right).$$

3.1 Latency-optimized SubBytes

We evaluate AES S-boxes using polynomial interpolation, writing the S-box applied to byte j in the form

$$f(\mathbf{x}) = \sum_{\mathbf{y}\in\{0,1\}^8} L_{\mathbf{y}}(\mathbf{x})\cdot\boldsymbol{\alpha}_{\mathbf{y}} : \{0,1\}^8 \to \{0,1\}^8. \tag{5}$$

where each $\boldsymbol{\alpha}_\ell = (\alpha_{i,\ell})_{0\le i<8} \in \mathbb{Z}^8$ is a vector, and the family $L_{\mathbf{y}}(\mathbf{x}) = \prod_{i=0}^{7}(x_i - y_i)$, for $\mathbf{y}\in\{0,1\}^8$, is the Lagrange basis of multivariate polynomials over $\{0,1\}^8$.

We start by studying the case of a single byte $(b_0,\dots,b_7)$ and assume that this byte is packed redundantly as $\mathbf{b} = (1)^{\times 32}\otimes(b_0,\dots,b_7)\in\{0,1\}^{256}$. We rely on the following Lemma:

Lemma 3.1. *Let $\mathbf{y}_0,\dots,\mathbf{y}_{255}$ be vectors in $\{0,1\}^8$, and let $\boldsymbol{\theta}_i = ((\mathbf{y}_j)_{i+j \bmod 8})_{0\le j<256}$, for $0\le i<8$.*
Then, one has

$$\prod_{i=0}^{7}(\mathrm{Rot}_i(\mathbf{b}) - \boldsymbol{\theta}_i) = (L_{\mathbf{y}_j}(\mathbf{b}))_{0\le j<256}. \tag{6}$$

Proof. For $0\le j<256$, the j-th coordinate of the left hand side of (6) is $\prod_{i=0}^{7}(b_{i+j \bmod 8} - (\mathbf{y}_j)_{i+j \bmod 8})$, which, up to reindexing, is $\prod_{i=0}^{7}(b_i - (\mathbf{y}_j)_i) = L_{\mathbf{y}_j}(\mathbf{b})$. □

Handling several bytes at once. We now handle 16 input bytes $\mathbf{b}^{(j)} = (b_i^{(j)})_{0\le i<8,0\le j<16}$ at once. We start from the vector $\mathbf{b}$ containing those 16 bytes in row-major packing and still assume that this data is replicated 32 times.

We replace Identity (6) by the identity

$$\prod_{i=0}^{7}\mathrm{Rot}_{16i}(\mathbf{b} - \boldsymbol{\theta}_i\otimes(1)^{\times 16}) = (L_{\mathbf{y}_\ell}(\mathbf{b}^{(j)}))_{0\le\ell<256,0\le j<16}, \tag{7}$$

meaning that we apply the previous algorithm independently for each byte.

Arbitrary output packing. Lemma 3.1 illustrates that we can obtain the Lagrange basis values in an arbitrary order in the output packing by adjusting the ordering of $\boldsymbol{\theta}_i$.

Evaluating SubBytes in Lagrange basis. We let $\mathbf{A}$ be the matrix $(\alpha_{i,\ell})_{0\le i<8,0\le\ell<256}\in\mathbb{Z}^{8\times 256}$ and $\mathbf{b}'$ be the vector produced by the Lagrange basis computation. Equation (5) is then a matrix-vector product $\mathbf{A}\cdot\mathbf{b}'$ and becomes $(\mathbf{A}\otimes\mathbf{I}_{16})\cdot\mathbf{b}'$ in the case of 16 bytes as an entry, where $\mathbf{A}\otimes\mathbf{I}_{16}\in\mathbb{Z}^{128\times 4096}$.

3.2 Redundant linear algebra

When aggressively targeting latency, we can take advantage of the redundancy of the input, as we did in the Lagrange basis method in the previous paragraph, by designing a variant of the classical diagonal method [28].

Let d, d', r, r' be four positive integers such that r divides d' and $r'd' = rd$. We consider a matrix-vector product $\mathbf{M}\cdot\mathbf{v}$, where $\mathbf{M} = (m_{ij})_{0\le i<d',0\le j<d}$ and $\mathbf{v} = (v_j)_{0\le j<d}$ have respective dimensions $d'\times d$, d.

We assume that input is in the redundant format $\tilde{\mathbf{v}} = \mathbf{v}\otimes(1)^{\times r} = (v_{\lfloor j/r\rfloor})_{0\le j<rd}\in\mathbb{R}^{rd}$. When we write m_{xy} or v_y, we assume that x is defined modulo d' and y is defined modulo d.

We build rd-dimensional extended diagonals of the matrix of the form

$$\Delta_\ell = (m_{i,\ell+\lfloor\frac{i}{r}\rfloor})_{0\le i<dr}, 0\le\ell<d'/r.$$

The i-th term of $\Delta_\ell\odot\mathrm{Rot}_{r\ell}(\tilde{\mathbf{v}})$ is $m_{i,\ell+\lfloor\frac{i}{r}\rfloor}v_{\lfloor\frac{i}{r}\rfloor+\ell}$. We deduce the following lemma:

Lemma 3.2. *Let $\mathbf{w} = \sum_{\ell=0}^{d'/r-1}\Delta_\ell\odot\mathrm{Rot}_{r\ell}(\tilde{\mathbf{v}})$. Then,*

$$\mathbf{v}' = \sum_{j=0}^{r'-1}\mathrm{Rot}_{d'j}(\mathbf{w}) = (1)^{\times r'}\otimes(\mathbf{M}\cdot\mathbf{v}).$$

This Lemma suggests a two step strategy, first evaluating $\mathbf{w}$, then deducing the result $\mathbf{v}'$.

From a computational perspective, $\mathbf{w}$ is evaluated by using the BSGS technique using Eq. (3). The final result, $\sum_{j=0}^{r'-1}\mathrm{Rot}_{d'j}(\mathbf{w})$ is evaluated using the rotate and sum technique for a cost of $\log r'$ rotations.

We obtain an algorithm which costs $\approx |S_1|+|S_2|+\log r'$ rotations; in the dense case, we obtain $|S_1|\approx|S_2|\approx\sqrt{d'/r}$ and a cost of $O(\sqrt{d'/r}+\log(r'))$ key-switchings.

We shall use this technique with $d'=128$, $r=4$, reducing $(|S_1|,|S_2|)$ from $(16,8)$ to $(8,4)$ at the cost of 2 final rotations; this

reduces the number[3] of key-switchings from 22 to 10, plus the two final rotations.

3.3 Connecting the dots

We now describe our latency-optimized circuit, which focuses on homomorphically evaluating a single AES block. To pack it into a single CKKS ciphertext of degree N, we use a *fewer-slot encoding* (explained in Section 2.2), which encodes the 128 bits into $N/2$ slots according to the following interleaved layout:

$$\mathbf{b} = (1)^{\times N/256} \otimes \left(b_0^{(0)}, \ldots, b_0^{(15)}, b_1^{(0)}, \ldots, b_1^{(15)}, b_7^{(0)}, \ldots, b_7^{(15)}\right).$$

We assume this layout as input to each AES round, and thus have to ensure that this structure holds at the end of each AES round.

3.3.1 SubBytes step. The fewer-slot layout directly provides the replicated inputs required for the Lagrange-based SubBytes evaluation (7). Recall that computing 16 input bytes of a single AES block simultaneously requires $16 \times 256 = 4096$ slots. In order to use the redundant linear algebra algorithm from Section 3.2, with $N/2$-slot ciphertext, we use redundant packing, which the same data are repeated $r = N/8192$ times. The replicated version of (7) can be expressed as:

$$\prod_{i=0}^{7} \mathrm{Rot}_{16i}\left(\mathbf{b} - (1)^{\times r} \otimes \boldsymbol{\theta}_i \otimes (1)^{\times 16}\right) = (1)^{\times r} \otimes (L_{\mathbf{y}_\ell}(\mathbf{b}^{(j)}))_{0 \le \ell < 256, 0 \le j < 16}.$$

The right-replication is used to handle all the bytes of the block at once, while the left-replication prepares the correct output structure for using the redundant linear algebra algorithm.

Homomorphic computation. The computation is realized homomorphically as follows. We first generate ciphertexts encrypting $\mathrm{Rot}_{16i}(\mathbf{b})$ for $0 \le i < 8$. This costs 7 rotations of a single input vector ("hoisted rotations", following [28]). Then we encode each $\mathrm{Rot}_{16i}\left((1)^{\times r} \otimes \boldsymbol{\theta}_i \otimes (1)^{\times 16}\right) \in \{0,1\}^{N/2}$ as a plaintext. Subtracting these plaintexts from the rotated ciphertexts requires 8 subtractions. Finally, the eight resulting ciphertexts are multiplied together in a tree-manner, requiring 7 ciphertext-ciphertext multiplications and consuming 3 multiplicative depth. In total, this step uses **7 hoisted rotations and 7 key switchings (ciphertext-ciphertext multiplication) and 3 multiplicative levels.**

3.3.2 Fused linear algebra. Seeing each byte as a vector of bits and interpreting XOR as addition modulo 2 (i.e., $x + y \pmod 2$), the two steps following SB in an AES round, SR and MC become matrix-vector operations modulo 2, as does the final step of SB in the Lagrange method. Following the XBOOT rationale, we see those steps as linear algebra steps over the integers, and perform modular reduction opportunistically.

The corresponding matrix, by which we multiply the Lagrange basis elements, is $\mathbf{M}_{\mathrm{AES}} = \mathbf{MC} \cdot \mathbf{SR} \cdot \mathbf{SB} \in \mathbb{Z}^{128 \times 4096}$. Since the matrix is rectangular, we lift it to a square block diagonal matrix as follows. We first split $\mathbf{M}_{\mathrm{AES}}$ column-wise into 32 square blocks of size 128: $\mathbf{M}_{\mathrm{AES}} = (\mathbf{M}_1 \mid \cdots \mid \mathbf{M}_{32}) \in \mathbb{Z}^{128 \times 4096}$. Splitting the input vector $\mathbf{v} \in \mathbb{Z}^{4096}$ accordingly as $\mathbf{v} = (\mathbf{v}_1 \mid \cdots \mid \mathbf{v}_{32})$ with $\mathbf{v}_i \in \mathbb{Z}^{128}$, the matrix vector product can be represented as: $\mathbf{M}_{\mathrm{AES}} \cdot \mathbf{v} = \sum_{i=1}^{32} \mathbf{M}_i \cdot \mathbf{v}_i$. From this decomposition, we introduce the block-diagonal matrix $\mathbf{D}_{\mathrm{AES}} = \mathrm{diag}(\mathbf{M}_1, \ldots, \mathbf{M}_{32}) \in \mathbb{Z}^{4096 \times 4096}$. By construction, $\mathbf{D}_{\mathrm{AES}} \cdot \mathbf{v} = (\mathbf{M}_1 \mathbf{v}_1) \mid \cdots \mid (\mathbf{M}_{32} \mathbf{v}_{32})$, and summing its 32 blocks recovers $\mathbf{M}_{\mathrm{AES}} \cdot \mathbf{v}$.

In the same fewer-slots setting, the StC matrix is, up to ordering, $\mathbf{I}_{N/256} \otimes \mathbf{DFT}_{128} \in \mathbb{C}^{N/2 \times N/2}$. We have:

$$\mathrm{StC} \cdot \mathbf{D}_{\mathrm{AES}} = \mathrm{diag}(\mathrm{StC} \cdot \mathbf{M}_1, \ldots, \mathrm{StC} \cdot \mathbf{M}_{32}) \in \mathbb{Z}^{4096 \times 4096},$$

meaning that $\mathbf{D}_{\mathrm{AES}}$ can be fully merged within StC: the resulting matrix has exactly 128 non-zero diagonals, which is the same as evaluating the StC matrix alone.

On input a vector $\mathbf{v} \in \mathbb{Z}^{4096}$, in order to obtain the redundant vector $(1)^{\times N/256} \otimes (\mathbf{D}_{\mathrm{AES}} \cdot \mathbf{v})$ as an output, we perform the following two-step strategy:

(1) $\mathbf{v}_{\mathrm{tmp}} = (\mathrm{StC} \cdot \mathbf{D}_{\mathrm{AES}}) \cdot \mathbf{v}$,
(2) $\mathbf{v}' = \sum_{i=0}^{31} \mathrm{Rot}_{128i}(\mathbf{v}_{\mathrm{tmp}})$.

For $N > 8192$, we perform Step (1) using Section 3.2 with repetition parameter $r = N/8192$. The number of diagonals is reduced to $128/r$ (In our implementation, $r = 4$, thus 32, see Section 5), while we combine the final step of Section 3.2 and the second step of the previous strategy as a single rotate-and-sum iteration summing $32r$ elements.

Skipping the final rotate-and-sum via bootstrapping. Interaction with bootstrapping allows us to go even further and suppress this final rotate-and-sum iteration, saving $\log(32r)$ sequential key-switchings. The purpose of this iteration is to gather contributions from replicated inputs and restore a fewer-slot ciphertext; this corresponds to the homomorphic evaluation of the trace map $\mathrm{Tr}_{R_{2^{14}}/R_{128}}$ (see Section 2.2), which is already performed within the bootstrapping procedure prior to CtS.[4]

Our Latency method uses complex encoding (so that r needs to be $\le N/8192$), $N = 2^{15}$ and $r = 4$, thus requiring **10 key-switchings and 1 multiplicative level** overall for the linear algebra step (see end of Section 3.2): added to the SubBytes cost, this yields a grand total of 24 key-switchings per AES round, with 2×7 hoisted key-switchings.

3.4 Extension to several blocks

We outline the adaptation of the previous methods to the case of several blocks.

Packing structure in the case of several blocks. In the case of several blocks, the input to the AES circuit becomes $(b_i^{(j,k)})_{i,j,k}$ for $0 \le i < 8$, $0 \le j < 16$, and $0 \le k < K$, which corresponds to a three-dimensional tensor, with dimensions ordered (bit, byte, block). We use row-major packing, meaning that the ordering becomes

$$\left(b_0^{(0,0)}, \ldots, b_0^{(0,K-1)}, \quad b_0^{(1,0)}, \ldots, b_0^{(1,K-1)}, \ldots, \quad b_0^{(15,0)}, \ldots, b_0^{(15,K-1)}, \ldots \quad b_7^{(0,0)}, \ldots, b_7^{(15,K-1)}\right). \tag{8}$$

Extension of the SubBytes method. The Lagrange basis method extends to this setting, up to changing $(1)^{\times 16}$ to $(1)^{\times 16K}$ in (7).

[3] In both cases, $0 \in S_1 \cap S_2$, which reduces the number of key-switchings to $|S_1| + |S_2| - 2$

[4] Since the explicit summation is now omitted, the scalar factor $32r$ must be absorbed into the preceding linear transform coefficients, so that the bootstrapping trace yields the correct result.

Structure of the matrices for several blocks. The matrices **SB**, **SR** and **MC** are built to operate at the block level. When turning to several blocks, we build extended matrices containing K copies of **SB**, **SR**, and **MC**.

Under this packing, the elements of a given block appear at positions distant from K. Each copy of **SB**, **SR**, **MC** thus acts on coordinates separated by K elements, and the matrices become $\overline{\mathbf{SB}}_K = \mathbf{SB} \otimes \mathbf{I}_K \in \mathbb{Z}^{128K \times 4096K}$, $\overline{\mathbf{SR}}_K = \mathbf{SR} \otimes \mathbf{I}_K \in \mathbb{Z}^{128K \times 128K}$, $\overline{\mathbf{MC}}_K = \mathbf{MC} \otimes \mathbf{I}_K \in \mathbb{Z}^{128K \times 128K}$.

3.4.1 Fusing with StC. For a matrix $\mathbf{A}$, let $\delta(\mathbf{A})$ be its number of non-zero diagonals. For the fusion of two linear algebra steps $\mathbf{A} \cdot (\mathbf{B} \cdot \mathbf{v})$ into $(\mathbf{A} \cdot \mathbf{B}) \cdot \mathbf{v}$, the compared costs of the two approaches (fused and unfused) are expected to be $\delta(\mathbf{A} \cdot \mathbf{B})$ vs $\delta(\mathbf{A}) + \delta(\mathbf{B})$ (without BSGS). For the fusion to be clearly beneficial, $\delta(\mathbf{A} \cdot \mathbf{B})$ should be of the same order of magnitude as $\max(\delta(\mathbf{A}), \delta(\mathbf{B}))$. Generically, however, we expect $\delta(\mathbf{A} \cdot \mathbf{B}) \approx \delta(\mathbf{A}) \cdot \delta(\mathbf{B})$ if $\mathbf{A}$ and $\mathbf{B}$ are sparse, unless the two matrices share a similar diagonal structure.

