Low-Depth Bootstrapping for Matrix-Native FHE

Rostin Shokri and Nektarios Georgios Tsoutsos

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

Abstract. Fully homomorphic encryption (FHE) enables computation directly on encrypted data and is increasingly researched for privacy-preserving machine-learning inference. Such workloads are dominated by matrix multiplication, which is not represented natively by conventional vector-oriented FHE schemes. CKKS is widely used for approximate encrypted inference because it supports packed real and complex arithmetic; however, matrix multiplication generally requires specific packing formats that are tailored to certain models, and involves many expensive ciphertext rotations. Depending on the matrix dimensions and packing strategy, these techniques can introduce substantial latency and memory overhead. The recently proposed Gentry–Lee (GL) scheme instead provides native slot-encoded matrix arithmetic through a multivariate ring construction while retaining approximate packed computation. Although promising, the prime-p GL instances considered here require prime-length NTTs and a three-dimensional DFT message encoding, whose efficient implementation has received limited study. Moreover, the original scheme does not specify a concrete bootstrapping procedure, which is required to support the deep computations used by large neural-network models.

In this work, we present a low-depth complex bootstrapping framework tailored to the GL scheme. Our construction combines a two-stage GL StC, in which the X-axis inversion uses native plaintext–ciphertext matrix multiplication and the W-axis inversion uses a diagonal BSGS transform, with a slice-wise adaptation of SHIP to the Gaussian (X, W) algebra. The public masked-column candidates evaluate the complete W-axis monomial action and the fine X-axis residue, leaving only the coarse X-axis displacement for hidden HMux selection and eliminating encrypted W-axis selection. We further algebraically reduce the online plaintext-encoding requirement of the masked-column interface from the fully expanded candidate-family upper bound of $8\theta\varphi(p) + 2$ to $2\varphi(p) + 2$ plaintexts per Y-slice and optimize the Rader NTT through fused kernels and other arithmetic optimizations.

Across the evaluated configurations, the faster GL axis order gives amortized per-slice StC speedups of 3.44×–4.10× at one thread and 7.75×–9.04× at 32 threads over the matched CKKS baselines. For full complex bootstrapping, the CKKS-to-GL per-slice latency ratios are 0.78–1.21 at one thread and 1.05–1.95 at 32 threads.

Keywords: Fully Homomorphic Encryption · Gentry-Lee Scheme · Matrix Arithmetic · Bootstrapping · Privacy-Preserving Machine Learning.

1 Introduction

Privacy-preserving machine learning [21, 28, 31] has received sustained attention, motivated in part by the growing availability of machine learning as a service (MLaaS), through which remotely hosted models are exposed via prediction APIs [19,30], and by the need to protect sensitive inference inputs. Fully homomorphic encryption (FHE) [14] is one of the most compelling cryptographic approaches for this setting, as it enables computation directly on encrypted data without requiring decryption by the service provider. In a typical private inference workflow, a client encrypts its input under its secret key, the server evaluates the model over ciphertexts, and the client decrypts the prediction locally. This model offers strong confidentiality guarantees, but its practicality depends critically on the efficiency of the underlying FHE scheme and, in particular, on the cost of the ciphertext-refresh operation known as bootstrapping [15].

In leveled schemes based on ring learning with errors (RLWE) [24], such as CKKS [10] and GL [16], homomorphic evaluation progressively consumes the available noise, and modulus budget. Once a computation exhausts the modulus budget and noise, further computations become impossible. Bootstrapping addresses

this problem by restoring the modulus, applying a homomorphic modular reduction, and producing a new ciphertext with renewed computational depth. As a result, bootstrapping is a central building block for deep homomorphic workloads, including modern neural networks with many layers, nonlinearities, and repeated linear algebra operations. As hardware accelerators and FHE algorithms continue to improve, enabling deeper and more expressive encrypted models, the cost of bootstrapping increasingly determines end-to-end system performance.

Many recent FHE-based private-inference systems use CKKS because it supports packed approximate arithmetic over real and complex values [8, 10, 22, 32]. Substantial work has studied the latency, precision, depth, and parallelism of CKKS bootstrapping [4,5]. Matrix multiplication is a major computational kernel in many neural-network workloads. Although CKKS is primarily organized around packed vector arithmetic, matrix multiplication need not rely exclusively on rotation-based layouts and sequences of element-wise operations [1, 23]. Reduction-based constructions reduce plaintext–ciphertext and ciphertext–ciphertext matrix multiplication to a small number of plaintext modular matrix multiplications [2, 26], while the SinC construction improves batched multiplication for small and medium matrix dimensions [9]. These methods substantially improve matrix-multiplication efficiency but retain coefficient-oriented CKKS representations that may require conversions when computations alternate between matrix operations and slot-domain SIMD operations.

The recently proposed GL scheme introduces a significantly different design point. GL provides native slot-encoded matrix arithmetic through a multivariate ring construction and a Gaussian-integer viewpoint, reducing homomorphic matrix multiplication to modular matrix multiplications on ciphertext coefficients [16]. Because graphics processing units (GPUs) are designed for high-throughput matrix multiplication, this structure motivates evaluating GL for matrix-intensive encrypted workloads, such as private inference, and investigating GPU acceleration in future work. The original GL paper notes that bootstrapping can in principle be applied to its ciphertext slices but does not specify a procedure specialized to the GL algebra [16]. Such a procedure is required to apply GL to computations that exceed its initial modulus budget.

Designing bootstrapping for GL is substantially more challenging than directly applying a CKKS construction. The difficulty is algebraic, not merely implementation-related. Conventional CKKS bootstrapping constructions typically operate over a one-dimensional power-of-two cyclotomic ring, where slots-to-coefficients (StC) and coefficients-to-slots (CtS) transforms, automorphisms, and masking structures are well understood. In contrast, a GL ciphertext slice lives in a multivariate (X, W) ring containing the cyclotomic component $\Phi_p(W)$. In this work, we restrict p to be prime. This multivariate algebra changes the semantics of rotations and the action of sparse secret-key monomials during the decryption circuit. Consequently, standard CKKS bootstrapping techniques do not transfer directly. Any practical GL bootstrapping algorithm must therefore be derived from the slice algebra itself rather than obtained by a straightforward reuse of CKKS machinery.

Concurrent work by Crockett *et al.* [12] develops a classical EvalMod-based pipeline for GL. However, EvalMod inherently requires high multiplicative depth, restricting its feasibility to large RLWE ring dimensions (e.g., $\log_2 N \geq 15$). Because EvalMod depth reductions have largely saturated, our framework is the first to bypass these limitations in GL. By adapting the shallow decryption-in-the-exponent approach of SHIP [6], we unlock low-depth bootstrapping at smaller, more flexible ring dimensions.

Our construction begins with a two-stage GL StC: the X-axis inversion uses native plaintext–ciphertext matrix multiplication, whereas the W-axis inversion uses a diagonal baby-step/giant-step (BSGS) transform, with all W-axis number-theoretic transforms (NTTs) evaluated by our optimized Rader algorithm. Each resulting Y-slice is decomposed into two Gaussian channels. Encrypted one-hot masks select the complete W-axis action, fine X-axis residue, and Gaussian phase from a public candidate family without revealing them. Only the remaining coarse X-axis displacement is applied through hidden X-only homomorphic multiplexer (HMux) rotations, after which a balanced product tree assembles the decryption phase in the exponent. Factoring the public candidates reduces the online plaintext encodings from the fully expanded candidate-family upper bound of $8\theta\varphi(p)+2$ to $2\varphi(p)+2$ per Y-slice, where θ controls the fine/coarse X-axis decomposition and $\varphi(p) = p-1$. Our construction operates slice-wise in the (X, W) ring and preserves GL's native matrix representation.

Our contributions are as follows:

- We develop a tailored StC algorithm for GL that performs the X-axis inversion using native GL plaintext–ciphertext matrix multiplication and the W-axis inversion using a BSGS linear transform.
- We derive a slice-wise SHIP-style half-bootstrap over the two Gaussian channels, show that hidden selection requires only X-axis HMux rotations, and prove the resulting construction correct.
- We reduce the number of online plaintext encodings and optimize the prime-length W-axis transform through a fused Rader NTT implementation.