Fusing with StC thus requires a precise study of the interplay of the diagonal structure of the $\overline{\mathbf{MC}}_K \cdot \overline{\mathbf{SR}}_K$ (or $\overline{\mathbf{MC}}_K \cdot \overline{\mathbf{SR}}_K \cdot \overline{\mathbf{SB}}_K$) and the diagonal structure of blocks of the StC decomposition.

We start by a mathematical study of which type of matrices can be merged into the StC evaluation without increasing its total number of diagonals.

3.4.2 Fusing $\mathbf{M} \otimes \mathbf{I}$ with **DFT**. We claim that $\mathbf{M}_t \otimes \mathbf{I}_k$ can be merged into $\mathbf{DFT}_{tk}$, where $\mathbf{M}_t$ is a square matrix of size t and t, k are powers of two by decomposing the $\mathbf{DFT}_{tk}$ matrix via the FFT factorization. By Equation (1), $\mathbf{DFT}_{tk}$ can be decomposed into $\log tk$ sparse matrices $D_{tk} \cdot D_{tk/2} \cdots D_2$. The first $\log k$ matrices with a small block size collapse into the block-diagonal matrix $\mathbf{D}_{in}$, while the last $\log t$ matrices with larger block size collapse into a matrix $\mathbf{D}_{out}$ whose nonzero diagonal indices are multiple of k. In summary,

$$\mathbf{DFT}_{tk} = \mathbf{D}_{out} \cdot \mathbf{D}_{in}, \qquad \mathbf{D}_{in} = \mathbf{I}_t \otimes \mathbf{DFT}_k, \qquad \delta(\mathbf{D}_{out}) = k \cdot \mathbb{Z}_t.$$

In view of the properties of the Kronecker product, $\mathbf{M}_t \otimes \mathbf{I}_k$ commutes with $\mathbf{D}_{in}$, as

$$(\mathbf{I}_t \otimes \mathbf{DFT}_k) \cdot (\mathbf{M}_t \otimes \mathbf{I}_k) = \mathbf{M}_t \otimes \mathbf{DFT}_k = (\mathbf{M}_t \otimes \mathbf{I}_k) \cdot (\mathbf{I}_t \otimes \mathbf{DFT}_k).$$

Substituting into $\mathbf{DFT}_{tk} \cdot (\mathbf{M}_t \otimes \mathbf{I}_k)$ yields

$$\begin{aligned}\mathbf{DFT}_{tk} \cdot (\mathbf{M}_t \otimes \mathbf{I}_k) &= \mathbf{D}_{out} \cdot \mathbf{D}_{in} \cdot (\mathbf{M}_t \otimes \mathbf{I}_k) \\ &= \big(\mathbf{D}_{out} \cdot (\mathbf{M}_t \otimes \mathbf{I}_k)\big) \cdot \mathbf{D}_{in}.\end{aligned}$$

Since both $\delta(\mathbf{M}_t \otimes \mathbf{I}_k)$ and $\delta(\mathbf{D}_{out})$ are contained in $k \cdot \mathbb{Z}_t$, we have $\delta\big(\mathbf{D}_{out} \cdot (\mathbf{M}_t \otimes \mathbf{I}_k)\big) \subseteq k \cdot \mathbb{Z}_t = \delta(\mathbf{D}_{out})$. Hence $\mathbf{M}_t \otimes \mathbf{I}_k$ is merged into $\mathbf{D}_{out}$ at no extra cost.

3.4.3 Extension to fewer-slot packing. When the input is repeated L times, the matrix takes the form $\mathbf{I}_L \otimes \mathbf{M}_t \otimes \mathbf{I}_k$. This can be fused with $\mathbf{I}_L \otimes \mathbf{DFT}_{tk}$, which corresponds to the first stage of the FFT factorization of $\mathbf{DFT}_{Ltk}$:

$$(\mathbf{I}_L \otimes \mathbf{DFT}_{tk}) \cdot (\mathbf{I}_L \otimes \mathbf{M}_t \otimes \mathbf{I}_k) = \mathbf{I}_L \otimes \big(\mathbf{DFT}_{tk} \cdot (\mathbf{M}_t \otimes \mathbf{I}_k)\big).$$

The inner factor $\mathbf{DFT}_{tk} \cdot (\mathbf{M}_t \otimes \mathbf{I}_k)$ is then fused exactly as in the previous subsection.

3.4.4 Extension to block-diagonal matrices. The same idea extends to block-diagonal matrices: instead of $\mathbf{I}_L \otimes \mathbf{M}_t \otimes \mathbf{I}_k$, we can fuse $\mathrm{diag}(\mathbf{M}_0, \ldots, \mathbf{M}_{L-1}) \otimes \mathbf{I}_k$ with $\mathbf{I}_L \otimes \mathbf{DFT}_{tk}$, where $\mathbf{M}_i \in \mathbb{C}^{t \times t}$ for $i \in [L]$.

3.4.5 Extension to the conjugate-invariant ring. The fusion technique described in this section extends to the context where the conjugate invariant technique [33] is used to obtain N real slots in CKKS rather than $N/2$ complex ones. We did not use it in the Latency algorithm, as our estimates suggest that it would degrade the latency significantly. As the fusion of linear layers with StC in the conjugate invariant context remains of independent interest, we include the discussion in Appendix C.

3.4.6 Application to Several Blocks. Given the packing structure for several blocks (Equation (8)), the ciphertext goes from 128 distinct slots to $128K$ slots. As a consequence, StC becomes $\mathbf{I}_{N/256K} \otimes \mathbf{DFT}_{128K}$, while the fused steps SB, SR and MC give rise to the fused matrix $\mathbf{M}_{\mathrm{AES}} \otimes \mathbf{I}_K \in \mathbb{Z}^{128K \times 4096K}$. As in the previous case, we decompose $\mathbf{M}_{\mathrm{AES}}$ into 32 $128K \times 128K$ blocks, and build the $4096K \times 4096K$ matrix $\mathbf{D}_{\mathrm{AES}} \otimes \mathbf{I}_K$. By using a 2-step DFT decomposition for StC as $\mathbf{DFT}_{128K} = (\mathbf{DFT}_{128} \otimes I_K) \cdot (\mathbf{I}_{128} \otimes \mathbf{DFT}_K)$, we observe following Section 3.4.4 that the multiplication by $\mathbf{D}_{\mathrm{AES}}$ commutes with the first step of decomposition and can be fused with the second step of the decomposition at no additional cost.

If $N > 8192K$, redundancy can be used as in the 1-block case, and fusion of rotate-and-sum and bootstrapping still holds.

Our $\mathrm{Latency}_4$ circuit uses this method with $K = 4$, $N = 2^{15}$ and complex encoding (hence no redundancy). It thus requires 22 key-switchings for the linear step, to which we must add the 2 key-switchings required by $\mathbf{I}_{128} \otimes \mathbf{DFT}_4$, for a total of **24 key-switchings and 2 multiplicative levels** and a grand total of 38 key-switchings, with 7, then 15 being hoisted.

4 Optimizing Throughput via Finite Field Encoding

In this section, rather than relying on CKKS slots to represent bits, we describe a method allowing to natively encode finite fields elements of GF(16) into CKKS message space, giving direct access to homomorphic computations over GF(16) using CKKS. The same idea has been simultaneously described, with a different focus and a more systematic study of its capabilities, in [38].

This idea, in turn, gives a description of GF(256) as a pair of elements of GF(16) allowing to view the SubBytes step as an inverse, using the underlying mathematical structure to evaluate it in a much more efficient way than via polynomial interpolation.

Similar ideas would yield a direct encoding of GF(256) into CKKS. We show in Section 4.8 that such a choice leads to numerically unstable computations for AES evaluation and should thus be avoided; it also leads to a deeper and less efficient inversion algorithm.

4.1 Encoding GF(16) elements

We let $\Phi_5(X) = X^4 + X^3 + X^2 + X + 1$ be the 5-th cyclotomic polynomial. As Φ_5 is irreducible over GF(2), we can define

$$\mathrm{GF}(16) = \mathrm{GF}(2)[X]/(\Phi_5(X))$$

which realizes elements of GF(16) as polynomials of degree ≤ 3 over GF(2). Seeing bits as integers, we obtain a representation of elements of GF(16) as elements of the ring $\mathcal{R}' = \mathbb{Z}[X]/(\Phi_5(X))$, i.e. as polynomials of degree ≤ 3 with integer coefficients, where polynomial coefficients are to be taken modulo 2.

Let $\zeta_5 = \exp(2I\pi/5)$ be a primitive 5th root of unity. The ring $\mathcal{R}'$ is the 5-th cyclotomic ring, with canonical embedding

$$T_5 : \mathbb{Z}[X]/(\Phi_5(X)) \to \mathbb{C}^2. \quad \widetilde{m}(X) \mapsto (\widetilde{m}(\zeta_5), \widetilde{m}(\zeta_5^2)).$$

In the sequel, to help the reader navigate the rings and maps, we shall use the notations:

- m for elements of GF(16);
- $\widetilde{m}$ for elements of $\mathbb{Z}[X]/(\Phi_5(X))$, or a lift of $m \in$ GF(16);
- $z \in \mathbb{C}^2$ for elements of $T_5(\mathbb{Z}[X]/(\Phi_5(X)))$.

Concerning the second point, note that the lifting choice is non-canonical: various lifts can be chosen. For the sake of intuition, one can consider that elements of GF(2) are lifted to $\{0, 1\}$. We shall discuss this question in more detail in section 4.8.

We shall use boldface for vectors of elements of GF(16), $\mathbb{Z}[X]/\Phi_5(X)$, and $\mathbb{C}^2$.

If $z, z' \in \mathbb{C}^2$, we use the notation $z \equiv z'$ for the relation $T_5^{-1}(z) = T_5^{-1}(z')$ mod 2, namely z and z' encode the same GF(16) element. If z, z' are produced through the encoding process, this means that $z = z'$; however, if z or z' is the result of computations this needs not be true until modular reduction modulo 2 is performed.

4.2 Packing $N/4$ elements of GF(16) in a CKKS ciphertext

The T_5 map sends $\mathbb{Z}[X]/(\Phi_5(X))$ to $\mathbb{C}^2$. As CKKS message space is $\mathbb{C}^{N/2}$, we map $N/4$ elements of $\mathbb{Z}[X]/(\Phi_5(X))$ to a CKKS message as $N/4$ slots on which we can compute in parallel. Let $\mathbf{m} = (m_i)_{0 \le i < N/4-1} \in$ GF(16)$^{N/4}$ and $\widetilde{m}$ be its lift to $(\mathbb{Z}[X]/(\Phi_5(X)))^{N/4}$. Rather than the natural packing, i.e.,

$$(\widetilde{m_0}, \widetilde{m_1}, \ldots, \widetilde{m_{N/4-1}}) \mapsto (T_5(\widetilde{m_0}), \ldots, T_5(\widetilde{m_{N/4-1}}))$$

we use the packing

$$T_5^{N/4} : \widetilde{\mathbf{m}} = (\widetilde{m_i})_{0 \le i < N/4} \mapsto (\widetilde{\mathbf{m}}(\zeta_5) || \widetilde{\mathbf{m}}(\zeta_5^2)), \tag{9}$$

for efficiency reasons; see Sections 4.4.2 and 4.9.2.

4.3 Elementary homomorphic operations

Arithmetic operations. Let $\widetilde{\mathbf{m}_1}, \widetilde{\mathbf{m}_2} \in (\mathbb{Z}[X]/(\Phi_5(X)))^{N/4}$ be two vectors of elements of GF(16) lifted to the integers. As the map $T_5^{N/4}$ is a ring homomorphism from $(\mathbb{Z}[X]/(\Phi_5(X)))^{N/4}$ to $\mathbb{C}^{N/2}$, and CKKS encoding gives an approximate ring homomorphism Ecd from $\mathbb{C}^{N/2}$ to $\mathbb{Z}[X]/(X^N + 1)$, we have

$$\mathsf{Ecd} \circ T_5^{N/4}(\widetilde{\mathbf{m}_1} \diamond \widetilde{\mathbf{m}_2}) \equiv (\mathsf{Ecd} \circ T_5^{N/4}(\widetilde{\mathbf{m}_1})) \diamond (\mathsf{Ecd} \circ T_5^{N/4}(\widetilde{\mathbf{m}_2})),$$

for $\diamond \in \{+, \odot\}$. CKKS homomorphic properties make this identity extend to the encryption layer, meaning that homomorphic operations on CKKS messages $T_5^{N/4}(\widetilde{\mathbf{m}_1})$ and $T_5^{N/4}(\widetilde{\mathbf{m}_2})$ implement homomorphic addition and multiplication over $(\mathbb{Z}[X]/\Phi_5(X))^{N/4}$; up to reduction modulo 2, we thus have an homomorphic implementation of GF(16) arithmetic.

Conjugation. We observe that the conjugation operation of CKKS message space has a natural interpretation in GF(16). Indeed, for $m \in$ GF(16), $T_5(\widetilde{m}) = (\widetilde{m}(\zeta), \widetilde{m}(\zeta^2))$, so that

$$\overline{T_5(\widetilde{m})} = (\overline{\widetilde{m}(\zeta)}, \overline{\widetilde{m}(\zeta^2)}) = (\widetilde{m}(\zeta^4), \widetilde{m}(\zeta^3)).$$

As $\zeta^3 = (\zeta^2)^4$, and given the fact that $x \mapsto x^4$ is linear over GF(16), we thus have

$$\overline{T_5(\widetilde{m})} \equiv T_5(\widetilde{m}^4),$$

so that evaluating conjugation on CKKS message space corresponds to the map $m \mapsto m^4$ in GF(16) (up to reduction modulo 2).

4.4 Linear maps under the GF(16) encoding

Since almost every step of an AES round outside the S-box map is a GF(2)-linear map acting in GF(16)$^{N/4}$, we discuss these maps in detail and show that they admit a cheap and uniform realization: a single real-linear map on the CKKS slots, evaluated with a few rotations and one conjugation.

4.4.1 Action on a single GF(16) *element.* We start with the case of a single element. We consider the map $(T_5, \overline{T_5})$, which goes from $\mathbb{Z}[X]/(\Phi_5(X))$ to $\mathbb{C}^4$ and sends $\widetilde{m} \in \mathbb{Z}[X]/(\Phi_5(X))$ to the pair $(z, \overline{z}) = (T_5(\widetilde{m}), \overline{T_5(\widetilde{m})})$. In the canonical bases, this map has the matrix

$$\mathbf{M}_5 = (\zeta_5^{j2^i})_{0 \le i < 4, 0 \le j < 4} \in \mathbb{C}^{4\times 4},$$

and $\mathbf{M}_5$, so that for $z \in T_5(\mathbb{Z}[X]/(\Phi_5(X)))$, $T_5^{-1}(z)$ has coordinates $\mathbf{M}_5^{-1} \begin{pmatrix} z \\ \overline{z} \end{pmatrix}$ in the canonical basis $1, X, X^2, X^3$.

We now have the following lemma:

Lemma 4.1. *Let $L :$ GF(16) $\to$ GF(16) be an* GF(2)*-linear map, and $\mathbf{M}_L$ be its matrix in the polynomial basis $1, X, X^2, X^3$. Let $\widetilde{\mathbf{M}_L} \in \mathbb{Z}^{4\times 4}$ be such that $\widetilde{\mathbf{M}_L} \equiv \mathbf{M}_L$ mod 2, and $\widetilde{L}$ be the corresponding linear map over $\mathbb{Z}[X]/(\Phi_5(X))$.*

Let $(\mathbf{A}_L, \mathbf{B}_L)$ be the 2×2 blocks of the first two rows of the matrix $\mathbf{M}_5 \cdot \widetilde{\mathbf{M}}_L \cdot \mathbf{M}_5^{-1} \in \mathbb{C}^{4\times 4}$. Then, for all $\widetilde{m} \in \mathbb{Z}[X]/(\Phi_5(X))$, one has

$$T_5(\widetilde{L}(\widetilde{m})) \equiv \mathbf{A}_L \cdot T_5(\widetilde{m}) + \mathbf{B}_L \cdot \overline{T_5(\widetilde{m})}. \tag{10}$$

Proof. We write $T_5(\widetilde{L}(\widetilde{m})) \equiv T_5 \circ \widetilde{L} \circ T_5^{-1}(T_5(\widetilde{m}))$. The result follows from

$$\mathbf{M}_5 \cdot \widetilde{\mathbf{M}}_L \cdot \mathbf{M}_5^{-1} \cdot \begin{pmatrix} T_5(\widetilde{m}) \\ \overline{T_5(\widetilde{m})} \end{pmatrix} \equiv \begin{pmatrix} T_5(\widetilde{L}(\widetilde{m})) \\ \overline{T_5(\widetilde{L}(\widetilde{m}))} \end{pmatrix}.$$

□

In more explicit terms, the map L can be homomorphically evaluated at $m \in$ GF(16) by evaluating $\mathbf{A}_L$ and $\mathbf{B}_L \circ (z \mapsto \overline{z})$ on the slots corresponding to $\widetilde{m}$ in the CKKS message space. We note again that equality only holds up to reduction modulo 2. In particular, the matrices $\mathbf{A}_L, \mathbf{B}_L \in \mathbb{C}^{2\times 2}$ are not uniquely defined, and can be replaced by $\mathbf{A}'_L \in \mathbf{A}_L + 2(T_5(\mathbb{Z}^4 \times \mathbb{Z}^4)), \mathbf{B}'_L \in \mathbf{B}_L + 2(T_5(\mathbb{Z}^4 \times \mathbb{Z}^4))$.