Roadmap: The rest of the paper is organized as follows. In Section 2, we review the necessary background on the CKKS and GL schemes, alongside existing bootstrapping paradigms for approximate HE. In Section 3, we present our proposed low-depth bootstrapping architecture, detailing the two-stage slots-to-coefficients (StC) transform that retains the Y-slice decomposition, followed by our slice-wise decryption-in-the-exponent half-bootstrap. Section 4 provides our performance evaluation and benchmarks, including comprehensive comparisons against CKKS baselines. Section 5 discusses related work on homomorphic matrix computations and positions our contribution relative to CKKS bootstrapping algorithms and other matrix-oriented FHE schemes. Finally, Section 6 concludes the paper.

2 Background

In this section, we briefly review the CKKS and GL schemes and summarize the main paradigms for bootstrapping CKKS ciphertexts. We then highlight why bootstrapping the GL scheme is substantially more challenging than directly reusing existing CKKS constructions.

2.1 Overview of the CKKS and GL Schemes

CKKS scheme. The CKKS scheme [10] is an RLWE-based fully homomorphic encryption scheme for *approximate* arithmetic over real and complex numbers. It is typically instantiated over the power-of-two cyclotomic ring

$$R_q^{\mathrm{CKKS}} = \mathbb{Z}_q[X]/(X^N + 1),$$

where N is a power of two and q is the ciphertext modulus. Through the canonical embedding, a plaintext polynomial can encode $N/2$ complex values, enabling SIMD-style packed computation. To represent real or complex values with high precision, CKKS uses a scaling factor Δ, so that plaintext values are mapped to approximate ring elements.

A CKKS ciphertext has the usual RLWE form (b, a), and decryption satisfies $b + as \approx m$, where s is the secret key and m is the encoded plaintext. Homomorphic addition is component-wise, while homomorphic multiplication produces a higher-degree ciphertext under the enlarged secret-key basis $(1, s, s^2)$. Therefore, multiplication must be followed by *relinearization*, which is a form of key switching that returns the ciphertext to the standard two-component form. In addition, multiplication increases the effective scale, so CKKS uses *rescaling* to reduce the modulus and bring the scale back under control. These operations make CKKS especially well suited for approximate numerical workloads such as neural network inference, where additions and multiplications dominate.

GL scheme. The Gentry–Lee (GL) scheme [16] is a recently proposed RLWE-based FHE construction designed for *native matrix arithmetic*. Instead of working in a univariate cyclotomic ring and treating matrices as packed vectors, GL uses a multivariate Gaussian-integer ring

$$R_q'^{,\mathrm{GL}} = \mathbb{Z}_q[I, X, Y, W]/\langle I^2 + 1,\ X^n - I,\ Y^n - I,\ \Phi_p(W)\rangle,$$

while the secret key is sampled from the smaller ring

$$R_q^{\mathrm{GL}} = \mathbb{Z}_q[I, X, W]/\langle I^2 + 1,\ X^n - I,\ \Phi_p(W)\rangle.$$

Here, p is odd, Φ_p is the p-th cyclotomic polynomial, and, following GL's presentation, $\mathbb{Z}_p^*$ is assumed cyclic. Throughout this work, we restrict to prime p; consequently, $\varphi(p) = p - 1$, and the primitive p-th roots are exactly the nontrivial p-th roots. With two Gaussian coordinates in the basis $\{1, I\}$, X-axis degree n, and W-axis degree $\varphi(p)$, each (X, W) slice has RLWE ring dimension $N = 2n\varphi(p)$. Throughout this paper, I denotes the formal Gaussian-ring generator, represented by the residue class satisfying $I^2 = -1$ in $R_q'^{,\mathrm{GL}}$. In contrast, $\mathrm{i} = \sqrt{-1} \in \mathbb{C}$ denotes the usual imaginary unit used by the canonical embedding, roots of unity, and complex approximation formulas. These symbols are not interchangeable. Under this construction, one plaintext polynomial encodes $\varphi(p)$ matrices of size $n \times n$ by evaluation at roots of unity:

$$m(\mathrm{i}, \zeta_j^{(X)}, \zeta_k^{(Y)}, \eta_\ell) = M_{j,k}^{(\ell)},$$

where $\zeta_j^{(X)}$ and $\zeta_k^{(Y)}$ are the selected $4n$-th roots satisfying $(\zeta_j^{(X)})^n = (\zeta_k^{(Y)})^n = \mathrm{i}$, and η_ℓ ranges over the primitive p-th roots associated with the W-axis. This yields a native matrix encoding in which row and column structure are represented directly in the plaintext algebra. Under this restriction, the W-axis canonical transform is evaluated at the nontrivial p-th roots of unity and can be implemented using a length-p NTT. Rader's algorithm reduces this prime-length transform to a cyclic convolution of length $\varphi(p) = p - 1$ [27].

As a simple example, if $n = 2$ and $\varphi(p) = 2$ (which means $p = 3$), then a single GL plaintext can encode 2 separate 2×2 complex matrices at once. In this case, each Y-slice is an (X, W)-ciphertext with RLWE ring dimension $2n\varphi(p) = 8$, so one GL ciphertext can be viewed as two slice ciphertexts. The Y-slices are coefficient components of the message along the Y-axis; their joint Y-DFT reconstructs the encoded matrices.

Supported homomorphic operations in GL. The GL scheme naturally supports a broad set of matrix operations through its multivariate ring structure [16]. In particular:

- **Encrypted matrix multiplication:** it reduces plaintext–ciphertext matrix multiplication to two plaintext modular matrix multiplications, one on each ciphertext component. Ciphertext–ciphertext matrix multiplication uses four modular matrix multiplications between ciphertext components, followed by relinearization using two switching keys and rescaling.
- **Addition:** ciphertext addition is component-wise and directly corresponds to matrix addition in the encoded domain.
- **Hadamard multiplication:** element-wise matrix multiplication is realized by ordinary ciphertext multiplication, followed by relinearization and rescaling.
- **Conjugation:** complex conjugation of each encoded matrix is implemented by a ring automorphism together with key switching.
- **Row rotation:** cyclic row shifts inside each encrypted matrix are induced by automorphisms on the X-variable, followed by key switching.
- **Column rotation:** cyclic column shifts are induced by automorphisms on the Y-variable; since the secret key is independent of Y, this operation does not require key switching.
- **Inter-matrix rotation:** rotations across the batched matrices are induced by automorphisms $W \mapsto W^{g_W^j}$, where g_W generates $\mathbb{Z}_p^*$; these automorphisms permute the W-slots. This operation is distinct from multiplication by $W^v \bmod \Phi_p(W)$, which acts linearly on the W-coefficients and is evaluated in the public candidate table of our bootstrap.
- **Transpose:** matrix transposition is implemented by permuting the X- and Y-variables, followed by key switching.
- **Conjugate transpose:** Conjugate transposition is obtained through a combined automorphism on X, Y, and W, together with key switching.

These operations are important because they show that GL is not only a scheme for fast encrypted matrix multiplication, but a broader matrix-native homomorphic computation model. As noted in the GL paper, the resulting ciphertext structure can be combined with existing RLWE-based bootstrapping techniques by applying bootstrapping independently to each Y-slice [16].

Why GL is different from CKKS for bootstrapping. Although GL can be viewed as CKKS-like in the sense that it supports approximate arithmetic and is based on RLWE, its internal algebra is fundamentally different from standard CKKS. In CKKS, bootstrapping is designed around a one-dimensional power-of-two cyclotomic ring, where StC and CtS transforms, automorphisms, and masking structures are well understood. In contrast, a GL ciphertext slice lives in a multivariate (X, W) ring containing the cyclotomic component $\Phi_p(W)$, and the decryption circuit must account for Gaussian-integer coefficients and the interaction between the X- and W-axes. Consequently, standard CKKS bootstrapping methods do not transfer directly. This algebraic mismatch is precisely what makes GL bootstrapping both challenging and interesting.

2.2 Bootstrapping Paradigms for Approximate HE

Bootstrapping maps a low-level ciphertext to a high-level ciphertext that encrypts approximately the same message. Classical CKKS bootstrapping [5] follows

$$C_{\text{slot}} \xrightarrow{StC} C_{\text{coeff}} \xrightarrow{ModRaise_{q\to Q}} \widehat{C}_{\text{coeff}} \xrightarrow{CtS} \widehat{C}_{\text{slot}} \xrightarrow{EvalMod_q} C'_{\text{slot}}.$$

After modulus raising, decryption over Q yields $\mu + qk$, where μ is the desired approximate plaintext and k is an integer-valued overflow term. The CtS transform exposes these lifted coefficients as slots, and $EvalMod_q$ evaluates a polynomial approximation to $z \mapsto z \bmod q$, so that $EvalMod_q(\mu + qk) \approx \mu$. This method is accurate but its polynomial modular reduction generally requires several multiplicative levels [20, 29].

SHIP bootstrapping. SHIP [6] instead evaluates decryption in the exponent. After StC, it switches the resulting level-zero coefficient ciphertext from the dense secret s_{d} to the sparse secret $s_{\mathsf{sp}} = \sum_{e=0}^{h-1} \varepsilon_e X^{j_e}$, where j_e are its nonzero monomial positions, $\varepsilon_e \in \{-1, 1\}$, and h is its Hamming weight. Let q_0 denote the level-zero ciphertext modulus. For the resulting ciphertext (b, a) under s_{sp}, its modular decryption phase is

$$m = b + a s_{\mathsf{sp}} \pmod{q_0}.$$

If μ is the target approximate message and γ is SHIP's encoding gap, their coefficients satisfy

$$\langle m_k \rangle_{q_0} = \left\lfloor \frac{q_0 \mu_k}{\gamma} \right\rceil$$

up to the ordinary CKKS encoding and encryption error. Let $\langle f_k \rangle_{q_0} \in [-q_0/2, q_0/2)$ denote the centered lift of the k-th coefficient of $f \in R_{q_0}^{\text{CKKS}}$. Following SHIP, with $\omega = \exp(2\pi \mathrm{i}/q_0)$, define the two coefficient-wise root-of-unity vectors

$$\Omega_1(f) = \left(\omega^{\langle f_k \rangle_{q_0}}\right)_{0 \le k < N/2}, \qquad \Omega_2(f) = \left(\omega^{\langle f_{k+N/2} \rangle_{q_0}}\right)_{0 \le k < N/2}.$$

SHIP encodes these vectors and their conjugates as plaintexts. Masked selection derives the required shifted factors, and the product tree computes

$$\Omega_1(m) = \Omega_1(b) \odot \prod_{e=0}^{h-1} \Omega_1\left(\varepsilon_e a X^{j_e}\right),$$

where each shifted polynomial is reduced in $R_{q_0}^{\text{CKKS}}$, $\odot$ denotes element-wise multiplication, and the product is evaluated element-wise. Thus, Ω_1 and Ω_2 apply roots of unity coefficient-wise after reduction and centered lifting modulo q_0; they are not exponentiation in the polynomial ring. Masked-column selection chooses the candidate for the fine interval offset, and blind rotation applies the remaining coarse interval offset for each j_e. The balanced product tree then assembles the resulting factors. Combining the result with its complex conjugate extracts $\frac{\gamma}{2\pi} \sin(2\pi \langle m_k \rangle_{q_0} / q_0)$, which is approximately μ_k over SHIP's valid input range.

StC output order. SHIP evaluates the StC as one dense plaintext-matrix/ciphertext-vector product with diagonal BSGS, consuming one multiplicative level and returning the coefficients in natural order [6]. Cooley–Tukey factorizations reduce the cost of the StC and CtS linear maps at the expense of additional stages and multiplicative depth [17], and their radix-2 factors return the coefficients in bit-reversed order. The half-bootstrap selects its masks by coefficient position, so a bit-reversed StC still refreshes every coefficient, but the refreshed values reappear in bit-reversed slot order. Natural order requires a compensating permutation. Absorbing it into a transform factor preserves the multiplicative depth but can increase the number of nonzero diagonals and the evaluation cost. SHIP instead selects the direct transform [6]. Section 4.4 compares the GL StC with the standard two-stage transform and its bit-reversed output.

Window and hybrid parameters. SHIP writes the bounded support interval using a radius w_{SHIP}, giving an interval of width $2w_{\mathrm{SHIP}}$. Throughout this paper, w instead denotes the full implemented window width; thus, before any implementation-specific alignment, $w = 2w_{\mathrm{SHIP}}$, and after alignment it corresponds to SHIP's implemented interval length ℓ_0. We therefore write the e-th public interval as $j_e \in [L_e, L_e + w)$. This w is unrelated to the GL ring variable W. The position offset is decomposed as

$$j_e - L_e = r_e + \theta t_e, \qquad 0 \le r_e < \theta, \qquad 0 \le t_e < \left\lceil \frac{w}{\theta} \right\rceil .$$

The masked column selects the fine interval offset r_e, while blind rotation selects the coarse interval index t_e. Hence, θ trades masked-column material against the blind-rotation range: increasing θ uses more column candidates but fewer blind-rotation steps.

Dense-to-sparse key switching and the automorphisms used by SHIP employ hybrid key switching [18]. Abstractly, if $a = \sum_{\ell=0}^{d-1} a_\ell g_\ell$ is a gadget or RNS decomposition, an evaluation key contains encryptions $K_\ell = \mathrm{Enc}_{s'}(P g_\ell s)$, and

$$\mathrm{KSwitch}_{s \to s'}(b, a) = (b, 0) + \mathrm{ModDown}_P \left(\sum_{\ell=0}^{d-1} a_\ell K_\ell \right).$$

The auxiliary modulus P is removed after accumulation, and $d = dnum$ denotes the number of RNS decomposition digits. Thus, θ controls the masked-column/HMux split, whereas $dnum$ controls RNS hybrid key switching. Our construction adopts this low-depth SHIP pipeline and adapts its masked selection and exponent computation to each GL ciphertext slice.

2.3 Threat Model

We consider an honest-but-curious outsourced-computation setting. A client provides ciphertexts and the corresponding public and evaluation keys to an untrusted server, which follows the prescribed homomorphic computation but may attempt to infer information from the material it observes.

Our sparse secret has the same ternary description used by SHIP. Write it in a real coefficient basis as

$$s_{\mathsf{sp}} = \sum_{v=0}^{\varphi(p)-1} \sum_{u=0}^{n-1} \sum_{\chi=0}^{1} t_{2nv+2u+\chi} I^{\chi} X^u W^v, \qquad t_j \in \{-1, 0, 1\}, \qquad \|t\|_0 = h = 31.$$

Thus the search dimension is $N = 2n\varphi(p)$. The notation $\pm 1, \pm I$ for a sparse term packages its sign and its choice of real or imaginary coordinate. It does not replace a ternary secret by a dense four-valued secret, and it does not add a second independently chosen sign to each nonzero. Most real coordinates remain zero.

The public information is the same bounded-support information as in SHIP: the e-th nonzero real coordinate lies in a known interval of width w, while its position and sign are hidden. Flattening (v, u, χ) is a bijection on these coordinates. The secret remains an unknown ternary vector in a linear noisy decryption equation; expressing the public linear operator with X and W does not change the combinatorial position/sign search. We accordingly adopt SHIP's application of May's attack-cost estimate [6, 25], with

the same dimension, Hamming weight and public windows. In our full-width convention its estimate is of order $(2w)^{0.4h}$ scalar products of dimension N. Matched CKKS and GL sets use the same windows, $w = 384, 576, 1056$; SHIP's radius $\max(175, N/(2h))$ corresponds to full widths 350, 528 and 1056, so our windows are at least SHIP's. For example, the smallest full width gives $\log_2((2 \cdot 384)^{0.4 \cdot 31}) \approx 118.85$ scalar-product bits, exceeding SHIP's stated 115-bit threshold for $N \geq 2^{13}$. The cost is counted in dimension-N scalar products, rather than individual scalar operations. This is SHIP's estimate applied under the same assumptions.

Dense-to-sparse key switching also requires a separate modulus check. Only the dense-to-sparse key-switching key is encrypted under the sparse secret, at modulus $q_0 p_0$; dense-key material uses the full QP basis. We respect SHIP's published modulus bounds for sparse key switching of 55, 100 and 105 bits for $N = 2^{13}, 2^{14}, 2^{15}$, respectively, with its noise and $h = 31$ assumptions [6]. As in SHIP, secret-dependent encrypted masks and switching material rely on the evaluation-key security assumption.

CKKS and GL achieve IND-CPA security by instantiating encryption as an RLWE encryption problem: a ciphertext is a noisy affine masking of the plaintext under a secret ring element, so distinguishing encryptions of two chosen messages is computationally hard assuming RLWE is hard over the underlying ring. In CKKS this ring is the standard cyclotomic RLWE ring, while in GL the multivariate Gaussian-integer construction is chosen so that it remains reducible to standard RLWE hardness over an appropriate cyclotomic ring [16].

3 Proposed Low-Depth Bootstrapping Architecture

3.1 Core Bootstrapping Construction for the GL Algebra

A GL ciphertext holds $\varphi(p)$ complete $n \times n$ matrices. Our goal is to refresh all their entries while retaining this matrix representation. The key observation is that the secret depends on X, W, but not on Y. Expanding in powers of Y therefore gives n smaller decryption problems under the same secret. A slice is an internal coefficient component, not a complete matrix: the joint Y-DFT of all slices reconstructs the matrices.