4.4.2 Action on GF(16)$^{N/4}$. The previous discussion extends to $N/4$ GF(2)-linear maps over GF(16)$^{N/4}$ evaluated in an SIMD manner. Let

$$\mathcal{L} = (L_0, L_1, \ldots, L_{N/4-1})$$

be such a map, acting on an $N/4$-uple of elements $\mathbf{m} = (m_i)_{0 \le i < N/4} \in$ (GF(16))$^{N/4}$. By applying the previous discussion blockwise under the packing (9), we obtain the two matrices

$$\mathcal{A}_\mathcal{L}, \mathcal{B}_\mathcal{L} \in \mathbb{C}^{N/2\times N/2},$$

where $\mathcal{A}_{\mathcal{L}}$ is defined by its $2x2$ blocks at position $\{i, i+N/4\} \times \{i, i+N/4\}$:

$$\begin{pmatrix} \mathcal{A}_{\mathcal{L}\,i,i} & \mathcal{A}_{\mathcal{L}\,i,i+N/4} \\ \mathcal{A}_{\mathcal{L}\,i+N/4,i} & \mathcal{A}_{\mathcal{L}\,i+N/4,i+N/4} \end{pmatrix} = \mathbf{A}_{L_i},$$

with 0s elsewhere, while $\mathcal{B}_{\mathcal{L}}$ is defined similarly from the $\mathbf{B}_{L_i}$.

For $(m_0, \ldots, m_{N/4-1}) \in \mathrm{GF}(16)^{N/4}$, if

$$\mathbf{z} = T_5^{N/4}(\widetilde{m_0}, \ldots, \widetilde{m_{N/4-1}}) \in \mathbb{C}^{N/2}$$

is the corresponding CKKS message, in view of Lemma 4.1, we have

$$T_5^{N/4}(\widetilde{\mathcal{L}}(\widetilde{m_0}, \ldots, \widetilde{m_{N/4-1}})) \equiv \mathcal{A}_{\mathcal{L}}\,\mathbf{z} + \mathcal{B}_{\mathcal{L}}\,\overline{\mathbf{z}}:$$

Homomorphically evaluating $\mathcal{L}$ on $\mathrm{GF}(16)^{N/4}$, up to reduction modulo 2, corresponds to computing the right hand side on this equation in CKKS message space.

The matrices $\mathcal{A}_{\mathcal{L}}$ and $\mathcal{B}_{\mathcal{L}}$ have two non-trivial diagonals: non-zero terms have $i-j \in \{0, N/4, -N/4\}$ and $N/4 = -N/4 \bmod N/2$. Letting $\Delta_d(\cdot)$ be the d-th diagonal of a matrix, we obtain

$$\mathcal{A}_{\mathcal{L}}\,\mathbf{z} + \mathcal{B}_{\mathcal{L}}\,\overline{\mathbf{z}} = \sum_{d\in\{0,N/4\}} \Delta_d(\mathcal{A}_{\mathcal{L}}) \odot \mathrm{Rot}_d(\mathbf{z}) + \overline{\sum_{d\in\{0,N/4\}} \overline{\Delta_d(\mathcal{B}_{\mathcal{L}})} \odot \mathrm{Rot}_d(\mathbf{z})}. \tag{11}$$

This formula can then be evaluated homomorphically using only two key-switchings: one rotation and one conjugation, as described in Algorithm 1.

Algorithm 1 PCMv: GF(2)-linear maps in GF(16)

Input: ct $\in \mathcal{R}_Q^2$ with $(T_5^{-1})^{N/4} \circ \mathrm{Dcd} \circ \mathrm{Dec}(\mathrm{ct}) = (m_{0,i}, m_{1,i}, m_{2,i}, m_{3,i})_{0\le i<N/4} \in \mathbb{Z}^{N/4}$.
Input: A vector of $N/4$ GF(2)-linear maps over GF(16) given as the 4 plaintexts encoding $\Delta_0(\mathcal{A}_L)$, $\Delta_{N/4}(\mathcal{A}_L)$, $\overline{\Delta_0(\mathcal{B}_L)}$, $\overline{\Delta_{N/4}(\mathcal{B}_L)}$.
Output: $\mathrm{ct}' \in \mathcal{R}_{Q/\Delta}^2$ with $(T_5^{-1})^{N/4} \circ \mathrm{Dcd} \circ \mathrm{Dec}(\mathrm{ct}') = (L_i(m_{0,i} + m_{1,i}X + m_{2,i}X^2 + m_{3,i}X^3)_{0\le i<N/4} \bmod 2 \in \mathrm{GF}(16)^{N/4}$.

1: $\mathrm{ct}_{N/4} \leftarrow \mathrm{Rot}_{N/4}(\mathrm{ct})$
2: $\mathrm{ct}_{\mathcal{A}} \leftarrow \mathsf{Mult}(\Delta_0(\mathcal{A}_L), \mathrm{ct}) + \mathsf{Mult}(\Delta_{N/4}(\mathcal{A}_L), \mathrm{ct}_{N/4})$
3: $\mathrm{ct}_{\mathcal{B}} \leftarrow \mathsf{Mult}(\overline{\Delta_0(\mathcal{B}_L)}, \mathrm{ct}) + \mathsf{Mult}(\overline{\Delta_{N/4}(\mathcal{B}_L)}, \mathrm{ct}_{N/4})$.
4: **return** $\mathrm{ct}_{\mathcal{A}} + \overline{\mathrm{ct}_{\mathcal{B}}}$.

4.4.3 Frobenius automorphisms. An important particular case is the case of the square map. For $m \in \mathrm{GF}(16)$, under our encoding, the square map $F : m \mapsto m^2$ takes the form:

$$T_5(\widetilde{F(m)}) \equiv \left(\widetilde{m}(\zeta_5^2), \widetilde{m}(\zeta_5^4)\right) \equiv \left(\widetilde{m}(\zeta_5^2), \overline{\widetilde{m}(\zeta_5)}\right), \tag{12}$$

In view of this equation, we can take $\mathbf{A}_F = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \mathbf{B}_F = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$ in [10][5].

This description extends to the 8-th power map $x \mapsto x^8$ which becomes, under the T_5 encoding, $(\widetilde{m}(\zeta_5), \widetilde{m}(\zeta_5^2)) \mapsto (\overline{\widetilde{m}(\zeta_5^2)}, \widetilde{m}(\zeta_5))$.

[5]The proof of the Lemma yields the same result when lifting GF(2) to $\mathbb{Z}$ as $\{0, 1\}$

4.5 Bootstrapping elements of GF(16)

Bootstrapping elements of GF(16) encoded as previously described could be accomplished via ordinary, complex CKKS bootstrapping. However, this choice would lead us to never perform reduction modulo 2, and would lead to progressive increase of the values in the CKKS slots, requiring higher precision and quickly leading to numerical instability.

Directly using CKKS binary bootstrapping provides reduction modulo 2, but operates on the slots of the CKKS message, i.e., on values encoded under the T_5 map; reducing these values modulo 2 is algebraically meaningless.

We rely on the CKKS binary bootstrapping, to enjoy modular reduction modulo 2, but correct the bootstrapping process so as to make the reduction modulo 2 operate on the correct object. A similar situation is already described (along with a more complicated bootstrapping process that we do not require), for similar reasons, in [23].

CKKS binary bootstrapping, on input a ciphertext $\mathrm{ct} \in \mathcal{R}_q^2$ decrypting and (approximately) decoding to $\mu \in \mathbb{Z}^{N/2} + I\mathbb{Z}^{N/2}$, returns $\mathrm{ct}' \in \mathcal{R}_Q^2$, $Q > q$, which decrypts and (approximately) decodes to $\mu' = \mu \bmod 2 \in \{0, 1, I, 1+I\}^{N/2}$. To this end, one evaluates the SlotToCoefficients operation in order to turn ct to $\mathrm{ct}_{\mathrm{coeff}} = (a, b) \in R_{q_0}^2$ decrypting to

$$\sum_{i=0}^{N/2-1} \Delta \cdot (\mathrm{Re}(\mu_i) + X^{N/2}\mathrm{Im}(\mu_i))X^i \bmod q_0.$$

By setting $\Delta = q_0/2$, one obtains reduction modulo 2 of the values μ_i, a property that is preserved by the following bootstrapping steps.

Starting with a CKKS message encrypting $T_5^{N/4}(\widetilde{\mathbf{m}})$, where $\widetilde{\mathbf{m}} \in (\mathbb{Z}[X]/(\Phi_5(X))^{N/4}$, we evaluate the map T_5^{-1} on pairs of slots to bring the coefficients of $\widetilde{\mathbf{m}}$ into the slots of a CKKS ciphertext. This map is evaluated, for $\mathbf{z} \in (\mathbb{C}^{N/2})$, as

$$\mathbf{z} \mapsto \left(\begin{pmatrix} 1 & I & 0 & 0 \\ 0 & 0 & 1 & I \end{pmatrix} \cdot \mathbf{M}_5^{-1}\right) \otimes \mathbf{I}_{N/4} \cdot \begin{pmatrix} \mathbf{z} \\ \overline{\mathbf{z}} \end{pmatrix};$$

if $(\widetilde{m}_{i,0}, \widetilde{m}_{i,1}, \widetilde{m}_{i,2}, \widetilde{m}_{i,3}) = T_5^{-1}(\mathbf{z}_i)$, this brings $\widetilde{m}_{i,0} + I\widetilde{m}_{i,1}, \widetilde{m}_{i,2} + I\widetilde{m}_{i,3}$ in the slots of the CKKS message. We let

$$\mathbf{S}_5 = \left(\begin{pmatrix} 1 & I & 0 & 0 \\ 0 & 0 & 1 & I \end{pmatrix} \cdot \mathbf{M}_5^{-1}\right) \otimes \mathbf{I}_{N/4} = (\mathbf{S}_5^{(\ell)} || \mathbf{S}_5^{(r)}),$$

where $\mathbf{S}_5^{(\ell)}, \mathbf{S}_5^{(r)} \in \mathbb{C}^{N/2\times N/2}$.

Applying CKKS binary bootstrapping (see [7]) we perform reduction modulo 2 of the CKKS slots and obtain $\mathrm{ct}' \in \mathcal{R}_Q^2$ decrypting and decoding to $(\widetilde{m}_{i,0} + I\widetilde{m}_{i,1}, \widetilde{m}_{i,2} + I\widetilde{m}_{i,3})_{0\le i<N/4} \bmod 2$.

We let

$$\mathbf{S}_5'^{(\ell)} = \frac{1}{2}\begin{pmatrix} 1 - I\zeta_5 & \zeta_5^2(1 - I\zeta_5) \\ 1 - I\zeta_5^2 & \zeta_5^4(1 - I\zeta_5^2) \end{pmatrix} \otimes \mathbf{I}_{N/4},$$

$$\mathbf{S}_5'^{(r)} = \frac{1}{2}\begin{pmatrix} 1 + I\zeta_5 & \zeta_5^2(1 + I\zeta_5) \\ 1 + I\zeta_5^2 & \zeta_5^4(1 + I\zeta_5^2) \end{pmatrix} \otimes \mathbf{I}_{N/4}.$$

For $\mathbf{z} \in \mathbb{C}^{N/2}$, we conclude by evaluating the map

$$\mathbf{z} \mapsto \frac{1}{2}\begin{pmatrix} 1 & \zeta_5^2 \\ 1 & \zeta_5^4 \end{pmatrix} \otimes \mathbf{I}_{N/4} \cdot (\mathbf{z} + \overline{\mathbf{z}}) - \frac{I}{2}\begin{pmatrix} \zeta_5 & \zeta_5^3 \\ \zeta_5^2 & \zeta_5 \end{pmatrix} \otimes \mathbf{I}_{N/4} \cdot (\mathbf{z} - \overline{\mathbf{z}}),$$
$$= \mathbf{S}_5'^{(\ell)} \cdot \mathbf{z} + \mathbf{S}_5'^{(r)} \cdot \overline{\mathbf{z}}.$$

which brings in the slots of the CKKS message the values $T_5^{N/4}(\widetilde{\mathbf{m}} \bmod 2)$. This completes the bootstrapping and reduces modulo 2 the underlying elements of $\mathbb{Z}[X]/(\Phi_5(X))$. We summarize this process as Algorithm 2.

Algorithm 2 Bootstrapping elements of GF(16)

Input: ct $\in \mathcal{R}_q^2$ with $T_5^{N/4-1} \circ \mathrm{Dcd} \circ \mathrm{Dec}(\mathrm{ct}) = (m_{0,i}, m_{1,i}, m_{2,i}, m_{3,i})_{0 \le i < N/4} \in \mathbb{Z}^{N/4}$.

Output: ct′ $\in \mathcal{R}_Q^2$ with $T_5^{N/4-1} \circ \mathrm{Dcd} \circ \mathrm{Dec}(\mathrm{ct}') = (m_{0,i}, m_{1,i}, m_{2,i}, m_{3,i})_{0 \le i < N/4} \bmod 2 \in \{0,1\}^{N/4}$.

1: $\mathrm{ct}_{\mathbf{m}} \leftarrow \mathrm{PCMv}\left(\Delta_0(\mathrm{S}_5^{(\ell)}), \Delta_{N/4}(\mathrm{S}_5^{(\ell)}), \overline{\Delta_0(\mathrm{S}_5^{(r)})}, \overline{\Delta_{N/4}(\mathrm{S}_5^{(r)})}, \mathrm{ct}\right)$
2: $\mathrm{ct}'_{\mathbf{m}} \leftarrow \mathrm{BinBoot}(\mathrm{ct}_{\mathbf{m}})$ ▷ with reduction modulo 2
3: $\mathrm{ct}' \leftarrow \mathrm{PCMv}\left(\Delta_0(\mathrm{S}'^{(\ell)}_5), \Delta_{N/4}(\mathrm{S}'^{(\ell)}_5), \overline{\Delta_0(\mathrm{S}'^{(r)}_5)}, \overline{\Delta_{N/4}(\mathrm{S}'^{(r)}_5)}, \mathrm{ct}'_{\mathbf{m}}\right)$
4: **return** ct′

4.6 Moving to GF(256)

4.6.1 Encoding elements of GF(256). We build on the encoding of GF(16) described in the previous sections to represent elements of GF(256). As a definition of the latter, we use

$$\begin{aligned}\mathrm{GF}(256) &= \mathrm{GF}(16)[U]/(U^2+U+\beta)\\ &= \mathrm{GF}(2)[X,U]/(\Phi_5(X), U^2+U+\beta),\end{aligned} \tag{13}$$

where $\beta \in \mathrm{GF}(16)$ is chosen so that $U^2+U+\beta$ is irreducible over GF(16).

We note that AES uses the definition

$$\mathrm{GF}(256)_{\mathrm{AES}} = \mathrm{GF}(2)[\gamma]/(\gamma^8+\gamma^4+\gamma^3+\gamma+1);$$

if $\Gamma \in \mathrm{GF}(256)$ is a root of the polynomial $X^8+X^4+X^3+X+1$ in the field $\mathrm{GF}(16)[U]/(U^2+U+\beta)$, the map $\gamma \mapsto \Gamma$ gives an explicit isomorphism between these two representations, which can be used to embed AES $\mathrm{GF}(256)_{\mathrm{AES}}$ elements into $\mathrm{GF}(16)[U]/(U^2+U+\beta)$.

Under the representation (13), an element $a+bU$ of GF(256) can then be seen as a pair $[a,b]$ of elements of GF(16), itself mapped to an element of $\mathbb{C}^4$. For $B=[a,b] \in \mathrm{GF}(256)$, $(i,j) \in \{0,1\}^2$, this embedding is described by the maps: $\alpha_{ij} : \mathrm{GF}(256) \to \mathbb{C}$ defined by

$$\alpha_{00}(B) = a(\zeta_5),\ \alpha_{01}(B) = a(\zeta_5^2),\ \alpha_{10}(B) = b(\zeta_5),\ \alpha_{11}(B) = b(\zeta_5^2).$$

We further let $\boldsymbol{\alpha} = (\alpha_{00}, \alpha_{10}, \alpha_{01}, \alpha_{11})$.

4.6.2 Inversion in GF(256). For $B=[a,b] \in \mathrm{GF}(256)$, we define its conjugate B^c to be

$$B^c = [a+b, b]$$

and its norm to be $\nu = B \cdot B^c \in \mathrm{GF}(16)$. We define the *quasi-inverse* of B to be $(B^{-1})^c$, and use the identity $(B^{-1})^c = \nu^{-1} \cdot B = \nu^{14} \cdot B$ to reduce the quasi-inverse computation to arithmetic in GF(16), obtaining Algorithm 3. The missing conjugation will be fused into the linear layer (see Section 4.7.2) following SubBytes.

The packing structure $\boldsymbol{\alpha}$ makes masking necessary in the evaluation of the algorithm; to make this fact explicit, we chose to describe the algorithm as it is implemented within CKKS, i.e., under the $\boldsymbol{\alpha}$ encoding, manipulating elements of $\mathbb{C}^4$ rather than pairs of elements of GF(16). In the latter view, the algorithm could be summarized as $n \leftarrow \nu$, $m \leftarrow n \cdot n^4$, $m' \leftarrow m^2$, and the result is obtained as $(B \cdot n^4) \cdot m'$.

Algorithm 3 Quasi-inversion in GF(256)

Input: $\boldsymbol{\alpha}(B) \in \mathbb{C}^4$, where $B=[a,b] \in \mathrm{GF}(256)$.

Output: $x \equiv \boldsymbol{\alpha}(B') \in \mathbb{C}^4$, where $B' = [a',b']$ with $B^{-1} = [a'+b', b'] = B'^c$.

1: $v \leftarrow \mathrm{Rot}_1(\boldsymbol{\alpha}(B))$
2: $n_1 \leftarrow v \odot \boldsymbol{\alpha}(B)$
3: $n_2 \leftarrow \boldsymbol{\alpha}([a^2+\beta b^2, a^2+\beta b^2])$ ▷ GF(2)-linear map
4: $n \leftarrow n_1 + n_2$
5: $m \leftarrow n \odot \overline{n}$ ▷ $\in \mathbb{R}^4$
6: $m' \leftarrow \mathrm{Rot}_2(m)$
7: $\mu_1 \leftarrow \boldsymbol{\alpha}(B) \odot (1,0,1,0)$
8: $\mu_2 \leftarrow \mathrm{Rot}_1(\boldsymbol{\alpha}(B)) \odot (1,0,1,0)$
9: $x \leftarrow (\mu_1 \odot \overline{n}) \odot m' + \mathrm{Rot}_{-1}((\mu_2 \odot \overline{n}) \odot m')$
10: **return** x.

The following Theorem makes a precise list of operations performed within the quasi-inverse computation, in order to prepare for the packing description of Section 4.7.1 and optimization of Section 4.9.2.

Theorem 4.2. *Let $B=[a,b] \in \mathrm{GF}(256) - \{0\}$, and define the quasi-inverse of B to be $B' = (B^{-1})^c$, i.e. $B' = [a',b']$ such that $(a'+b'+b'U)\cdot B = 1$.*

Algorithm 3, on input $\boldsymbol{\alpha}(B)$, returns $v' \in \mathbb{C}^4$ such that $v' \equiv \boldsymbol{\alpha}(B')$. It uses three multiplicative levels and performs

- *one* GF(2)*-linear transform;*
- *four ciphertext-ciphertext multiplications.*
- *four rotations by* $\pm 1, 2$ *under* $\boldsymbol{\alpha}$ *encoding.*

Proof. We let $\boldsymbol{\alpha}([a,b]) = (a_1, b_1, a_2, b_2)$. Then[6] $n_1 = (a_1 \cdot b_1, *, a_2 \cdot b_2, *)$ and $n = (\nu_1, *, \nu_2, *)$, where $T_5(\nu) \equiv (\nu_1, \nu_2)$.

We recall that conjugation in T_5 encoding corresponds to evaluation of the fourth-power, while for a real message, swapping the coordinates of T_5 encoding corresponds to squaring. Hence, $\overline{n}$ contains $T_5(\nu^4) = (\overline{\nu_1}, \overline{\nu_2})$ at coordinates 0 and 2, while m contains $T_5(\nu^{10}) = (\overline{\nu_2} \cdot \nu_2, \overline{\nu_1} \cdot \nu_1)$ at coordinates 0 and 2.

We deduce that $m \cdot \overline{n}$ contains $T_5(\nu^{-1})$ in coordinates 0 and 2; the masking process (steps 7-9) multiplies by $\boldsymbol{\alpha}(B)$, minimizing the level consumption while propagating those values to coordinates 1 and 3, yielding $x \equiv \boldsymbol{\alpha}(B \cdot \nu^{14}) = \boldsymbol{\alpha}(B')$, as claimed.

Steps 2, 5, 9 exhibit the 4 ciphertext-ciphertext multiplications. □

Turning a quasi-inverse into a true inverse corresponds to the conjugation map $B \mapsto B^c$, which is a linear map. Evaluating it on its own would require one further level, while we can fuse it with other linear transforms corresponding to the steps of AES evaluation following the SubBytes step.

[6] We let * denote arbitrary values.

4.7 Fusing all linear layers

4.7.1 Efficient packing for SIMD operations. We now explain the SIMD packing used in our method introduced for the efficient homomorphic evaluation. One AES block, or state, consists of 16 bytes, each byte being mapped to $\mathbb{C}^4$ via the $\boldsymbol{\alpha}$ map. A CKKS message, which has $N/2$ complex slots, can thus accommodate $N/128$ AES blocks. We let $\mathcal{B}_i = (B_0^{(i)}, \ldots, B_{15}^{(i)}) \in \mathrm{GF}(256)^{16}$ be those blocks, for $0 \le i < N/128$.

Given a map μ on GF(256), we extend it (by abuse, we keep the same notation μ) to $\mathrm{GF}(256)^{16}$ as

$$\mu(\mathcal{B}_i) = (\mu(B_j^{(i)}))_{0 \le j < 16}.$$

We build a CKKS packing by embedding our blocks in CKKS message space as

$$\begin{aligned}&(\alpha_{00}(\mathcal{B}_0)\,||\alpha_{00}(\mathcal{B}_1)||\ldots||\alpha_{00}(\mathcal{B}_{N/128-1})||\\ &\alpha_{10}(\mathcal{B}_0)||\alpha_{10}(\mathcal{B}_1)||\ldots||\alpha_{10}(\mathcal{B}_{N/128-1})||\\ &\alpha_{01}(\mathcal{B}_0)||\alpha_{01}(\mathcal{B}_1)||\ldots||\alpha_{01}(\mathcal{B}_{N/128-1})||\\ &\alpha_{11}(\mathcal{B}_0)\,||\alpha_{11}(\mathcal{B}_1)||\ldots||\alpha_{11}(\mathcal{B}_{N/128-1})) \in \mathbb{C}^{N/2}. \end{aligned} \tag{14}$$

This layout has three properties on which the rest of the section relies.

- The two GF(16) coordinates a, b of a byte sit $N/8$ positions apart, so exchanging them, or copying one coordinate onto the other, is a rotation by $N/8$.
- The two slots $\alpha_{e0}(B), \alpha_{e1}(B)$ corresponding to evaluating T_5 on a single GF(16) coordinate of a byte ($e \in \{0,1\}$) lie at distance $N/4$ (Section 4.4);
- A GF(2)-linear map acting independently on each state, with single-state matrix $\mathbf{M}_L$, acts on the packed message as $\mathbf{I}_{N/128} \otimes \mathbf{M}_L$: this matrix has the same number of diagonals as $\mathbf{M}_L$, so that the cost of the linear map is the same independently of the number of blocks.

The second point, in particular, shows that when seeing the input as an element of $\mathrm{GF}(16)^{N/4}$, we recover the packing (9), meaning that Sections 4.4.2 and 4.5 remain fully valid in this setting.

4.7.2 Fused linear layer. Let $(B_0, \ldots, B_{15}) \in \mathrm{GF}(256)^{16}$ be a state. After the inversion procedure of Section 4.6.2 we obtain the quasi-inverses $\mathcal{B} = (B_0^{-1^c}, \ldots, B_{15}^{-1^c})$, and completing the round amounts to evaluating

$$\mathsf{MC} \cdot \mathsf{SR} \cdot (\mathsf{SB}_L \cdot \mathsf{Conj} \oplus \mathsf{0x63}) \oplus K_r, \tag{15}$$

on the current AES state, where SB_L is the linear part of the SB affine map, $\mathsf{Conj} : \mathcal{B} \mapsto \mathcal{B}^c$ is the conjugation map, and K_r is the round key. We note that MC should be skipped during the final round.

Equation (15) corresponds to the evaluation of an affine map; the constant part of $(\mathsf{MC} \cdot \mathsf{SR} \cdot \mathsf{0x63} \oplus K_r)$ is precomputed and added at the end of evaluation of the linear part.

The remaining steps are linear operations: the conjugation $\mathcal{B} \mapsto \mathcal{B}^c$, the linear part of SubBytes SB_L, and SR and MC. We explore in Appendix I their fusion into a single map D, also integrating the next linear step: the $T_5^{N/4^{-1}}$ map that starts bootstrapping, see Section 4.5. We summarize the analysis of this structure of the resulting linear map in the following Theorem:

Theorem 4.3. *Let $z \in \mathbb{C}^{N/2}$ be a CKKS message encoding $\mathbf{m} \in \mathrm{GF}(16)^{N/4}$ corresponding to $N/128$ AES blocks packed as described in (14) and define the map*

$$\mathbf{m} \mapsto T_5^{N/4^{-1}} \cdot \widehat{\mathsf{MC}} \cdot \widehat{\mathsf{SR}} \cdot \widehat{\mathsf{SB}_L} \cdot \widehat{\mathsf{Conj}},$$

where $\widehat{\mathsf{MC}}, \widehat{\mathsf{SR}}, \widehat{\mathsf{SB}_L}, \widehat{\mathsf{Conj}}$ extend the maps $\mathsf{MC}, \mathsf{SR}, \mathsf{SB}_L, \mathsf{Conj}$ to $N/128$ AES blocks.

There exist matrices $\mathbf{A}_k^{(\ell)}, \mathbf{B}_k^{(\ell)} \in \mathbb{C}^{4\times4}$ and matrices $\mathbf{A}_k^{(r)}, \mathbf{B}_k^{(r)} \in \mathbb{C}^{16\times16}$ such that this map can be evaluated on the vector $\mathbf{z}$ as

$$\mathbf{z} \mapsto \sum_{k=0}^{3} \left(\mathbf{I}_{N/128} \otimes \mathbf{A}_k^{(\ell)} \otimes \mathbf{A}_k^{(r)} \cdot \mathbf{z} + \mathbf{I}_{N/128} \otimes \mathbf{B}_k^{(\ell)} \otimes \mathbf{B}_k^{(r)} \cdot \overline{\mathbf{z}}\right).$$

Appendix I explicitly describes the derivation of the corresponding matrices, but the structure is actually natural:

- The leftmost component, $\mathbf{I}_{N/128}$ corresponds to the fact that the linear maps act independently and simultaneously on the $N/128$ blocks;
- The middle component is related to the action within a single byte: conjugation, the linear part of SubBytes, part of MixColumns and T_5^{-1};
- The rightmost component corresponds to a linear (mostly permutation) action on the various 16 bytes of a state; it includes ShiftRows and the permutation part of MixColumns;
- The existence of 4 components is related to the fact that the MixColumns operator has no natural tensor decomposition between a part that acts on bytes and a part that independently acts on groups of 16 bytes; in order for this tensor decomposition to appear, we decompose MixColumns into a sum of 4 linear maps which can be decomposed as a tensor product.

Finally, the fact that the decomposition acts on both $\mathbf{z}$ and $\overline{\mathbf{z}}$ is in line with Section 4.4.2.

As in Section 3.4.2, we now consider fusion with the first StC layer, which we choose to have the form $\mathbf{I}_{N/32} \otimes \mathbf{DFT}_{16}$. Overall, the fused linear layers can be evaluated as:

$$\begin{aligned} z \mapsto \sum_{k=0}^{3} \Big(&\mathbf{I}_{N/128} \otimes \mathbf{A}_k^{(\ell)} \otimes (\mathbf{DFT}_{16} \cdot \mathbf{A}_k^{(r)}) \cdot \mathbf{z} \\ &+ \overline{\mathbf{I}_{N/128} \otimes \mathbf{B}_k^{(\ell)} \otimes (\mathbf{DFT}_{16} \cdot \mathbf{B}_k^{(r)}) \cdot \mathbf{z}}\Big). \end{aligned} \tag{16}$$

As we evaluate 8 matrix-vector products on the same input (and assuming equal cost for baby-steps and giant-steps), the number of baby-steps should be taken to be roughly eight times larger as the number of giant steps. We note that the set of non-zero diagonals of all the 8 matrices is included into the set $\{0, N/8, N/4, 3N/8\} + \{-15, -14, \ldots, 14, 15\}$. It suggests the natural choice that each of these matrix-vector products can thus be evaluated using the same baby-steps $\{-15, -14, \ldots, 14, 15\}$, and the giant steps $\{0, N/8, N/4, 3N/8\}$, for a total of $31 + 8 \cdot 3 + 1 = 56$ key-switchings.

4.8 Controlling error increase in CKKS slots

One of the challenges when implementing ideas similar to [23] is handling error increase. The latter comes from two main sources:

- Let $\mathrm{pt} + e$ be the result of a CKKS operation, where pt is a CKKS plaintext. The error in CKKS message space then

Table 1: Empirical embedding growth for binary lifts. The second column reports the maximum absolute value of a CKKS slot over all encodings of elements of GF(16) (or GF(256) for the last polynomial).

Polynomial	d	max. slot value	after 4 mults (bits)
Φ_5	4	1.618	11.1
$X^4 + X + 1$	4	3.170	26.6
$X^4 + X^3 + 1$	4	2.459	20.8
$X^8 + X^4 + X^3 + X + 1$	8	8.344	49.0

becomes $\mathbf{DFT}(e)$; in $(\mathbb{Z}[X]/\Phi_5(X))^{N/4}$, this error becomes $(T_5^{N/4})^{-1}(\mathbf{DFT}(e))$: each CKKS computation incurs an error increase related to the norm of the inverse map T_5^{-1};

- embedding GF(2) into the integers leads to a significant increase in the value of CKKS slots. When multiplying two noisy messages $\mathbf{z}_1+\mathbf{e}_1$ and $\mathbf{z}_2+\mathbf{e}_2$, the error increase in the result is related to a bound on the true value contained in CKKS slots, namely $\max \|\mathbf{z}_i\|_\infty$. Lazy reduction modulo 2 means that every addition may lead to a one bit increase in size, and every ciphertext-ciphertext multiplication may double the size of each CKKS slot, with large impact on the size of the error at the next step. When performing several ciphertext-ciphertext multiplications in a row as in Algorithm 3, there is a risk of error explosion, requiring very large scaling factors to retain precision and leading to inefficiency.

In comparison, plaintext-ciphertext multiplications lead to a more modest increase of error; the latter should thus be preferred over the former, especially when the corresponding ciphertext has already gone through several operations and may have large magnitude. This explains, in particular, the choice to implement m^2 as a linear map rather than a ciphertext-ciphertext multiplication in Algorithm 3.

4.8.1 Choosing the optimal embedding. We already partly addressed the first issue by choosing the fifth cyclotomic polynomial to build GF(16). We compare this choice to other choices, and to direct implementation of GF(256) using the AES polynomial, in Table 1.

In particular, in the worst case, when squaring the largest element 4 times consecutively, we obtain a total growth of the order of 11.1 bits, against 26.6, 20.8 and 49.0 bits for the other three polynomials, leading to much stable computations by using Φ_5.

We note that all polynomials with $\{0,1\}$ coefficients are mapped through the canonical embedding to vectors in $\mathbb{C}^2$ with supremum norm $\in \{0, 1, \frac{1+\sqrt{5}}{2}\}$; this is optimal, so the naive lifting for GF(2) to $\{0,1\}$ maps elements of GF(16) to elements of $\mathbb{C}^2$ in an optimal way.