The pipeline has three parts. First, StC converts the X, W slots into coefficients. The X transform uses native plaintext–ciphertext matrix multiplication, and the W transform uses diagonal BSGS. Their outputs expose the coefficient slices needed for decryption in the exponent. Second, after dense-to-sparse key switching, each slice follows SHIP: encrypted masks select public candidates corresponding to hidden sparse-secret terms, blind rotations complete their displacement, and a product tree multiplies the selected factors. Multiplying exponentials adds their exponents, so this shallow tree assembles the modular decryption phase without evaluating a deep modular-reduction polynomial.

Third, conjugation extracts a scaled sine, which approximates each small real message component. GL needs two such channels for its Gaussian coefficients. Their combination returns refreshed (X, W) slots, and reassembling all Y coefficients returns the original matrix encoding. The inverse Y-DFT has already made random slice messages smaller by $1/\sqrt{n}$ in RMS. This permits a different encoding gap γ from CKKS, but the forward reconstruction also amplifies slice-error RMS by $\sqrt{n}$. The two-stage StC uses one level and the product tree five, for six bootstrap levels in total.

Two-Stage Slots-to-Coefficients (StC) Transform Write $m(X, Y, W) = \sum_{y=0}^{n-1} m_y(X, W) Y^y$. Since the GL secret is independent of Y, its coefficient slices m_y can be bootstrapped independently; only the X- and W-axis transforms must be inverted homomorphically. For $S \subseteq \{X, Y, W\}$, let DFT_S mean that precisely the axes in S remain in canonical evaluation form. Thus,

$$\mathsf{DFT}_{XYW} \xrightarrow{\text{XInv}} \mathsf{DFT}_{YW} \xrightarrow{\text{WInv,Rescale}} \mathsf{DFT}_{Y},$$

while an extracted (X, W)-slice is in $\mathsf{DFT}_{\varnothing}$; DFT_{XW} denotes the refreshed slice. The two axis inversions act on independent axes and therefore commute. In the grouped schedule used in our experiments, both linear transforms are evaluated at the same ciphertext level and rescaling is delayed until the second transform finishes. This two-stage schedule consumes one RNS prime and one multiplicative level, taking a level-1

Algorithm 1 Matrix-multiplication-based X-axis inversion

Input: A GL ciphertext C in representation DFT_{XYW}
Input: $P_X = \mathrm{Encode}^{\mathrm{mat}}_{\Delta_X}(E^*_X)$
Output: A ciphertext C in representation DFT_{YW}
1: $C \leftarrow \mathrm{ConjTranspose}(C)$
2: $C \leftarrow \mathrm{MatMul}(C, P_X)$
3: $C \leftarrow \mathrm{ConjTranspose}(C)$
4: **return** C

input to level 0. Alternatively, rescaling after each transform uses two RNS primes and two levels. Either schedule may evaluate X before W or W before X; the implementation can select the faster order, and the last transform in the chosen order performs the delayed rescale. The algorithms below present the grouped X-then-W order.

These two stages are not a Cooley–Tukey factorization of one Fourier transform. They are full linear maps on separate axes: native matrix multiplication realizes the X map, and diagonal BSGS realizes the W map within each slice. BSGS only groups terms of that matrix product; it preserves the prescribed output indices. Consequently, neither axis order introduces the bit-reversal permutation discussed for CKKS. Grouping saves one rescaling operation and one RNS prime, while natural ordering comes from the maps themselves. For a fixed total plaintext-scale budget, grouping does not by itself save that budget's number of modulus bits.

The two axes are inverted differently because they mean different things. The X axis indexes rows inside each matrix, so undoing its transform is one fixed $n \times n$ matrix applied to every encoded matrix, which is exactly what GL's native plaintext–ciphertext product computes. The W axis indexes the $\varphi(p)$ matrices themselves, so its transform is a small mix across matrices, which rotation-based BSGS evaluates cheaply. The Y axis is left untouched: the secret does not involve it, so its coefficients can be separated without any homomorphic transform.

X-axis homomorphic decoding. This step realizes the X-axis inversion using the forward X-axis DFT matrix independently for each fixed W-slot. Let

$$(E_X)_{j,x} = (\zeta_j^{(X)})^x.$$

For each W-slot we need the left action $E_X M$. GL provides a right plaintext–ciphertext matrix product, so we use $E_X M = (M^* E_X^*)^*$ with $P_X = \mathrm{Encode}^{\mathrm{mat}}_{\Delta_X}(E_X^*)$, repeated across the W-slots. The matrix encoder includes GL's trace normalization, and the final grouped rescale removes this stage's scale growth.

The second conjugate transpose is needed because GL's plaintext–ciphertext product acts on the column axis Y. The first transpose moves the X axis into the column position so that the product can invert it; its intermediate $(E_X M)^*$ therefore holds the inverted axis in the Y position, while the X axis is still slot-encoded. Without the second transpose, the Y-coefficient slices extracted below would still carry a slot-encoded X axis, and each slice would need its own X-axis linear transform with X-rotation keys, which the matrix-multiplication design avoids; extracting slices along X instead is not possible, because the secret depends on X. The second transpose is thus the cheapest way to return the inverted axis to X. Both uses share the same large key.

W-axis homomorphic decoding. This step realizes the W-axis inversion using the forward W-axis DFT matrix across the $\varphi(p)$ batched-matrix slots. Let g_W generate $\mathbb{Z}_p^*$, order the slots as $\eta_r = e^{2\pi \mathrm{i} g_W^r/p}$, and define

$$(U_W)_{r,w} = \eta_r^w, \qquad (\rho_W^j z)_r = z_{r+j \bmod \varphi(p)},$$
$$d_t[r] = (U_W)_{r,r+t \bmod \varphi(p)}, \qquad U_W z = \sum_{t=0}^{\varphi(p)-1} d_t \odot \rho_W^t(z).$$

Here U_W is the forward W-axis DFT matrix, $D_t = \mathrm{Encode}_{\Delta_W}(d_t)$, and $\odot$ is element-wise plaintext–ciphertext multiplication. On ciphertexts, ρ_W^j is the automorphism $W \mapsto W^{g_W^j}$ followed by key switching

Algorithm 2 BSGS-based W-axis inversion

Input: A GL ciphertext C in representation DFT_{YW}
Input: Precomputed diagonal plaintexts $\{D_t\}_{t=0}^{\varphi(p)-1}$
Input: Baby-step size m
Output: A ciphertext A in representation DFT_Y

$G \leftarrow \left\lceil \frac{\varphi(p)}{m} \right\rceil$
for $j = 0$ to $m-1$ **do**
    $C_j \leftarrow \rho_W^j(C)$
$A \leftarrow 0$
for $b = 0$ to $G-1$ **do**
    $T_b \leftarrow 0$
    for each valid $t \in \{bm, \ldots, \min(\varphi(p)-1, bm+m-1)\}$ **do**
        $j \leftarrow t - bm$
        $T_b \leftarrow T_b + C_j \odot \rho_W^{-bm}(D_t)$
    $A \leftarrow A + \rho_W^{bm}(T_b)$
$A \leftarrow \mathrm{Rescale}(A)$
return A

with a small W-rotation key; on plaintexts it is a free permutation. It is distinct from coefficient multiplication by $W^v \bmod \Phi_p(W)$.

Algorithm 2 uses $m-1$ baby and $\lceil \varphi(p)/m \rceil - 1$ giant rotations, with hoisted decomposition reused within each group.

Slice-Wise Decryption-in-the-Exponent (Half-Bootstrapping) Decrypting a slice means adding the $h = 31$ secret terms of As_{sp} to B. SHIP's idea is to place each term in the exponent of a root of unity, so that the sum becomes a product of h factors; a product of 32 ciphertexts needs only five multiplication levels. The price is that the server must obtain each factor without learning the secret term behind it. GL adds one complication: its coefficients are Gaussian integers $m_0 + Im_1$, and the exponent trick works on integers, so the real and imaginary parts are handled as two channels and recombined at the end.

The half-bootstrap accepts a level-zero ciphertext in DFT_Y, switches it from the dense to the sparse bootstrapping secret, and extracts its n slices in $\mathsf{DFT}_\varnothing$. For one slice (B, A), write $A = A_0 + IA_1$, $B = B_0 + IB_1$, and $s_{\mathsf{sp}} = s_0 + Is_1$. For the level-zero modulus q_0, its modular phase and Gaussian channels are

$$\begin{aligned} m = B + As_{\mathsf{sp}} = m_0 + Im_1 \pmod{q_0}, \\ m_0 = B_0 + A_0 s_0 - A_1 s_1, \qquad m_1 = B_1 + A_0 s_1 + A_1 s_0. \end{aligned} \tag{1}$$

For $Z = Z_0 + IZ_1$, let $[Z]_\alpha = Z_\alpha$. At logical coordinate k, SHIP's gap-γ encoding relates the centered phase to the target approximate message $\mu = \mu_0 + I\mu_1$ by

$$m_{\alpha,k} := \langle [m]_{\alpha,k} \rangle_{q_0} = \lfloor q_0 \mu_{\alpha,k} / \gamma \rceil, \qquad \alpha \in \{0, 1\}. \tag{2}$$

Encoding and homomorphic errors are treated separately. If $\rho = \varepsilon I^\chi$, then

$$[\rho Z]_\alpha = \sigma_{\alpha,\rho} Z_{\beta_{\alpha,\rho}}, \qquad \beta_{\alpha,\rho} = \alpha \oplus \chi, \qquad \sigma_{\alpha,\rho} = \varepsilon(-1)^{(1-\alpha)\chi}. \tag{3}$$

Thus α is the output channel, while β and σ select the source component and sign. The two channels are recombined with the formal element I after both SHIP branches.

Sparse support and target. In W-major order, sparse-secret term e has

$$f_e = v_e(2n) + 2u_e + \chi_e, \qquad 0 \le u_e < n, \quad 0 \le v_e < \varphi(p), \quad \chi_e \in \{0, 1\}. \tag{4}$$

Its sign is $\varepsilon_e \in \{\pm 1\}$. It represents

$$c_e X^{u_e} W^{v_e}, \qquad c_e = \varepsilon_e I^{\chi_e} \in \{\pm 1, \pm I\}.$$

For a candidate f, define $v(f) = \lfloor f/(2n) \rfloor$, $u(f) = \lfloor (f \bmod 2n)/2 \rfloor$, and $\chi(f) = f \bmod 2$. SHIP's bounded-window assumption gives the public full-width interval

$$f_e \in \mathcal{I}_e = [L_e, L_e + w) \subseteq [0, 2n\varphi(p)), \tag{5}$$

where L_e is its public start and w its width. This W-major interval includes the Gaussian component bit and is not an interval over $\varphi(p)u + v$.

With $s_{\mathsf{sp}} = \sum_{e=0}^{h-1} c_e X^{u_e} W^{v_e}$, each channel targets

$$\begin{aligned} [m]_\alpha &= [B]_\alpha + \sum_{e=0}^{h-1} [A c_e X^{u_e} W^{v_e}]_\alpha \pmod{q_0}, \\ \omega^{[m]_\alpha} &= \omega^{[B]_\alpha} \prod_{e=0}^{h-1} \omega^{[A c_e X^{u_e} W^{v_e}]_\alpha}. \end{aligned} \tag{6}$$

Here $\omega = e^{2\pi \mathrm{i}/q_0}$. Each channel therefore needs one B-factor and one secretly selected factor per sparse term; a balanced product tree combines them. Two channels are required because $c_e \in \{\pm 1, \pm I\}$ mixes the Gaussian components.

Each secret term is a hidden address: which matrix index v_e, which row offset u_e, and which phase c_e. Enumerating every address as a public candidate would be too large, so the address is split. The coarse part of the row offset is applied by a hidden rotation (HMux), while the matrix index, the fine row residue and the phase are selected by an encrypted one-hot mask from a small public table. Row wrap-around makes the phase row-dependent: in GL a row shifted past the end of a matrix picks up the factor $X^n = I$, so the masks are chosen per row rather than per term.

Masked column and coarse X-axis HMux. For the hybrid decomposition parameter $\theta \mid n$, write

$$u_e = u_{0,e} + U_e, \qquad u_{0,e} = u_e \bmod \theta, \qquad U_e = \theta \lfloor u_e/\theta \rfloor. \tag{7}$$

SHIP's displayed algorithm and its implementation description use opposite assignments of the fine and coarse components [6]. We follow the latter: masked-column selection handles the fine X-axis residue $u_{0,e}$, and HMux applies the coarse displacement U_e. For $f \in \mathcal{I}_e$ and $\varepsilon \in \{\pm 1\}$, set

$$u_0(f) = u(f) \bmod \theta, \quad U(f) = u(f) - u_0(f), \quad \rho_{f,\varepsilon}(r_0) = \varepsilon I^{\chi(f) + \mathbf{1}[r_0 < u_0(f)] + \mathbf{1}[r_0 \geq n - U(f)]}.$$

The row-wise reachable candidates and their public union are

$$\begin{aligned} \mathcal{C}_e(r_0) &= \left\{ \big(u_0(f), v(f), \rho_{f,\varepsilon}(r_0)\big) : f \in \mathcal{I}_e,\ \varepsilon \in \{\pm 1\} \right\}, \\ \mathcal{C}_e &= \bigcup_{r_0=0}^{n-1} \mathcal{C}_e(r_0). \end{aligned} \tag{8}$$

The key stores one encrypted row-wise mask per $c \in \mathcal{C}_e$, hence $\sum_e |\mathcal{C}_e|$ masks in total.

The window also determines the possible coarse X-axis indices

$$\mathcal{H}_e = \{ \lfloor u(f)/\theta \rfloor \bmod (n/\theta) : f \in \mathcal{I}_e \} \subseteq \{ (j_{\mathrm{base},e} + \ell) \bmod (n/\theta) : 0 \leq \ell < M_e \},$$

where the right side is the smallest cyclic interval containing $\mathcal{H}_e$. Cyclicity is necessary because a W-major window can cross a W-block boundary and wrap around n/θ. The secret residual is

$$j_{\mathrm{res},e} = \left(\frac{U_e}{\theta} - j_{\mathrm{base},e} \right) \bmod \frac{n}{\theta}, \qquad 0 \leq j_{\mathrm{res},e} < M_e. \tag{9}$$

For radix $\mathcal{B} \geq 2$, the key hk_e for sparse term e encrypts the base-$\mathcal{B}$ digits of $j_{\mathrm{res},e}$; $\lceil \log_{\mathcal{B}} M_e \rceil$ HMux stages, including the public base shift, compute

$$\mathsf{HMux}_e^X(C; \mathsf{hk}_e) = \mathrm{Rot}^X_{\theta(j_{\mathrm{base},e}+j_{\mathrm{res},e})}(C) = \mathrm{Rot}^X_{U_e}(C), \tag{10}$$

with indices modulo n, without revealing U_e or $j_{\mathrm{res},e}$. At each stage, the evaluator forms the possible X-rotated branches for one radix digit and selects one using encrypted digit selectors. The selected displacements accumulate across the stages, revealing neither the digits nor the combined rotation index. Thus the masked column selects the fine X-axis residue, complete W exponent, and row-dependent Gaussian phase, while only the coarse X-axis displacement requires hidden rotation. We reserve q, q_0 for moduli; r is an output row and r_0 its row before this rotation.

Let $r_0 = (r - U_e) \bmod n$. Write $\widetilde{A}$ and $\widetilde{B}_\alpha$ for the centered coefficient arrays of A and $[B]_\alpha$ modulo q_0, and extend $\widetilde{A}^\star_{x,p-1} = 0$. Under our prime-$p$ restriction, $\Phi_p(W) = 1 + \cdots + W^{p-1} = 0$ implies $W^p = 1$ and $W^{p-1} = -(1 + \cdots + W^{p-2})$, shifting the extended vector by v and eliminating its W^{p-1} coefficient gives

$$\mathcal{A}_{u_0,v}(r_0, t) = \widetilde{A}^\star_{r_0-u_0 \bmod n,\,(t-v) \bmod p} - \widetilde{A}^\star_{r_0-u_0 \bmod n,\,(p-1-v) \bmod p}. \tag{11}$$

This public map is coefficient multiplication by $W^v \bmod \Phi_p(W)$, not a cyclic rotation of the stored coefficients. The required pre-HMux phase is

$$\rho_e(r_0) := \rho_{f_e,\varepsilon_e}(r_0) = c_e I^{\mathbf{1}[r_0 < u_{0,e}] + \mathbf{1}[r_0 \geq n - U_e]}. \tag{12}$$