4.8.2 Lifting matrices. When evaluating $\mathbf{M}_L \cdot z$ for a given matrix $\mathbf{M}_L \in \mathrm{GF}(2)^{4\times 4}$, several different liftings $\widetilde{\mathbf{M}}_L^{4\times 4}$ can be selected. This choice should be made carefully, as it impacts error increase in CKKS. Indeed, what is really evaluated in CKKS slots is the quantity

$$T_5(\widetilde{\mathbf{M}}_L \cdot u),$$

where u is a lifting of an element of GF(16). This quantity should be kept small to minimize the error. For a given matrix $\mathbf{M}_L$, we thus solve the optimization problem select $\widetilde{\mathbf{M}}_L \in \{-1,0,1\}^{4\times 4}$ such that $\widetilde{\mathbf{M}}_L \equiv \mathbf{M}_L \bmod 2$ and $\widetilde{\mathbf{M}_L}$ minimizes

$$\min_{\substack{\widetilde{\mathbf{M}}_L \in \{-1,0,1\}^{4\times 4} \\ \widetilde{\mathbf{M}}_L \equiv \mathbf{M}_L \bmod 2}} \max_{u \in \mathrm{GF}(16)} \|T_5(\widetilde{\mathbf{M}}_L(u))\|_\infty.$$

Table 2: (Unscaled) slot values for the underlying message, after the fused linear layer.

Configuration	max $\lvert m\rvert$	improvement
Unsigned lifts, arbitrary (β, Γ)	1.47×10^8	1.0×
Unsigned lifts, optimized (β, Γ)	2.13×10^7	6.9×
Signed lifts, optimized (β, Γ)	1.40×10^6	105×

4.8.3 Choosing β and Γ. Finally, we stress that the choice of β and Γ is not canonical: a number of different choices are possible and lead to different magnitudes for the slot values under the T_5 encoding; this impacts, in particular, the fused linear layer which depends on the algebraic representation of the field GF(256), see Appendix I.

4.8.4 Global optimization. The previous sections outline how to optimize each individual choice. In order to maximize efficiency, we need to target the minimum possible computational precision, meaning that global optimization of all the parameters is desirable. We used the following heuristic strategy:

- We first optimize each linear step separately using the strategy from Section 4.8.2;
- We then optimize β, Γ under this choice for the linear steps;
- Given this choice of β and Γ, we reoptimize all the linear steps together.

The rationale underlying this strategy is that β, Γ primarily impact the fused linear layer; as the latter occurs at minimum level just before reduction modulo 2, it already handles the largest values in the circuit and specific care must be taken in order to avoid accuracy loss.

The objective function that we minimize for a lifting $\tilde{\phi}$ of a (sequence of) step ϕ is

$$\min_{\tilde{\phi}} \max_{u \in \mathrm{GF}(16)} \|\widetilde{\phi}(T_5(u))\|_\infty.$$

Table 2 reports on the impact of each step. Overall, we save 7 bits over the maximum slot size after the fused linear layer. This allows a reduction of 7 bits of all the scaling factors during the AES circuit, and a less precise bootstrapping, overall reducing depth significantly. In particular, without these numerical optimizations, running the AES circuit with small FHE parameters ($\log N = 15$, see Table 3) is not possible.

4.9 Optimizations

4.9.1 Batching Ciphertexts for throughput. Finally, when targeting maximum throughput, our algorithm benefits from the use of the shared-a batching technique described in [5]; this technique allows to speedup both the fused linear parts (SR + MC + StC_0 transformation) and the CtS step of bootstrapping. The inversion itself remains evaluated in ordinary RLWE format. We used a batch size of 4 ciphertexts, and mention this variant in our tables as $\mathrm{GF}(16)_{\mathrm{MS}}$.

4.9.2 Rotation-invariant secret key. When performing rotations of index $N/4$ or $\pm N/8$, we make use of the rotation-invariant secret-key technique. By switching the ciphertext (a, b) decrypting to a plaintext pt to a ciphertext (a', b') for a key $s'(X^4) \in \mathcal{R}_{N/4}$, we obtain a ciphertext for which rotations by $N/4$ and $N/8$ positions do not require key-switching (see Lemma A.1)

To evaluate such a homomorphic rotation with a key $s'(X^4)$ one thus merely applies the Frobenius map to a' and b'. In terms of security, this approach is equivalent to working with ciphertexts in $\mathcal{R}_{N/4,Q}$, meaning that it can be used when the modulus is small enough.

This impacts the following steps of AES evaluation:

- The rotations of index $\pm 1, 2$ in Theorem 4.2 correspond, under the global packing, to rotations of $N/4$ or $N/8$, which no longer require key-switching: the cost of quasi-inverse is reduced to 4 ct-ct multiplications and one GF(2) linear map;
- When evaluating Equation (16), it eliminates the cost of giant steps, reducing total cost from 56 key-switchings down to 32.

5 Implementation and experiments

We implemented all AES evaluation circuits—including our low-latency and GF(16) variants as well as reproductions of [37] and [6]—using the HEaaN2 [20] library, which provides both algorithmic and GPU-level optimizations for CKKS homomorphic operations. We compared those with our algorithm on an Intel Xeon Gold processor with an NVIDIA RTX 5090 GPU. In order to provide a fair comparison with [41] on GPU, we also ran our implementation on an Intel Xeon Gold 6542Y with an NVIDIA RTX 4090 GPU, the same configuration used in [41]. All reported timings are averaged over 100 runs; precisions report the worst case ($\max(-\log \epsilon)$).

5.1 Parameters

All implementations in our experiments, including the reproduced baselines utilize the *grafting* technique [14], which provides two complementary benefits: (i) it enables sufficiently small scale factors (around 20 bits) without sacrificing precision, and (ii) it reduces the number of RNS primes by employing word-size (~60-bit) special primes, thereby decreasing the modulus and accelerating homomorphic operations. Since grafting benefits all methods uniformly, we apply it to the parameter sets of [37] and [6] as well, yielding *optimized* parameter sets (denoted [37]*, and [6]*) that achieve smaller $\log Q_{\max}$ while maintaining the same precision.

Table 3 summarizes the parameters for each configuration. For more details on the modulus chain of each parameter, refer to Table 7 in Appendix G.1 . All parameter sets achieve 128-bit security according to the lattice estimator [2] using commit `8d38f52`.

5.2 Experimental Results

5.2.1 Low-Latency. Table 4 shows the performance breakdown of our latency algorithm described in Section 3. We achieve **26.4** ms latency for evaluating AES-128 over 1 block and **29.2**ms latency for evaluation over 4 blocks.

Table 3: FHE parameter sets for AES-CKKS transciphering. We use secret key with Hamming weight 256 for $\mathsf{Latency}_{1,4}$ and 1024 for GF(16) and temporary sparse secret [11] with Hamming weight 32 for both parameter sets. The last two row include temporary parameters used in SB.

	$\log N$	$\log(PQ)$	$dnum$
$\mathsf{Latency}_1$	15	770	2
$\mathsf{Latency}_4$	15	790	2
$\mathrm{GF}(16)_{15}$	15 (14)	854 (426)	2 (2)
GF(16)	16 (14)	1445 (426)	1 (2)

Accuracy. $\mathsf{Latency}_1$ and $\mathsf{Latency}_4$ achieve $\max(-\log \epsilon)$ of 9.28 and 8.77, respectively, both exceeding the threshold required for correct rounding to {0,1}.

Table 4: Per-step breakdown of our circuits on RTX 5090. Times in ms (ratio of total in parentheses). $\mathsf{Latency}_1$ (resp. $\mathsf{Latency}_4$, $\mathrm{GF}(16)_{15}$) handle 1 (resp. 4, 256) block encrypted into one $\log N = 15$ ciphertext; GF(16) handles 512 blocks encrypted into one $\log N = 16$ ciphertext. $\mathrm{GF}(16)_{\mathrm{MS}}$ handles 2048 blocks encrypted into four $\log N = 16$ ciphertexts, using shared-a for all linear layers.

Variant	ProdTree / SB	FusedStC	ARK	HalfBTS	Total
$\mathsf{Latency}_1$	6.4 (24.2%)	1.3 (4.9%)	0.2 (0.7%)	18.5 (70.1%)	26.4
$\mathsf{Latency}_4$	6.8 (23.1%)	3.3 (11.3%)	0.2 (0.7%)	19.0 (65.0%)	29.2
$\mathrm{GF}(16)_{15}$	6.94 (17.5%)	4.26 (10.8%)	0.36 (0.9%)	28.03 (70.8%)	39.6
GF(16)	6.69 (13.2%)	6.02 (11.9%)	0.31 (0.6%)	37.77 (74.4%)	50.8
$\mathrm{GF}(16)_{\mathrm{MS}}$	15.38 (11.2%)	12.54 (9.1%)	0.88 (0.6%)	108.47 (79.0%)	137.3

5.2.2 GF(16). Table 5 compares our GF(16) variants against [37][7] and [6]. For a fair comparison, all methods use grafting-optimized parameters (Section 5.1).

Throughput. Our $\mathrm{GF}(16)_{\mathrm{MS}}$ circuit achieves a throughput of **238.7KB/s**, a 3.41× improvement over [6]* (70.1KB/s), a 5.54× improvement over [37]*-Batch (43.1KB/s), and a 6.84× improvement over [37]* (34.9KB/s). Notably, this is achieved with only 2048 packed blocks batched across four ciphertexts, compared to 16384 for [37]* and 32768 for [6]*, yielding a significantly better latency and wider applicability.

Latency. Our $\mathrm{GF}(16)_{15}$ circuit processes 256 AES blocks in **39.6 ms**, roughly 152×–188× faster than the baselines in absolute latency.

[7] [37] introduces two variants, which we denote by [37] and [37]-Batch. The latter utilizes more slots per bootstrapping, thereby reducing the number of bootstrappings thus the total latency.

Accuracy. GF(16) and $\mathrm{GF}(16)_{15}$ achieve $\max(-\log \epsilon)$ of 9.84 and 9.26, which are well above the margin required for correct rounding to $\{0, 1\}$.

5.3 Comparison with other AES-FHE Evaluations

We compare the latency and throughput with previous work on homomorphic AES evaluation under CKKS and TFHE.

In order to provide a fair comparison, we have reimplemented all the CKKS-based approaches, which are thus run in the same setting as ours. In contrast, as Maverick [41] does not provide publicly available code, we include its results by referring to the figures reported in the original paper. The latter also introduces a GPU-accelerated implementation of the TFHE scheme and reports AES transciphering benchmarks on various GPU platforms. In Table 5, we compare our results with those reported in [41]. As a fair comparison basis, we also ran our code on RTX 4090.

In this context, our latency-oriented approaches achieve approximately 7× lower latency. Furthermore, our GF(16) approach also attains lower latency while achieving more than two orders of magnitude higher throughput. We also include CPU comparison in Appendix G.2.

6 Applications

We outline and evaluate two applications of homomorphic AES evaluation under threshold FHE: threshold AES decryption and AES-based key derivation function (KDF). We place ourselves in the semi-honest model.

6.1 Threshold-AES decryption

Threshold-AES decryption is based on the universal thresholdizer property of FHE [10]. Given AES ciphertexts, a threshold-FHE setup and an FHE-encrypted AES key, threshold-AES decryption collaboratively decrypts without revealing the AES key in plaintext. After key expansion from the encrypted AES key, the server evaluates the AES decryption circuit homomorphically, and the final output is recovered through collaborative threshold decryption.

Benchmark. We measure the latency of threshold-AES decryption, including (1) homomorphic key expansion from the encrypted AES key, (2) AES circuit evaluation, and (3) threshold decryption with communication cost.

Although homomorphic key expansion is a one-time setup cost (typically considered offline), we include it to present the complete end-to-end threshold-AES decryption pipeline.

For threshold decryption [3, 10], we account for local partial decryption, final decryption, and communication overhead of transmitting decryption shares. To enable secure noise flooding, sufficient precision is required before partial decryption. We achieve this precision via a several homomorphic bit cleaning steps introduced in [19], without introducing a separate high-precision circuit.

The communication costs are estimated from per-party decryption share sizes for $N = 5$ parties using standard RTT and bandwidth models; detailed assumptions are given in Table 9 of Appendix J.1. The polynomial modulus sizes before partial decryption are 2 and 4 limbs of 64 bits for $\mathsf{Latency}_1$ and GF(16), respectively, and reduce to a single 64-bit limb after noise flooding.

Under the intra-region setting, e2e-transciphering of a single AES block utilizing our $\mathsf{Latency}_1$ algorithm reports 88.4 ms latency. For 512 blocks, GF(16) reports 138.6 ms latency. More detailed results, including cross country and global deployment settings, are provided in Table 6.

6.2 Homomorphic AES-CMAC-based KDF

A Key Derivation Function (KDF) derives session or subordinate keys from a master secret. We implement the PRF-based KDF construction standardized in NIST SP 800-108 [13] using AES-CMAC, which is particularly suitable for AES-centric systems. Given an FHE-encrypted master key and public derivation inputs, the server homomorphically evaluates the full AES-CMAC computation in counter mode. This approach, performing key expansion, subkey generation, and AES evaluation, enables secure key derivation under threshold FHE without ever exposing the master key in plaintext. Detailed circuit descriptions are provided in Appendix K.

Benchmark. The evaluation consists of a single-time subkey generation step followed by iterative CMAC evaluations for key derivation. Subkey generation requires one AES evaluation to compute $(L = E_K(0^{128}))$, followed by two homomorphic linear transforms to derive (K_1) and (K_2), and is performed once per master key.

We achieved 112.2 ms and 123.3 ms for 128-bit and 512-bit derived keys, respectively (full results are described in Table 10 of Appendix K), demonstrating that low-latency key derivation remains practical for session establishment under threshold FHE.

Acknowledgments

We thank Damien Stehlé for useful discussions.

References

[1] Ehud Aharoni, Nir Drucker, Gilad Ezov, Eyal Kushnir, Hayim Shaul, and Omri Soceanu. 2023. E2E near-standard and practical authenticated transciphering. Cryptology ePrint Archive, Paper 2023/1040.

[2] Martin Albrecht, Rachel Player, and Sam Scott. 2015. On the concrete hardness of Learning with Errors. *Journal of Mathematical Cryptology* 9 (10 2015). doi:10.1515/jmc-2015-0016

[3] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. 2012. Multiparty Computation with Low Communication, Computation and Interaction via Threshold FHE. In *Advances in Cryptology – EUROCRYPT 2012*, David Pointcheval and Thomas Johansson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 483–501.

[4] Youngjin Bae, Jung Hee Cheon, Wonhee Cho, Jaehyung Kim, and Taekyung Kim. 2022. META-BTS: Bootstrapping Precision Beyond the Limit. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 223–234. doi:10.1145/3548606.3560696

[5] Youngjin Bae, Jung Hee Cheon, Guillaume Hanrot, Jai Hyun Park, and Damien Stehlé. 2024. Plaintext-Ciphertext Matrix Multiplication and FHE Bootstrapping: Fast and Fused. In *Crypto*.

[6] Youngjin Bae, Jung Hee Cheon, Minsik Kang, and Taeseong Kim. 2025. High-Throughput AES Transciphering using CKKS: Less than 1ms. In *Proceedings of the 13th Workshop on Encrypted Computing & Applied Homomorphic Computing* (Taipei, Taiwan) *(WAHC '25)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3733811.3767314

[7] Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, and Damien Stehlé. 2024. Bootstrapping Bits with CKKS. In *Advances in Cryptology – EUROCRYPT 2024*, Marc Joye and Gregor Leander (Eds.). Springer Nature Switzerland, Cham, 94–123.

[8] The timings for AES decryption are larger than in Table 5 because the last bootstrapping targets higher modulus required for cleaning.

Table 5: Latency (in seconds) and throughput (in KB/s) comparison for AES-128 evaluation on NVIDIA GPUs. Latency is averaged over 100 runs. Throughput is measured in KB/s. Param. denotes the parameters used in the experiments, and #Blocks represents the number of AES blocks.

Scheme	Method (param.)	Machine	#Blocks	Latency (s)	Throughput (KB/s)	Library
TFHE†	[41]-(CBS-I)	RTX 4090	8	0.29	0.434	Maverick
	[41]-(CBS-I)	H100	8	0.17	0.754	
CKKS	Ours (Latency_1)	RTX 4090	1	0.031	0.52	HEaaN2
	Ours (Latency_4)		4	0.036	1.78	
	Ours ($\text{GF}(16)_{15}$)		256	0.052	78.8	
	Ours ($\text{GF}(16)$)		512	0.074	110.7	
	Ours ($\text{GF}(16)_{\text{MS}}$)		2048	0.239	137.1	
CKKS	XBOOT* [37]	RTX 5090	16384	7.52	34.9	HEaaN2
	XBOOT* [37]-Batch		16384	6.08	43.1	
	WAHC* [6]		32768	7.48	70.1	
	Ours (Latency_1)		1	**0.026**	0.61	
	Ours (Latency_4)		4	0.029	2.19	
	Ours ($\text{GF}(16)_{15}$)		256	0.040	103.5	
	Ours ($\text{GF}(16)$)		512	0.051	161.3	
	Ours ($\text{GF}(16)_{\text{MS}}$)		2048	0.137	**238.7**	

† Results are directly taken from the original paper.

Table 6: End-to-end latency breakdown of threshold-AES decryption under threshold FHE ($N = 5$ parties). Server-side computation is measured on an RTX 5090 GPU. All latency values are in ms unless otherwise noted.

Component	Latency_1	$\text{GF}(16)$
Computation (Server), ms		
Key expansion	49.7	75.0
AES decryption	33.6[8]	54.6[8]
Partial decryption	3.5	6.3
Final decryption	0.02	0.02
Communication (Network) per Party		
Comm. Size (MB)	0.75	2.1
Comm. Latency (Intra-region, ms)	1.6–2.6	2.7–3.7
Comm. Latency (Cross country, ms)	66–76	77–87
Comm. Latency (Global, ms)	150–210	258–318
Total E2E (Intra-region)	88.4–89.4	138.6–139.6
Total E2E (Cross country)	152.8–162.8	212.9–222.9
Total E2E (Global)	236.8–296.8	393.9–453.9

[8] Youngjin Bae, Jaehyung Kim, Damien Stehlé, and Elias Suvanto. 2025. Bootstrapping Small Integers With CKKS. In *Advances in Cryptology – ASIACRYPT 2024*, Kai-Min Chung and Yu Sasaki (Eds.). Springer Nature Singapore, Singapore, 330–360.

[9] Sonia Belaïd, Nicolas Bon, Aymen Boudguiga, Renaud Sirdey, Daphné Trama, and Nicolas Ye. 2025. Further Improvements in AES Execution over TFHE. *IACR Communications in Cryptology* 2, 1 (2025), 1–24. doi:10.62056/ahmp-4tw9

[10] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. 2018. Threshold Cryptosystems from Threshold Fully Homomorphic Encryption. In *Advances in Cryptology – CRYPTO 2018*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer International Publishing, Cham, 565–596.

[11] Jean-Philippe Bossuat, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2022. Bootstrapping for Approximate Homomorphic Encryption with Negligible Failure-Probability by Using Sparse-Secret Encapsulation. In *Applied Cryptography and Network Security: 20th International Conference, ACNS 2022, Rome, Italy, June 20–23, 2022, Proceedings* (Rome, Italy). Springer-Verlag, Berlin, Heidelberg, 521–541. doi:10.1007/978-3-031-09234-3_26

[12] Hao Chen, Ilaria Chillotti, and Yongsoo Song. 2019. Improved Bootstrapping for Approximate Homomorphic Encryption. In *Advances in Cryptology – EUROCRYPT 2019*, Yuval Ishai and Vincent Rijmen (Eds.). Springer International Publishing, Cham, 34–54.

[13] Lily Chen. 2022. *Recommendation for Key Derivation Using Pseudorandom Functions.* NIST Special Publication 800-108 Rev. 1. National Institute of Standards and Technology. doi:10.6028/NIST.SP.800-108r1-upd1 Updated February 2024.

[14] Jung Hee Cheon, Hyeongmin Choe, Minsik Kang, Jaehyung Kim, Seonghak Kim, Johannes Mono, and Taeyeong Noh. 2025. Grafting: Decoupled Scale Factors and Modulus in RNS-CKKS. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (Taipei, Taiwan) *(CCS '25)*. Association for Computing Machinery, New York, NY, USA, 1098–1112. doi:10.1145/3719027.3765083

[15] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. Bootstrapping for Approximate Homomorphic Encryption. In *Advances in Cryptology – EUROCRYPT 2018*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer International Publishing, Cham, 360–384.

[16] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer International Publishing, Cham, 409–437.

[17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2017. Faster Packed Homomorphic Operations and Efficient Circuit Bootstrapping for TFHE. In *Advances in Cryptology – ASIACRYPT 2017*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer International Publishing, Cham, 377–408.

[18] Jihoon Cho, Jincheol Ha, Seongkwang Kim, Byeonghak Lee, Joohee Lee, Jooyoung Lee, Dukjae Moon, and Hyojin Yoon. 2021. Transciphering Framework for Approximate Homomorphic Encryption. In *Advances in Cryptology – ASIACRYPT 2021*, Mehdi Tibouchi and Huaxiong Wang (Eds.). Springer International Publishing, Cham, 640–669.

[19] Hyeongmin Choe, Jaehyung Kim, Damien Stehlé, and Elias Suvanto. 2025. Leveraging Discrete CKKS to Bootstrap in High Precision. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (Taipei, Taiwan) *(CCS '25)*. Association for Computing Machinery, New York, NY, USA, 1083–1097. doi:10.1145/3719027.3765080

[20] HEaaN2. CryptoLab. 2026. HEaaN2 library. https://www.cryptolab.co.kr/en/products-en/heaan-he/.

[21] Christoph Dobraunig, Lorenzo Grassi, Lukas Helminger, Christian Rechberger, Markus Schofnegger, and Roman Walch. 2023. Pasta: A Case for Hybrid Homomorphic Encryption. *IACR Transactions on Cryptographic Hardware and*

Embedded Systems 2023, 3 (June 2023), 30–73. doi:10.46586/tches.v2023.i3.30-73 Artifact available at https://artifacts.iacr.org/tches/2023/a13.

[22] Nir Drucker, Guy Moshkowich, Tomer Pelleg, and Hayim Shaul. 2023. BLEACH: Cleaning Errors in Discrete Computations Over CKKS. *J. Cryptol.* 37, 1 (Nov. 2023), 35 pages. doi:10.1007/s00145-023-09483-1

[23] Mingyu Gao and Hongren Zheng. 2026. FHE for SIMD Arithmetic Logic Units with Amortized $O(1)$ Bootstrapping per Ciphertext. Cryptology ePrint Archive, Paper 2026/233. https://eprint.iacr.org/2026/233

[24] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Homomorphic Evaluation of the AES Circuit. In *Advances in Cryptology – CRYPTO 2012*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 850–867.

[25] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Homomorphic Evaluation of the AES Circuit. Cryptology ePrint Archive, Paper 2012/099. https://eprint.iacr.org/2012/099

[26] Lorenzo Grassi, Irati Manterola Ayala, Martha Norberg Hovd, Morten Øygarden, Håvard Raddum, and Qingju Wang. 2023. Cryptanalysis of Symmetric Primitives over Rings and a Key Recovery Attack on Rubato. In *Crypto*.

[27] Jincheol Ha, Seongkwang Kim, Byeonghak Lee, Jooyoung Lee, and Mincheol Son. 2022. Rubato: Noisy Ciphers for Approximate Homomorphic Encryption. In *Advances in Cryptology – EUROCRYPT 2022*, Orr Dunkelman and Stefan Dziembowski (Eds.). Springer International Publishing, Cham, 581–610.

[28] Shai Halevi and Victor Shoup. 2014. Algorithms in HElib. In *Advances in Cryptology – CRYPTO 2014*, Juan A. Garay and Rosario Gennaro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 554–571.

[29] Shai Halevi and Victor Shoup. 2018. Faster Homomorphic Linear Transformations in HElib. In *Advances in Cryptology – CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I* (Santa Barbara, CA, USA). Springer-Verlag, Berlin, Heidelberg, 93–120. doi:10.1007/978-3-319-96884-1_4

[30] Kyoohyung Han, Minki Hhan, and Jung Hee Cheon. 2019. Improved Homomorphic Discrete Fourier Transforms and FHE Bootstrapping. *IEEE Access* 7 (2019), 57361–57370. doi:10.1109/ACCESS.2019.2913850

[31] Lin Jiao, Yongqiang Li, Yonglin Hao, and Xinxin Gong. 2024. Differential Fault Attacks on Privacy Protocols Friendly Symmetric-Key Primitives: RAIN and HERA. *IET Inf. Secur.* (2024).

[32] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: a low latency framework for secure neural network inference. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 1651–1668.

[33] Duhyeong Kim and Yongsoo Song. 2019. Approximate Homomorphic Encryption over the Conjugate-Invariant Ring. In *Information Security and Cryptology – ICISC 2018*, Kwangsu Lee (Ed.). Springer International Publishing, Cham, 85–102.

[34] Jaehyung Kim and Taeyeong Noh. 2025. Modular Reduction in CKKS. *IACR Communications in Cryptology* 2, 2 (2025). doi:10.62056/aevur-iuc

[35] Fukang Liu, Abul Kalam, Santanu Sarkar, and Willi Meier. 2024. Algebraic Attack on FHE-Friendly Cipher HERA Using Multiple Collisions. *IACR Trans. Symmetric Cryptol.* (2024).

[36] Fukang Liu, Tianning Wang, Jianqiang Ni, Daisuke Yamamoto, Guilherme T. S. Sato, Haoyang Wang, and Willi Meier. 2025. Chosen-IV Algebraic Attack on Randomized Ciphers FASTA and HERA. *IACR Trans. Symmetric Cryptol.* (2025).

[37] Chao Niu, Zhicong Huang, Zhaomin Yang, Yi Chen, Liang Kong, Cheng Hong, and Tao Wei. 2025. XBOOT: Free-XOR Gates for CKKS with Applications to Transciphering. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2025, 4 (Sep. 2025), 118–144. doi:10.46586/tches.v2025.i4.118-144

[38] Tim Seuré and Elias Suvanto. 2026. Finite-Field Arithmetic in CKKS. Cryptology ePrint Archive, Paper 2026/1102. https://eprint.iacr.org/2026/1102

[39] Daphné Trama, Pierre-Emmanuel Clet, Aymen Boudguiga, and Renaud Sirdey. 2023. A Homomorphic AES Evaluation in Less than 30 Seconds by Means of TFHE. In *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (Copenhagen, Denmark) *(WAHC '23)*. Association for Computing Machinery, New York, NY, USA, 79–90. doi:10.1145/3605759.3625260

[40] Ruida Wang, Jincheol Ha, Xuan Shen, Xianhui Lu, Chunling Chen, Kunpeng Wang, and Jooyoung Lee. 2025. Refined TFHE Leveled Homomorphic Evaluation and Its Application. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (Taipei, Taiwan) *(CCS '25)*. Association for Computing Machinery, New York, NY, USA, 2309–2323. doi:10.1145/3719027.3744873

[41] Zhiwei Wang, Haoqi He, Lutan Zhao, Qingyun Niu, Dan Meng, and Rui Hou. 2026. 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* (USA) *(ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 1168–1184. doi:10.1145/3779212.3790186

A Efficient slot-encoding ring-switching

Subring and trace. Before describing the main circuit, we introduce a key technique that reduces the cost of the homomorphic ring switching step. Let M and N be powers of two with $M \mid N$, and write $Y = X^{N/M}$, so that $\mathcal{R}_M = \mathbb{Z}[Y]/(Y^M + 1)$.

For $P = \sum_{i=0}^{N-1} p_i X^i \in \mathcal{R}_N$, we define

$$\mathrm{Tr}_M(P) := \sum_{i=0}^{M-1} \frac{N}{M} p_i X^{iN/M} \quad \text{and} \quad \mathrm{Ext}_M(P) := \sum_{i=0}^{M-1} \frac{N}{M} p_i Y^i,$$

so that $\mathrm{Tr}_M(P)(X) = \mathrm{Ext}_M(P)(X^{N/M})$. For a plaintext $\mathrm{pt} \in \mathcal{R}_N$ encoding $(z_0, \ldots, z_{N/2-1})$, $\mathrm{Ext}_M(\mathrm{pt})$ encodes the slot vector

$$\left(\sum_{j=0}^{N/M-1} z_{i+jM/2} \right)_{0 \le i < M/2} = \left(\sum_{i=0}^{N/M-1} \mathrm{Rot}_{i \cdot M/2}(z_0, \ldots, z_{N/2-1}) \right) \Big|_{[0, M/2)}$$

Our strategy relies on the following lemma. The key observation is that if the secret key is the embedded form $\tilde{s}_M$, Ext_M can be homomorphically computed only with automorphisms, without any key switching.

Lemma A.1. *Let $s_M \in \mathcal{R}_M$, and define $\tilde{s}_M(X) = s_M(X^{N/M}) \in R_N$. Let $\mathrm{ct} = (a, b) \in \mathcal{R}_N^2$ with $m = a \cdot \tilde{s}_M + b$. Then, $\mathrm{Ext}_M(m) = \mathrm{Ext}_M(a) \cdot s_M + \mathrm{Ext}_M(b)$.*

Proof. For $0 \le i < N/M$, we define an automorphism σ_i of R_N by $\sigma_i(X) = X^{5^{iM/2}}$. For $M > 2$ a power of two, we have $5^{iM/2} = 1 \bmod M$, so that $\frac{N}{M} 5^{iM/2} = \frac{N}{M} \bmod N$.

Hence, for $X \in \mathcal{R}_N$, we have $\sigma_i(\tilde{s}_M)(X) = s_M(X^{5^{iM/2} N/M}) = s_M(X^{N/M}) = \tilde{s}_M(X)$. Hence,

$$\sum_{i=0}^{N/M-1} \sigma_i(a) \tilde{s}_M(X) + \sum_{i=0}^{N/M-1} \sigma_i(b) = \sum_{i=0}^{N/M-1} \sigma_i(m).$$

The result follows from $\mathrm{Tr}_M = \sum_{i=0}^{N/M-1} \sigma_i$ and the relation between Tr and Ext. □

B Proof of Lemma C.1

We describe the action of FromCI from a linear algebra point of view. Assume that the ciphertext to be converted encrypts the message $\mathbf{v} \in \mathcal{R}^N$. After FromCI, one obtains a ciphertext that encrypts:

$$\mathsf{FromCI}(\mathbf{v}) \quad = (v_j + I \cdot v_{j+N/2})_{0 \le j < N/2}.$$

Hence, we have

$$\begin{aligned} \mathbf{DFT}_{N/2} \circ \mathsf{FromCI}(\mathbf{v}) &= \mathrm{Tr}_{N \to N/2} \left(\begin{pmatrix} \mathbf{DFT}_{N/2} & 0 \\ 0 & I\,\mathbf{DFT}_{N/2} \end{pmatrix} \cdot \mathbf{v} \right) \\ &= \mathrm{Tr}_{N \to N/2} \left(\left(\begin{pmatrix} 1 & 0 \\ 0 & I \end{pmatrix} \otimes \mathbf{DFT}_{N/2} \right) \cdot \mathbf{v} \right). \end{aligned}$$

We now describe the fusion. Let $\mathbf{M}_t \otimes \mathbf{I}_k$ be given with $\mathbf{M}_t \in \mathbb{R}^{t \times t}$ and $N = tk$. Since $N/2 = (t/2) \cdot k$, we have

$$\mathbf{DFT}_{N/2} = \mathbf{D}_{out} \cdot \mathbf{D}_{in}, \qquad \mathbf{D}_{in} = \mathbf{I}_{t/2} \otimes \mathbf{DFT}_k, \qquad \delta(\mathbf{D}_{out}) \subseteq k \cdot \mathbb{Z}_{t/2}.$$

Tensoring with the 2×2 block on the left gives

$$\begin{pmatrix} 1 & 0 \\ 0 & I \end{pmatrix} \otimes \mathbf{DFT}_{N/2} = \begin{pmatrix} 1 & 0 \\ 0 & I \end{pmatrix} \otimes \left(\mathbf{D}_{out} \cdot \mathbf{D}_{in}\right) = \underbrace{\left(\begin{pmatrix} 1 & 0 \\ 0 & I \end{pmatrix} \otimes \mathbf{D}_{out}\right)}_{=: \mathbf{D}'_{out}} \cdot \underbrace{\left(\mathbf{I}_2 \otimes \mathbf{D}_{in}\right)}_{= \mathbf{I}_t \otimes \mathbf{DFT}_k =: \mathbf{D}'_{in}}.$$

Since $\begin{pmatrix} 1 & 0 \\ 0 & I \end{pmatrix}$ is diagonal, the left tensor product preserves the diagonal structure, so $\delta(\mathbf{D}'_{out}) \subseteq k \cdot \mathbb{Z}_t$.