The indicators encode the two $X^n = I$ wraps. Their intervals are disjoint because $u_{0,e} + U_e < n$, so the exponent is zero or one. The public channel candidate is

$$\Omega^{(\alpha)}_{u_0,v,\rho}(r_0, t) = \omega^{[\rho\, \mathcal{A}_{u_0,v}(r_0,t)]_\alpha}. \tag{13}$$

All exponents are centered modulo the level-zero modulus q_0. Candidate $c = (u_0, v, \rho) \in \mathcal{C}_e$ has row-wise mask

$$M_{e;u_0,v,\rho}(r_0, t) = \mathbf{1}[u_0 = u_{0,e}]\mathbf{1}[v = v_e]\mathbf{1}[\rho = \rho_e(r_0)]. \tag{14}$$

It is constant in t but selected independently per row because the wrap phase can vary with r_0. The key stores $\mathsf{M}_{e,c} = \mathrm{Enc}_{s_\mathrm{d}}(M_{e;u_0,v,\rho})$ under the dense output secret s_d. Writing $\Omega^{(\alpha)}_c = \Omega^{(\alpha)}_{u_0,v,\rho}$, the masked column is

$$\mathsf{MC}^{(\alpha)}_e = \mathrm{ModDown}_P\left(\sum_{c \in \mathcal{C}_e} \mathsf{M}_{e,c} \odot \mathrm{Encode}\left(\Omega^{(\alpha)}_c\right)\right). \tag{15}$$

Let $Q_\ell = \prod_{j=0}^{\ell} q_j$ be the computational modulus available to the half-bootstrap and let P be the auxiliary key-switching modulus. The masks are encrypted over PQ_ℓ, with their selectors at scale P, whereas the online candidate plaintexts are encoded at the product-tree scale Δ. Thus, $\mathrm{ModDown}_P$ in Equation (15) removes only P and returns $\mathsf{MC}^{(\alpha)}_e$ modulo Q_ℓ at scale Δ. The level-zero components $A, B \bmod q_0$ are used only to generate the public exponent tables; the encrypted masks thereby produce factors modulo Q_ℓ. The HMux is an auxiliary-modulus key switch, so it preserves this modulus and scale and consumes no multiplicative level. Exactly one candidate contributes in each row, without revealing the selected candidate or sparse-secret position. The result is already encrypted under s_d; both channels reuse the same masks and differ only in their public, ciphertext-dependent $\Omega^{(\alpha)}$ plaintexts. Consequently, no encrypted W-axis selection is required.

One sparse-secret term. For $c_e X^{u_e} W^{v_e}$, first split $u_e = u_{0,e} + U_e$. The public map $\mathcal{A}_{u_0,v}$ applies the complete $W^{v_e} \bmod \Phi_p(W)$ action and the fine X-axis residue $u_{0,e}$, while $\rho_e(r_0)$ accounts for c_e and the wraps induced by $X^n = I$. The encrypted row masks select, at row r_0, the candidate $(u_{0,e}, v_e, \rho_e(r_0))$, and the X-only HMux applies the remaining coarse X-axis displacement U_e. Thus, for each channel α,

$$D^{(\alpha)}_e \longmapsto \omega^{[A c_e X^{u_e} W^{v_e}]_\alpha},$$

which is the factor contributed by term e to the product tree.

Algorithm 3 Bootstrapping in our GL bootstrapping framework

Input: $h+1$ is a power of two; keys $\mathsf{ks}_{\mathrm{d}\to\mathsf{sp}}, \{\mathsf{M}_{e,c}\}, \{\mathsf{hk}_e\}$

function HalfBootstrap(C_{coeff})
  $C_{\mathsf{sp}} \leftarrow \mathrm{KeySwitch}(C_{\mathrm{coeff}}; \mathsf{ks}_{\mathrm{d}\to\mathsf{sp}})$
  Extract $\{C_y = (B, A)\}_{y=0}^{n-1}$ from C_{sp}
  for each slice $C_y = (B, A)$ **do**
    Decompose $A = A_0 + IA_1$ and $B = B_0 + IB_1$
    for $\alpha \in \{0, 1\}$ **do**
      $\mathsf{PT}_B^{(\alpha)} \leftarrow \mathrm{Encode}(-\mathrm{i}\gamma/(4\pi) \cdot \omega^{\widetilde{B}_\alpha})$
      Generate $\Omega_{u_0,v,\rho}^{(\alpha)}$ from A using Equation (13)
    for each sparse term e and channel α **do**
      $\mathsf{MC}_e^{(\alpha)} \leftarrow \mathrm{MaskedColumn}_e(\Omega^{(\alpha)}; \{\mathsf{M}_{e,c}\}_c)$; $D_e^{(\alpha)} \leftarrow \mathsf{HMux}_e^X(\mathsf{MC}_e^{(\alpha)}; \mathsf{hk}_e)$
    for $\alpha \in \{0, 1\}$ **do**
      $P_\alpha \leftarrow \mathsf{PT}_B^{(\alpha)} \cdot \prod_e D_e^{(\alpha)}$
    $C'_y \leftarrow (P_0 + IP_1) + \mathrm{Conj}_{XW}(P_0 - IP_1)$
  return $\mathrm{Reassemble}_Y(\{C'_y\}) \in \mathsf{DFT}_{XYW}$
function FullBootstrap(C)
  $C_{\mathrm{coeff}} \leftarrow StC_{\mathrm{GL}}(C)$
  return HalfBootstrap(C_{coeff})

Worked example. Let $n = 4$, $\theta = 2$, $p = 5$, and let one sparse term be $I\,X^3W^2$, so $u_e = 3$, $v_e = 2$ and $c_e = I$. The split $u_e = u_{0,e} + U_e$ gives $u_{0,e} = 1$ for the masked column and $U_e = 2$ for the HMux. Along W, multiplying the coefficient vector (a_0, a_1, a_2, a_3) of A by W^2 and reducing by $W^4 = -(1 + W + W^2 + W^3)$ gives $(a_3 - a_2,\ -a_2,\ a_0 - a_2,\ a_1 - a_2)$: every coefficient moves up by two positions, and the coefficient that lands on W^4 is subtracted from all four. This is Equation (11) with $v = 2$. Along X, the fine residue $u_{0,e} = 1$ shifts the rows by one, so row 0 wraps past $X^4 = I$; the coarse displacement $U_e = 2$ later wraps rows 2 and 3. By Equation (12), row 1 therefore needs the phase I, and rows 0, 2 and 3 need $I \cdot I = -1$. Two of the stored masks for this term are nonzero: one selects the candidate $(1, 2, I)$ on row 1, the other selects $(1, 2, -1)$ on the remaining rows. The masked column assembles the four rows from the public tables, and the HMux finally rotates the result by two rows. The server sees only the public window and the encrypted masks; u_e, v_e and c_e are never revealed.

Plaintext encoding. $\mathrm{Encode}(z)$ applies the inverse X, W canonical transforms, scales, and rounds a logical table z, yielding a plaintext whose logical state is DFT_{XW}. The public, ciphertext-dependent B-factor and Ω tables are generated online from A, B; only the encrypted masks and HMux keys are pre-generated. For dense-key $C \in \mathsf{DFT}_{XW}$, $\mathrm{Conj}_{XW}(C)$ is the homomorphic conjugation $(I, X, W) \mapsto (-I, X^{-1}, W^{-1})$, including key switching, and therefore conjugates every logical value.

Algorithm 3 exposes both the half- and full-bootstrap interfaces. For each slice, the evaluator generates public candidate plaintexts from A and one B-dependent base plaintext per channel. For each sparse-secret term e, the masked column selects the fine X-axis residue, complete W action, and Gaussian phase, while the term-specific HMux applies the remaining coarse X-axis displacement. A balanced tree multiplies the resulting h factors with the B-factor, using relinearization and rescaling. Homomorphic conjugation extracts the scaled sine approximation, and the two channels are recombined using the formal element I. For $h = 31$, the tree consumes five levels; together with the two-stage StC, the bootstrap consumes six.

Output representation. Each channel leaves the slice in DFT_{XW}. Multiplication of the second channel by formal I maps $(z_0, z_1) \mapsto (-z_1, z_0)$, after which the channels are added. For a fixed (X, W) coordinate, let $(F_Y)_{j,y} = (\zeta_j^{(Y)})^y$. GL stores $c = F_Y^{-1}M = n^{-1}F_Y^*M$; reassembly places refreshed c_y at Y^y, so its logical

DFT_{XYW} value is

$$\widehat{M}_j = \sum_{y=0}^{n-1} \widehat{c}_y (\zeta_j^{(Y)})^y. \tag{16}$$

No homomorphic Y-transform is performed: the refreshed slices are placed directly at their corresponding Y-coefficients, and their joint Y-DFT gives Equation (16).

Formal Correctness of the Half-Bootstrap

Statement. Let $m = B + As_{\mathsf{sp}} \pmod{q_0}$ and let μ satisfy Equation (2). Within SHIP's approximation range [6], Algorithm 3 returns a dense-key encryption of $\widehat{\mu} = \widehat{\mu}_0 + I\widehat{\mu}_1$, where

$$\widehat{\mu}_{\alpha,k} = \frac{\gamma}{2\pi} \sin\!\left(\frac{2\pi m_{\alpha,k}}{q_0}\right) + \varepsilon^{\mathrm{HE}}_{\alpha,k} \approx \mu_{\alpha,k}. \tag{17}$$

Here $\varepsilon^{\mathrm{HE}}$ collects encoding, key-switching, rescaling, multiplication, and ciphertext-noise errors; the remaining deviation is the rounding and sine-approximation error. The argument below establishes the decryption identity up to these errors. Section 3.1 analyzes the sine approximation and reconstruction; complete-bootstrap error is evaluated in Section 4.5.

Proof. For channel α, Equations (11) and (12) show that, at row r_0, the row-wise mask selects the candidate $(u_{0,e}, v_e, \rho_e(r_0))$. Equation (10) then applies U_e, so at every logical coordinate

$$D_e^{(\alpha)} \longmapsto \omega^{[Ac_e X^{u_e} W^{v_e}]_\alpha},$$

up to homomorphic error. Multiplying by the base plaintext gives

$$P_\alpha = \left(-\frac{\mathrm{i}\gamma}{4\pi}\omega^{[B]_\alpha}\right)\prod_e D_e^{(\alpha)} = \frac{\gamma}{4\pi\mathrm{i}}\omega^{[m]_\alpha}.$$

At coordinate k, homomorphic conjugation yields

$$P_{\alpha,k} = \frac{\gamma}{4\pi\mathrm{i}}\exp\!\left(\frac{2\pi\mathrm{i}m_{\alpha,k}}{q_0}\right), \qquad \overline{P_{\alpha,k}} = -\frac{\gamma}{4\pi\mathrm{i}}\exp\!\left(-\frac{2\pi\mathrm{i}m_{\alpha,k}}{q_0}\right).$$

Therefore

$$P_{\alpha,k} + \overline{P_{\alpha,k}} = \frac{\gamma}{2\pi}\sin\!\left(\frac{2\pi m_{\alpha,k}}{q_0}\right), \qquad \frac{2\pi m_{\alpha,k}}{q_0} = \frac{2\pi\mu_{\alpha,k}}{\gamma} + \delta_{\alpha,k}, \qquad |\delta_{\alpha,k}| \le \frac{\pi}{q_0}.$$

The near-linear sine range gives Equation (17). Applying the argument to both channels and using $(P_0 + IP_1) + \mathrm{Conj}(P_0 - IP_1) = (P_0 + \mathrm{Conj}(P_0)) + I(P_1 + \mathrm{Conj}(P_1))$ gives $\widehat{\mu} \approx \mu$ in DFT_{XW}. Equation (16) proves the reassembled half-bootstrap; composing with GL StC proves the full bootstrap. □

Error Analysis Two effects shape the GL error. Reassembling the refreshed slices is a Y-DFT, which spreads each slice's error over all n outputs; the error grows by $\sqrt{n}$ in RMS, exactly undoing the $1/\sqrt{n}$ shrink of the messages on the way in. And the sine approximation at the heart of SHIP is most accurate for small inputs, which the shrunken slice messages are; this is why GL can use a smaller gap than CKKS.

Reconstruction. For one fixed (X, W) position, write $c = F_Y^{-1}M$ and let $e = \widehat{c} - c$ be the errors of the refreshed slices. The decoded matrix error is $E = F_Y e + \delta$, where δ is numerical reconstruction error. Since $F_Y^* F_Y = nI$,

$$\mathrm{RMS}(c) = \mathrm{RMS}(M)/\sqrt{n}, \qquad \mathrm{RMS}(F_Y e) = \sqrt{n}\ \mathrm{RMS}(e).$$

Both identities hold for complete vectors without an independence assumption. Reassembly introduces no new homomorphic operation, but the unnormalized forward transform amplifies existing slice errors. There is no corresponding exact multiplier for maximum or mean absolute error.

Sine approximation. The $1/\sqrt{n}$ normalization also sets the sine error: for a real component t, $\frac{\gamma}{2\pi}\sin(2\pi t/\gamma) - t = -\frac{2\pi^2}{3}\,t^3/\gamma^2 + O(t^5/\gamma^4)$, cubic in the input to leading order. Slice entries are $\sqrt{n}$ smaller than matrix entries, so their leading sine error is $n^{3/2}$ smaller, and reconstruction multiplies it by $\sqrt{n}$. Since this error scales as γ^{-2}, matching the CKKS sine error to leading order allows a gap about $\sqrt{n}$ smaller, $\gamma_{\mathrm{GL}} \approx \gamma_{\mathrm{CKKS}}/\sqrt{n}$, up to a constant of order one set by the shape of the input distribution. This is a statistical scaling, not a worst-case guarantee for structured matrices, and it predicts the approximation error only.

Maximum error depends on both the error distribution and the number of values being compared. A complete GL output contains n times as many complex values as one CKKS ciphertext at the same RLWE dimension.

3.2 Algorithmic and Arithmetic Optimizations

The public candidate tables are regenerated for every slice, so their size matters. Rather than one plaintext per combination of row residue, matrix index and phase, we store one plaintext per matrix index and obtain the others by permuting rows and conjugating, which are free on plaintexts.

Compact plaintext generation. For real-valued CKKS SHIP, five plaintexts depend on the ciphertext: one B-dependent and four A-dependent [6]. A complex bootstrap needs six distinct plaintexts, two B-dependent and four shared A-dependent. For GL, materializing both channels, θ fine X-axis residues, $\varphi(p)$ W exponents, and four phases gives the fully expanded upper bound $8\theta\varphi(p)+2$, although public windows may reach fewer candidates. We instead factor Equation (11). For $\beta \in \{0,1\}$, define

$$F^{+}_{\beta,y}(r_0) = \omega^{[\widetilde{A}^{\star}_{r_0,y}]_\beta}, \qquad F^{-}_{\beta,y}(r_0) = \omega^{-[\widetilde{A}^{\star}_{r_0,y}]_\beta},$$

and extend both tables by $F^{+}_{\beta,p-1} = F^{-}_{\beta,p-1} = 1$, and let $\mathit{\Pi}^{+}_{\beta,v}$ denote the stored positive base for component β and $u_0 = 0$:

$$\mathit{\Pi}^{+}_{\beta,v}(r_0,t) = F^{+}_{\beta,(t-v) \bmod p}(r_0)\cdot F^{-}_{\beta,(p-1-v) \bmod p}(r_0).$$

For channel α and phase ρ, Equation (3) selects $\mathit{\Pi}^{+}_{\beta_{\alpha,\rho},v}$ or its conjugate according to $\sigma_{\alpha,\rho}$. The fine-residue X-index mapping then gives $\mathit{\Omega}^{(\alpha)}_{u_0,v,\rho}$. Both Gaussian channels therefore share $2\varphi(p)$ base plaintexts, giving $2\varphi(p)+2$ encodings per Y-slice including the two B-plaintexts. These choices only reinterpret the stored plaintext entries; they require no ciphertext automorphism, additional encoding, temporary plaintext, or key switch. Plaintexts are generated and consumed independently for each Y-slice.

Fused Rader NTT. For the prime-p instances used here, $d = \varphi(p) = p-1$ is a power of two. Rader's algorithm [27] reindexes the nonzero indices by powers of a generator $g \in \mathbb{Z}_p^*$ and reduces the length-p transform to a length-d cyclic convolution. Fix a primitive p-th root $\eta \in \mathbb{Z}_q$, write $x(W) = \sum_{j=0}^{p-2} x_j W^j$, and set $x_{p-1} = 0$. For $a_t = x_{g^t}$ and $b^{+}_t = \eta^{g^{-t}}$,

$$Y_m = x(\eta^{g^m}) = x_0 + (a * b^{+})_{-m \bmod d}, \qquad (a*b)_r = \sum_{t=0}^{d-1} a_t b_{r-t \bmod d}.$$

Let $\mathcal{F}_d$ and $\mathcal{G}_d$ be the forward and unnormalized inverse radix-two NTTs over $\mathbb{Z}_q$, so that $\mathcal{G}_d(\mathcal{F}_d(z)) = dz$. Then

$$a * b^{+} = \mathcal{G}_d\big(\mathcal{F}_d(a) \odot K^{+}\big), \qquad K^{+} = d^{-1}\mathcal{F}_d(b^{+}).$$

Thus, the online computation is one generator permutation, one forward NTT, one pointwise product, one unnormalized inverse NTT, and output reconstruction. The inverse uses $b^{-}_t = \eta^{-g^{-t}}$ analogously, with $K^{-} = (pd)^{-1}\mathcal{F}_d(b^{-})$; the factor p^{-1} supplies the inverse length-p normalization. All inverses in these expressions are modulo q.