C Fusing linear layers with StC under conjugate-invariant encoding

Evaluating real-valued circuits in the conjugate-invariant ring is a well-known technique [33]: one evaluates the real-valued homomorphic circuit in the conjugate-invariant ring and then converts back to the standard ring before bootstrapping, since StC is an inherently complex-valued operation. We denote the conversion from the conjugate-invariant ring to the standard ring by FromCI (see [6] for its description). The following Lemma shows that $\mathbf{M}_t \otimes \mathbf{I}_k$ can still be absorbed:

Lemma C.1. *Let $N = tk$, and $\mathbf{DFT}_{N/2}$ be decomposed as $\mathbf{D}_{out} \cdot \mathbf{D}_{in}$, with $\mathbf{D}_{in} = \mathbf{I}_{t/2} \otimes \mathbf{DFT}_k$, and $\delta(\mathbf{D}_{out}) \subseteq k \cdot \mathbb{Z}_{t/2}$.*

Let $\mathbf{D}'_{out} = \begin{pmatrix} 1 & 0 \\ 0 & I \end{pmatrix} \otimes \mathbf{D}_{out}$ and $\mathbf{D}'_{in} = \begin{pmatrix} 1 & 0 \\ 0 & I \end{pmatrix} \otimes \mathbf{D}_{in}$. Then, we have

$$(\mathsf{StC} \circ \mathsf{FromCI}) \cdot (\mathbf{M}_t \otimes \mathbf{I}_k) = \mathsf{Tr}_{N \to N/2}\left(\left(\mathbf{D}'_{out} \cdot (\mathbf{M}_t \otimes \mathbf{I}_k)\right) \cdot \mathbf{D}'_{in}\right),$$

where $\delta\left(\mathbf{D}'_{out} \cdot (\mathbf{M}_t \otimes \mathbf{I}_k)\right) \subseteq k \cdot \mathbb{Z}_t$. Hence $\mathbf{M}_t \otimes \mathbf{I}_k$ can be absorbed into $\mathbf{D}'_{out}$.

Proof. See Appendix B. □

Cost trade-off. The linear transformation requires the same number of rotations, but is now performed in the larger ring $\mathbb{Z}[X]/(X^{2N}+1)$, which roughly doubles the cost of StC. However, we save two multiplicative levels—one from FromCI and one from the multiplication by $\mathbf{M}_t \otimes \mathbf{I}_k$.

D AES Circuit

Figure 1 illustrates the AES circuit. AES round consists of SB, SR, MC, and ARK. The number of rounds N_r depends on the AES key size: 10, 12, and 14 rounds for AES-128, AES-192, and AES-256, respectively, requiring $N_r + 1$ round keys $K_0, \ldots, K_{N_r}$ that are generated from the master key by the AES key schedule (Appendix F). As in standard AES, the final round omits the MC step.

Figure 2 illustrates the homomorphic AES evaluation circuit used in $\mathsf{Latency}_1$. Since XOR operations are replaced by homomorphic additions, the SB coefficient evaluation / linear part, SR, MC, and StC can be fused into a single linear map. The StC step may be divided into one or more stages. The ARK operation is evaluated at level 0, where the round keys are encrypted using coefficient encoding. After ARK, binary bootstrapping is performed to refresh the ciphertext and recover the delayed XOR results modulo 2.

In AES-CTR mode, both encryption and decryption use the same AES encryption circuit, and the only difference between encryption and decryption lies in whether the keystream is XORed with the plaintext or the ciphertext. This makes AES-CTR particularly suitable for homomorphic transciphering, since only one AES circuit needs to be evaluated repeatedly for different counter values.

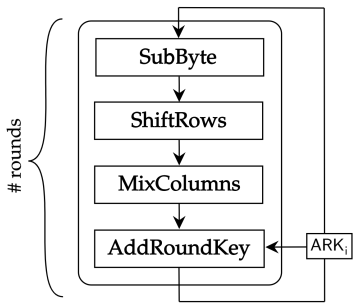


Figure 1: Standard AES round structure. The final round omits the MC step, and the total number of rounds depends on the AES key size.

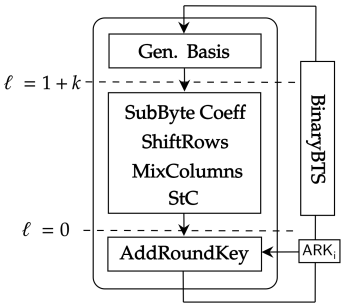


Figure 2: Homomorphic AES evaluation structure. The linear layers (SB coefficient evaluation, SR, MC, and StC) are fused into a single layer.

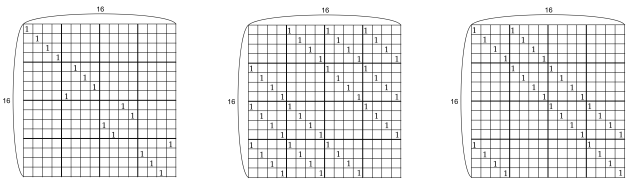
E Matrix Structure of AES Linear Steps in $\mathsf{Latency}_1$

SubByte Coefficient Matrix. Let $\mathbf{A} \in \mathbb{Z}^{8 \times 256}$ denote the LUT coefficient matrix of a single AES S-box. In the single-block setting, under the (bit, byte) packing order, the linear transformation following the Lagrange-basis SB evaluation is represented by

$$\mathbf{SB} = \mathbf{A} \otimes \mathbf{I}_{16} \in \mathbb{Z}^{128 \times 4096}.$$

ShiftRows. This step cyclically shifts each row of the byte state according to a predefined pattern and can be seen as a permutation of a 128 bit vector representation of a block.

$$\begin{bmatrix} B_0 & B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 & B_7 \\ B_8 & B_9 & B_{10} & B_{11} \\ B_{12} & B_{13} & B_{14} & B_{15} \end{bmatrix} \xrightarrow{\mathsf{SR}} \begin{bmatrix} B_0 & B_1 & B_2 & B_3 \\ B_5 & B_6 & B_7 & B_4 \\ B_{10} & B_{11} & B_8 & B_9 \\ B_{15} & B_{12} & B_{13} & B_{14} \end{bmatrix}$$



(a) $\mathbf{M}_{\mathrm{SR}}$ (b) $\mathbf{MC}_A$ (c) $\mathbf{MC}_B$

Figure 3: Sparse diagonal structures of $\mathbf{M}_{\mathrm{SR}}$, $\mathbf{MC}_A$, and $\mathbf{MC}_B$. $\mathbf{M}_{\mathrm{SR}}$ is a byte-wise permutation for SR. Both $\mathbf{MC}_A$ and $\mathbf{MC}_B$ are building block of MC linear transformation.

Under a bit-decomposed representation, SR reduces to a permutation between 128-bit bits. Based on our (bit, byte) encoding, SR operates at the byte level, and its permutation matrix can be expressed as

$$\mathbf{SR} = \mathbf{I}_8 \otimes \mathbf{M}_{\mathrm{SR}} = \mathrm{Diag}(\mathbf{M}_{\mathrm{SR}}, \ldots, \mathbf{M}_{\mathrm{SR}}) \in \mathbb{Z}_2^{128\times 128},$$

where $\mathbf{M}_{\mathrm{SR}}$ is a 16×16 submatrix, as illustrated in Figure 3 (a). Notably, $\mathbf{M}_{\mathrm{SR}}$ is not diagonal, but contains 7 non-zero diagonals. Thus, **SR** also has 7 non-zero diagonals, whose indices are given by

$$\{0, 1, 2, 3, -1, -2, -3\} \equiv \{0, 1, 2, 3, 125, 126, 127\} \pmod{128}.$$

MixColumns. Let $[B_0, B_1, B_2, B_3]^T$ denote a column of AES state in byte representation. Then, MC produces $[D_0, D_1, D_2, D_3]^T$ as

$$\begin{aligned} D_0 &= x \cdot (B_0 + B_1) + B_1 + B_2 + B_3, \\ D_1 &= x \cdot (B_1 + B_2) + B_2 + B_3 + B_0, \\ D_2 &= x \cdot (B_2 + B_3) + B_3 + B_0 + B_1, \\ D_3 &= x \cdot (B_3 + B_0) + B_0 + B_1 + B_2, \end{aligned}$$

where addition + in $GF(2^8)$ corresponds to XOR, and multiplication by x is defined as

$$\begin{aligned} &x \cdot (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0) \\ &\quad = (a_6, a_5, a_4, a_3 \oplus a_7, a_2 \oplus a_7, a_1, a_0 \oplus a_7, a_7). \end{aligned}$$

In our setting, XOR corresponds to homomorphic addition, while bit-shifting corresponds to a permutation. Therefore, the MC operation can be represented as a single linear transformation under our encoding order, i.e.,

$$\mathbf{MC} = (\mathbf{M}_{i,j})_{1\le i,j\le 8}, \quad \mathbf{M}_{i,j} = \begin{cases} \mathbf{MC}_A, & \text{if } j = i, \\ \mathbf{MC}_B, & \text{if } j = i-1, \\ \mathbf{MC}_B, & \text{if } (i,j) \in S, \\ 0, & \text{otherwise}, \end{cases}$$

$$S = \{(1, 8), (2, 8), (4, 8), (5, 8)\}.$$

where $\mathbf{MC}_A$ and $\mathbf{MC}_B$ are 16×16 matrices whose sparse diagonal structures are illustrated in Figure 3 (b) and (c).

From the **MC** block matrix structure, it has a total of 15 non-zero diagonals, whose indices are given by

$$\{4, 8, 12, 36, 48, 52, 64, 68, 84, 96, 100, 112, 116, 120, 124\}.$$

F AES Key expansion in FHE

AES-FHE evaluation can be divided into two phases: an offline phase, where AES round keys are generated from the FHE-encrypted AES secret key, and an online phase, where the homomorphic AES circuit is evaluated using the FHE-encrypted round keys. Here, we outline the details of the key expansion phase.

Algorithm 4 shows the standard AES-128 key expansion procedure. Starting from the 128-bit secret key, the algorithm generates $N_r + 1 = 11$ round keys, where each round key consists of 128 bits. The main nonlinear operation is SubWord, which applies the AES S-box independently to each byte. In contrast, RotWord is to shift byte position and Rcon is to XOR with some fixed value RC_i.

Therefore, from the homomorphic evaluation perspective, the complexity of key expansion is dominated by the repeated evaluation of SubWord, while the remaining operations contribute negligibly to the total multiplicative depth. Here, since RotWord can be interpreted as a permutation, we fuse the SB matrix and the permutation, as we did in the AES circuit. Since we follow the sequential XOR with the previously generated values, we would not fuse the StC in the circuit.

For SubWord, we use the same homomorphic evaluation method as in the main AES circuit. Similarly, XOR operations are replaced by homomorphic additions, followed by binary bootstrapping to recover the binary representation.

Algorithm 4 AES-128 Key Expansion

Require: 128-bit secret key K
Ensure: Expanded key scheduled $W[0 \ldots 43]$ words
$N_k \leftarrow 4$, $N_b \leftarrow 4$, $N_r \leftarrow 10$
Initialization:
for $i \leftarrow 0$ to $N_k - 1$ **do**
  $W[i] \leftarrow K[4i : 4i + 3]$
end for
Key Expansion:
for $i \leftarrow N_k$ to $N_b(N_r + 1) - 1$ **do**
  $temp \leftarrow W[i - 1]$
  if $i \bmod N_k = 0$ **then**
    $temp \leftarrow \mathrm{SubWord}(\mathrm{RotWord}(temp)) \oplus \mathrm{Rcon}[i/N_k]$
  end if
  $W[i] \leftarrow W[i - N_k] \oplus temp$
end for
return W

RotWord. RotWord performs a one-byte cyclic shift:

$$\mathrm{RotWord}(x_0, x_1, x_2, x_3) = (x_1, x_2, x_3, x_0).$$

Since this is a permutation, it can be fused with the SB matrix in our homomorphic evaluation.

SubWord. SubWord applies the AES S-box independently to each byte:

$$\mathrm{SubWord}(x_0, x_1, x_2, x_3) = (S(x_0), S(x_1), S(x_2), S(x_3)).$$

We use the same homomorphic SB evaluation method as in the main AES circuit.

Rcon. $\mathrm{Rcon}[i] = (RC_i, 0, 0, 0)$ denotes the round constant used in key expansion. Since this corresponds to XOR with a fixed constant, it is evaluated by homomorphic addition.

G More details on experimental evaluation

G.1 Detailed FHE Parameters

Table 7 summarizes the detailed modulus chain used in each step. FromCI denotes the conversion from the conjugate-invariant ring [33] to the original ring. This procedure requires an additional masking operation and thus consumes one multiplicative depth. Similarly, ToCI also requires one multiplicative depth. For more details on these conversions, we refer to [6].

G.2 AES-FHE Evaluation on CPU

While the main text focuses on GPU-based evaluation, we additionally report single-thread CPU (same spec noted in Section 5) results here to provide a more complete comparison across implementation settings. Table 8 compares our method with existing AES-FHE transciphering approaches on CPU.

Our latency-oriented parameters achieve significantly lower latency than existing CKKS-based methods [6, 37], while remaining competitive with TFHE-based approaches [9, 40] in terms of latency.

Our throughput-oriented parameter $\mathrm{GF}(16)_{\mathrm{MS}}$ achieves throughput significantly superior to the state-of-the-art CKKS implementation [6], while still providing substantially lower latency. This demonstrates that our method effectively balances the latency-throughput trade-off even in CPU-only environments.

For TFHE-based approaches [9, 40], we use the authors' open-source implementations and evaluate them on the same machine. For CKKS-based approaches [6, 37], we re-implement the reported parameter sets and reproduce the results on the same hardware using the HEaaN2 [20] library to ensure a fair comparison.

H Initial AddRoundKey for GF(16)

The GF(16) circuit begins with a ciphertext encrypting the initial key K_0 in GF(16) encoding with the counter block X as a plaintext operand. As it precedes the SB step of the first round, we want to keep the output of the initial AddRoundKey bounded by 1. With one operand known, this step is a GF(2)-affine map: on binary lifts

$$(x - y)^2 = (1 - 2y)\, x + y \qquad (x, y \in \{0, 1\}),$$

so for the known counter bit y it returns x if $y = 0$ and $1 - x$ if $y = 1$. This GF(2)-affine map of the key bit is a real-linear map (Section 4.4) plus a constant in GF(16) encoding.

I Matrix structure of the linear steps in the GF(16) approach

After the quasi-inversion, the remaining steps in a round are

- Recovery (conjugation)
- SubBytes affine layer
- ShiftRow
- MixColumn
- AddRoundKey

We note that all remaining steps are affine transformations; the affine part of the transformation is delayed as explained in Section 4.7.2, so that we focus on the linear part for which we derive an explicit matrix form.

We start from a single AES state consisting of 16 bytes viewed as elements of GF(256), then view GF(256) as a two-dimensional GF(16)-vector space, finally as a complex vector via GF(16) encoding. By batching multiple AES states, we obtain the matrix for a full CKKS message.

I.1 Linear steps as operators on GF(256)

We want to evaluate the composition of linear transformations:

$$\mathsf{MixColumn} \circ \mathsf{ShiftRow} \circ \mathsf{SubBytes}_L \circ \mathsf{Recovery}$$

where $\mathsf{SubBytes}_L$ denotes the SubBytes linear map and Recovery is the conjugation in GF(256) to recover the inverse from the quasi-inverse obtained from Algorithm 3.

We introduce the following operators:

(1) The recovery map $R : \mathrm{GF}(256) \to \mathrm{GF}(256)$ acts independently on each byte.
(2) The SubBytes linear part $\mathrm{SB}_L : \mathrm{GF}(256) \to \mathrm{GF}(256)$ also acts independently on each byte.
(3) The ShiftRow permutation $\mathrm{SR} : \mathrm{GF}(256)^{16} \to \mathrm{GF}(256)^{16}$ acts on a complete AES state;
(4) The MixColumn layer $\mathsf{MC} : \mathrm{GF}(256)^4 \to \mathrm{GF}(256)^4$ acts on one of the 4-byte columns of the AES state.

I.2 GF(16)-linear maps over GF(256)

Our encoding of GF(256) realizes the latter as a two-dimensional GF(16)-vector space; every element in GF(256) is a pair $[a, b]$ where $a, b \in \mathrm{GF}(16)$.

Therefore, a single AES state $(B_i)_{i=0,1,\ldots,15}$ consists of 32 GF(16) elements. In accordance to Section 4.7.1, for the discussion of one single state we consider that these 32 elements of GF(16) are ordered as:

$$(a_0, a_1, \ldots, a_{15}, b_0, b_1, \ldots, b_{15}) \tag{17}$$

where $B_i = [a_i, b_i]$.

MC. The original AES MixColumn matrix acting on the native AES representation of the finite field with 256 elements, $\mathrm{GF}(256)_{\mathrm{AES}}$, is seen as acting on vectors of $\mathrm{GF}(256)^4_{\mathrm{AES}}$ as

$$\begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix} \mapsto \begin{pmatrix} \gamma & \gamma+1 & 1 & 1 \\ 1 & \gamma & \gamma+1 & 1 \\ 1 & 1 & \gamma & \gamma+1 \\ \gamma+1 & 1 & 1 & \gamma \end{pmatrix} \begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix}$$

We let c be the map $(B_0, B_1, B_2, B_3) \mapsto (B_1, B_2, B_3, B_0)$, and L_θ be the map $B \mapsto \theta \cdot B$ on $\mathrm{GF}(256)^4_{\mathrm{AES}}$. Then, the previous map can be decomposed as

$$L_\gamma + L_{1+\gamma} \circ c + c^2 + c^3.$$

In the packing (17), we note that the map L_θ acts on the GF(16) coordinates, while c acts within each block of coordinates. When acting on $\mathrm{GF}(16)^{32}$, c is described by the matrix $\mathbf{I}_8 \otimes \mathbf{C}$ where $\mathbf{C}$ is

Table 7: Modulus chain composition. Each entry denotes $b\times k$ (k primes of b bits).

	Base	StC	FromCI	Mult	EvalMod	ToCI	CtS	P
WAHC [6]	38	20×3	33	33×18	38×8	38	38×3	61×6
WAHC* [6]	38	20×3	33	33×18	38×8	32	32×3	61×7
XBOOT [37]	35	36×2	-	37×3	38×8	-	38×3	61×3
XBOOT* [37]	35	20×2	-	28×3	35×8	-	30×3	61×4
Latency_1	21	20×1	-	25×3	31×8	-	20×2	61×6
Latency_4	21	20×2	-	25×3	31×8	-	20×2	61×6
$\text{GF}(16)_{15}$	37	24×3	36	36×3	43×8	20	20×3	61×2
GF(16)	37	24×3	36	36×3	45×8	25	25×3	61×12

Table 8: Comparison of AES-FHE transciphering methods. All measurements are conducted in a single-threaded CPU.

Schemes	Method	Latency (s)	Throughput (KB/s)	Library
BGV	[24]-SHE	252†	7.6×10^{-3}	HElib
	[24]-FHE	1050†	2.7×10^{-3}	
TFHE	[9]	35.56	4.5×10^{-4}	TFHE-rs
	[40]-Set I	17.35	9.22×10^{-4}	
	[40]-Set II	26.31	6.08×10^{-4}	
CKKS	XBOOT* [37]	2103	0.12	HEaaN2
	XBOOT* [37]-batch	1550	0.17	
	WAHC* [6]	1781	0.29	
	Ours (Latency_1)	2.74	5.8×10^{-3}	
	Ours (Latency_4)	4.24	1.5×10^{-2}	
	Ours ($\text{GF}(16)_{15}$)	6.58	0.62	
	Ours (GF(16))	13.5	0.61	
	Ours ($\text{GF}(16)_{\text{MS}}$)	46.8	0.70	

† Numbers are taken directly from the reference.

the matrix

$$\mathbf{C} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Similarly, and under our representation of GF(256) which represents γ by Γ, the map L_γ has the matrix form $\mathbf{M}_\Gamma \otimes \mathbf{I}_{16}$, where $\mathbf{M}_\Gamma \in \text{GF}(16)^{2\times2}$. Overall, writing $\theta = (\Gamma, 1+\Gamma, 1, 1)$, MC is described as

$$\sum_{i=0}^{3} (\mathbf{M}_{\theta_i} \otimes \mathbf{I}_{16}) \cdot (\mathbf{I}_8 \otimes \mathbf{C}^i) = \sum_{i=0}^{3} \mathbf{M}_{\theta_i} \otimes \mathbf{I}_4 \otimes \mathbf{C}^i \in \text{GF}(16)^{32\times32}.$$

SubBytes, Recovery. Both SubBytes and Recovery are GF(16)-linear maps on GF(256). In view of the description of the conjugation map as $[a, b] \mapsto [a+b, b]$, the recovery map can be seen, when acting on a full state, as

$$\begin{pmatrix} \mathbf{I}_{16} & \mathbf{I}_{16} \\ 0 & \mathbf{I}_{16} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \otimes \mathbf{I}_{16} \in \text{GF}(16)^{32\times32}.$$

We let $\mathbf{R} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$.

By seeing GF(256) as $\text{GF}(16)^2$, SB can be seen as a 2×2 matrix, with each entry of the matrix being a GF(2)-linear map over GF(16), in the form

$$\text{SB} = \begin{pmatrix} \text{SB}_{00} & \text{SB}_{01} \\ \text{SB}_{10} & \text{SB}_{11} \end{pmatrix}.$$

When acting on a full state, we obtain the matrix $\text{SB} \otimes \mathbf{I}_{16}$. Note that by abuse of notation, SB_{ij} are not matrices, but linear maps. In order to obtain a fully matricial formula, we would need to get down to GF(2) matrices, and would lose part of the algebraic structure of other steps. To mark the abuse of notation, we keep the notation SB in formulas rather than using $\mathbf{M}_{\text{SB}}$ as we do for other steps.

SR. ShiftRow is a map permuting the 16 elements of GF(256), hence is described by a permutation matrix in $\text{GF}(256)^{16\times16}$. It acts separately on the GF(16) coordinates and the corresponding linear map can thus be written under the form

$$I_2 \otimes \mathbf{M}_{\text{SR}}$$

where $\mathbf{M}_{\text{SR}} \in \text{GF}(16)^{16\times16}$.

Composing all the AES maps. Composing all the maps, we obtain

$$\left(\sum_{k=0}^{3} \mathbf{M}_{\theta_k} \otimes \mathbf{I}_4 \otimes \mathbf{C}^k\right) \cdot (\text{SB} \otimes \mathbf{I}_{16}) \cdot (I_2 \otimes \mathbf{M}_{\text{SR}}) \cdot (\mathbf{R} \otimes \mathbf{I}_{16})$$

which, by using the properties of the tensor product, can be rewritten as

$$\mathcal{M} = \sum_{k=0}^{3} \left(\mathbf{M}_{\theta_k} \cdot \mathrm{SB} \cdot \mathbf{R}\right) \otimes \left((\mathbf{I}_4 \otimes \mathbf{C}^k) \cdot \mathbf{M}_{\mathrm{SR}}\right) \in \mathrm{GF}(16)^{32\times 32}. \quad (18)$$

The first part of the tensor product has dimension 2×2, while the second part has dimension 16×16.

We note that the first part of the tensor product is defined over GF(16), while the second part is defined over GF(2). We can thus deduce the expression of the fused linear layer as a matrix over GF(2):

$$\widehat{\mathcal{M}} = \sum_{k=0}^{3} (\widehat{\mathbf{M}_{\theta_k}} \cdot \widehat{\mathrm{SB}} \cdot (\mathbf{R} \otimes \mathbf{I}_4)) \otimes \left((\mathbf{I}_4 \otimes \mathbf{C}^k) \cdot \mathbf{M}_{\mathrm{SR}}\right) \in \mathrm{GF}(2)^{128\times 128},$$

where $\widehat{\mathbf{M}_{\theta_k}}, \widehat{\mathrm{SB}} \in \mathrm{GF}(2)^{8\times 8}$ denote the GF(2) matrices obtained from $\mathbf{M}_{\theta_k}$, SB by restriction of scalars, with basis $1, X, X^2, X^3$ of $\mathrm{GF}(16) = \mathrm{GF}(2)[X]/(X^4 + X^3 + X^2 + X + 1)$.

The underlying tensor structure is thus

$$\widehat{\mathcal{M}} = \sum_{k=0}^{3} \widehat{\mathcal{M}}_k^{(\ell)} \otimes \widehat{\mathcal{M}}_k^{(r)},$$

with $\widehat{\mathcal{M}}_k^{(\ell)} \in \mathrm{GF}(2)^{8\times 8}$ is the part of AES linear layers that acts on individual bytes, and $\widehat{\mathcal{M}}_k^{(r)} \in \mathrm{GF}(2)^{16\times 16}$ is the part of AES linear layers that acts at the byte level.

By abuse, we still denote $\widehat{\mathcal{M}}_k^{(\ell)}$, $\widehat{\mathcal{M}}_k^{(r)}$, $\widehat{\mathcal{M}}$ liftings of the corresponding map over the integers in the sequel.

I.3 Moving to $\mathbb{C}^4$ and fusing with StC

As in Section 4.4, moving to $(\mathbb{C}^2)^2$ amounts to transporting the matrix action via the T_5 map. The T_5 maps acts separately on each of the two coordinates $[a, b]$ of an element of GF(256). Following Lemma 4.1, the action of $\mathcal{M}$ on an input $(B_0, \ldots, B_{15})$ under the packing (17) mapped to $\mathbf{z} \in \mathbb{C}^{64}$ under the packing (9) is described as

$$\mathbf{z} \mapsto \left(\mathbf{I}_2 \otimes \begin{pmatrix} \mathbf{M}_5^{(00)} & \mathbf{M}_5^{(01)} \end{pmatrix} \otimes \mathbf{I}_{16}\right) \cdot \widehat{\mathcal{M}} \cdot (\mathbf{I}_2 \otimes \mathbf{M}_5^{-1} \otimes \mathbf{I}_{16}) \cdot \begin{pmatrix} \mathbf{z} \\ \overline{\mathbf{z}} \end{pmatrix}.$$

In the AES computational circuit, this linear step is followed by the evaluation of the map T_5^{-1} which starts bootstrapping (see Section 4.5); fusing this with the AES linear maps cancels the left part of the map, and the resulting transform is then described as

$$\mathbf{z} \mapsto \left(\mathbf{I}_2 \otimes \begin{pmatrix} 1 & I & 0 & 0 \\ 0 & 0 & 1 & I \end{pmatrix} \otimes \mathbf{I}_{16}\right) \cdot \widehat{\mathcal{M}} \cdot (\mathbf{I}_2 \otimes \mathbf{M}_5^{-1} \otimes \mathbf{I}_{16}) \begin{pmatrix} \mathbf{z} \\ \overline{\mathbf{z}} \end{pmatrix}$$

In terms of structure, the action of these fused layers in CKKS message space, at the level of one AES state, is described as

$$z \mapsto \sum_{k=0}^{3} \mathbf{A}_k z + \sum_{k=0}^{3} \mathbf{B}_k \overline{z},$$

where each of $\mathbf{A}_k = \mathbf{A}_k^{(\ell)} \otimes \mathbf{A}_k^{(r)}$, $\mathbf{B}_k = \mathbf{B}_k^{(\ell)} \otimes \mathbf{B}_k^{(r)} \in \mathbb{C}^{4\times 4} \otimes \mathbb{C}^{16\times 16}$. This analysis extends to the operation on a full CKKS ciphertext by tensoring on the left by $\mathbf{I}_{N/128}$.

J Evaluation of Threshold-AES Decryption

J.1 Communication Cost Model

We summarize the assumptions used to estimate the communication latency of threshold decryption in Section 6. Our model focuses on the per-party communication cost of transmitting partial decryption shares and approximates the total latency using standard RTT and bandwidth models.

- **Round Trip Time (RTT):** The time required for a signal to travel from a sender to a receiver and back. In our benchmarks, we use 1–2 ms for LAN and 60–150 ms for WAN scenarios to reflect realistic network delays.
- **Bandwidth (BW):** The maximum data transfer rate of a network path, measured in bits per second (bps). We distinguish between high-speed internal networks (10 Gbps) and constrained wide-area links (100 Mbps–1 Gbps).
- **Round Complexity (R):** The total number of communication steps required between parties. The impact of RTT on total latency is scaled by this factor ($R \times \mathrm{RTT}$).
- **Transmission Delay (T_{trans}):** The time taken to push the protocol's data (e.g., partial decryptions) into the network, defined as:

$$T_{trans} = \frac{\text{Data Size (bits)}}{\text{Bandwidth (bps)}}$$

- **Total Communication Cost (T_{comm}):** The overall time spent on network activities, estimated as the sum of propagation delays and transmission time:

$$T_{comm} \approx (R \times \mathrm{RTT}) + \frac{\text{Total Data Sent}}{\text{Bandwidth}}$$

Table 9 summarizes the three representative network settings used in our experiments: intra-region, cross-country, and global deployment scenarios.

K Details and evaluation of threshold-FHE-based KDF

A Key Derivation Function (KDF) is a deterministic function that derives cryptographically usable keys from input key material. It serves two purposes: extracting uniform randomness from non-uniform secrets, and separating a master key into purpose-specific subkeys. Here, we focus on the second part: derivation of subkeys from master key.

The principal standards are:

- **NIST SP 800-108** [13]: Key-based KDF for deriving keys from an existing key, parameterized by a PRF (HMAC or AES-CMAC).
- **NIST SP 800-56C**: Extract-then-Expand KDF for key establishment protocols.

L The AES-CMAC Circuit

Let $E_K : \{0,1\}^{128} \to \{0,1\}^{128}$ denote AES encryption under key K. AES-CMAC consists of a one-time subkey generation step followed by a CBC-MAC–style chain with a corrected final block.

Table 9: Network configurations for benchmarking threshold decryption. RTT and bandwidth values are based on typical AWS inter-region metrics.

Scenario	Network	RTT (ms)	Bandwidth	Justification / AWS Regions
Intra-region	LAN	1–2 ms	10 Gbps	Within the same AWS region (e.g., us-east-1a to 1b).
Cross Country	WAN (US)	60–70 ms	1 Gbps	Cross-country latency (e.g., Virginia to Oregon).
Global	WAN (Inter)	90–150 ms	100 Mbps	Intercontinental connection (e.g., US to Europe).

Table 10: End-to-end latency breakdown (ms) of homomorphic AES-CMAC-based KDF in counter mode with $N = 5$ parties. Experiments are conducted on an RTX 5090. L denotes the output key size.

L	Key exap.	Subkey gen.	AES eval.	Total
128	49.7	36.1	26.4	112.2
512	53.8	40.3	29.2	123.3

Subkey generation. Two 128-bit subkeys K_1, K_2 are derived from K:

$$L = E_K(0^{128}), \qquad K_1 = L \cdot x, \qquad K_2 = L \cdot x^2,$$

where multiplication is in $\mathrm{GF}(2^{128})$ modulo $p(x) = x^{128} + x^7 + x^2 + x + 1$. The map $L \mapsto L \cdot x$ is GF(2)-linear and admits a 128×128 matrix representation: a shift by one position combined with a conditional XOR of the reduction constant `0x87` controlled by the most significant bit of L. Subkey generation therefore costs one AES call plus two such linear maps, and is performed once per key.

AES-CMAC Evaluation. Given an input M, partition it into 128-bit blocks $m_1, \ldots, m_n$ (padding m_n with $10 \cdots 0$ if $|m_n| < 128$). Define the final block as

$$m_n^\star = \begin{cases} m_n \oplus K_1 & \text{if } |m_n| = 128, \\ (m_n \,\|\, 10 \cdots 0) \oplus K_2 & \text{otherwise.} \end{cases}$$

The tag is computed via the chain

$$\begin{aligned} c_0 &= 0^{128}, \\ c_i &= E_K(c_{i-1} \oplus m_i) \text{ for } 1 \le i < n, \\ \mathrm{tag} &= E_K(c_{n-1} \oplus m_n^\star). \end{aligned}$$

The tag is the 128-bit CMAC output.

L.1 Homomorphic AES-CMAC-based KDF Evaluation

By encrypting the AES master key $\mathrm{sk}_{\mathrm{AES}}$ under a threshold FHE scheme with N-out-of-N secret sharing of the FHE decryption key, we eliminate the single point of compromise inherent in conventional key management. No individual party holds $\mathrm{sk}_{\mathrm{AES}}$ in plaintext at any stage of the derivation; recovering it requires the cooperation of all N parties. This distributed trust model is the central advantage of evaluating the KDF homomorphically.

Latency as the primary metric. KDF evaluation lies on the critical path of session establishment, so per-invocation latency governs end-to-end responsiveness. For the output lengths $L \in \{128, 512\}$ bits require only $n = 1$ CMAC invocations with our latency-aware approach. Table 10 shows an end-to-end implementation result.