The kernels $K^{\pm}$ depend only on (p, q) and are computed offline. We fuse the pointwise product with the inverse transform and use $\mathcal{F}_d(a)_0 = \sum_t a_t$ together with $\sum_t b_t^{\pm} = -1$ to simplify the zero-frequency contribution. Adding x_0 in this frequency bin broadcasts it to all outputs of $\mathcal{G}_d$. For the inverse transform, the omitted evaluation at 1 is recovered from the unused convolution entry, avoiding a separate weighted sum.

4 Performance Evaluation and Benchmarks

All experiments were conducted on an Ubuntu workstation equipped with an AMD 7970X CPU with 32 cores and 160 GB of RAM. All C++ code was compiled with GCC 13.3.0. CKKS bootstrapping, isolated StC and complete GL bootstrapping were measured with 1, 2, 4, 8, 16 and 32 computation threads. Context and evaluation-key generation use 32 threads and are excluded from online latency.

4.1 Implementation Methodology

We implemented both CKKS and GL from scratch in C++.We use OpenBLAS [33] as the backend for dense matrix multiplication. Both implementations use hybrid key switching [18] and the same rescaling, ModUp, and ModDown algorithms. The core NTT butterflies and radix algorithms are also implemented from scratch. In both CKKS and GL, we delay the final rescaling of the product tree until after the final conjugation to reduce its noise contribution. Because the original implementation is not publicly available, we implemented CKKS SHIP bootstrapping [6] from scratch and compare our measurements with its reported results. The Desilo implementation of GL is proprietary [13]. Although this library allows users to execute FHE operations via a Python interface, its closed-source nature obscures the internal cryptographic mechanisms. Therefore, evaluating and extending the scheme necessitated developing a custom C++ implementation.

A GL ciphertext contains $\varphi(p)$ matrices of size $n \times n$ and n internal Y-slices. We report total ciphertext latency T, amortized per-slice latency T/n and per-matrix latency $T/\varphi(p)$. Each slice contains $N/2$ complex values, so GL/CKKS latency ratios compare one GL slice with one CKKS ciphertext. Here, $p = 17$ and $\varphi(p) = 16$.

Both schemes sample real and imaginary input components independently and uniformly from $[-1, 1)$. For decrypted output $\widehat{m}_j$ and original input m_j, let $e_j = |\widehat{m}_j - m_j|$. Maximum and mean precision are $-\log_2(\max_j e_j)$ and $-\log_2(\text{mean}_j\, e_j)$, respectively, over all output entries. Real-valued experiments use the real-component error.

4.2 Cryptographic Parameter Selection

Table 1 summarizes the secure GL and CKKS SHIP parameter sets utilized in our experiments. The suffix `-prec` denotes a precision-optimized parameter set. Here, Q denotes the ciphertext-modulus chain and P denotes the auxiliary modulus used for hybrid key switching. GL bootstraps the inverse-Y coefficients of each slice, which are about $\sqrt{n}$ times smaller than the matrix entries for the inputs of Section 4.1. The GL gap γ is therefore about $\sqrt{n}$ times smaller than the CKKS gap for the same numerical range: scaled back by $\sqrt{n}$, the GL gaps are 32, 2896, 5793 and 6144 for n256p17, n512p17, n512p17-prec and n1024p17, compared with 32, 2048 and 2048 for the matched CKKS sets (Section 3.1). The gap $\gamma = q_0/\Delta$ depends on the modulus allocation: a larger gap reduces sine bias but can leave less precision for linear transforms, blind rotations and rescaling. The StC limb is the single RNS prime removed after both transform stages. The application limbs give the multiplicative levels available outside bootstrapping. They exclude the six levels consumed by the complete bootstrap: five for the balanced product tree and one for the two-stage StC, as described in Section 3.1.

All parameter sets target 128 bits of security and use a dense uniform-ternary main secret. For the sparse secret used during bootstrapping, we follow SHIP [6]: we use its bounded-window assumption and sparse-secret Hamming weight $h = 31$, which SHIP estimates to provide 128 bits of security under its bounded-window analysis. The sparse key-switching moduli also satisfy SHIP's bounds described in Section 2.

Table 1. Secure parameter sets utilized for our GL bootstrapping framework and CKKS SHIP. Every GL set uses $p = 17$, so $\varphi(p) = 16$ and $N = 2n\varphi(p)$. Q widths list q_0 and the StC limb, then the application limb(s), then the five product-tree limbs; the rightmost column gives the 128-bit $\log_2(PQ)$ limit for a uniform-ternary secret from the recent homomorphic encryption standard [3].

Scheme	Parameter set	N	Q widths (bits)	P widths	dnum	γ	$\log(PQ)$	Max. $\log(PQ)$
GL	n256p17	2^{13}	25,17 | 24 | 24,23,23,24,24	30	8	2	214	214
GL	n512p17	2^{14}	44,40 | 37 | 43,42,44,43,45	56,36	4	128	430	430
GL	n512p17-prec	2^{14}	48,47 | 40 | 48,47,49,49,50	52	8	256	430	430
GL	n1024p17	2^{15}	51,45 | 43×8 | 44,48,49,49,52	53,45,44,44	4	192	868	868
CKKS	LL13-secure	2^{13}	25,21 | 20 | 22,23,24,25,26	28	8	32	214	214
CKKS	LL14-secure	2^{14}	48,37 | 37 | 43,43,43,43,44	48,44	4	2048	430	430
CKKS	HT15-secure	2^{15}	50,39 | 39×8 | 48×5	55,55,55,55	4	2048	861	868

We report w as the complete implemented window width, rather than SHIP's window radius. The recent homomorphic encryption standard [3] gives $\log_2(PQ)$ limits of 214, 430, and 868 bits for $N = 2^{13}, 2^{14}, 2^{15}$, respectively. All utilized GL and matched CKKS parameter sets remain within their corresponding limits.

4.3 Baseline Validation Against CKKS SHIP

We compare our CKKS SHIP implementation against the reported results of SHIP in [6]. We report the published SHIP parameter sets and evaluation numbers alongside our own measurements in order to assess how closely our implementation matches their reported performance. This comparison uses the original SHIP parameter sets to match the published experiment; the security-adjusted parameter sets used elsewhere in this work are reported in Table 1. For consistency, all latency values in Table 2 are expressed in seconds. The table reports maximum precision, latencies from one to 32 threads, and the ratio $T_{\text{SHIP}}/T_{\text{ours}}$ at each thread count.

As shown in Table 2, our implementation is faster at one and two threads for all four parameter sets and slower at four and eight threads. At 16 and 32 threads it is faster for HT14, slower for LL14 and HT15, and alternates for LL13. These results should be interpreted as an implementation-level comparison because the measurements use different hardware and the reported SHIP implementation uses hand-optimized AVX2 kernels.

4.4 GL vs CKKS StC

Both the CKKS and GL StC transforms use two stages with one final rescaling. CKKS produces bit-reversed output, whereas GL produces natural-order output. For GL, either the X or W transform can be evaluated first. Table 3 compares the two schemes at 1 to 32 threads. The CKKS bootstrap in Section 4.5 uses SHIP's direct one-stage StC; Table 3 isolates the two-stage transform.

Using the faster GL axis order, the amortized per-slice StC latency gives a 3.44×–4.10× speedup over one CKKS ciphertext at one thread and a 7.75×–9.04× speedup at 32 threads. The advantage persists as the absolute transform latency increases with the ring dimension. This behavior is consistent with the different decomposition of the linear transform: GL evaluates the X-axis inversion through native plaintext–ciphertext matrix multiplication and confines the diagonal BSGS transform to the W-axis, whereas CKKS applies rotation-based linear transformations over its one-dimensional packed vector. Thus, the larger parameter sets preserve the structural advantage of the GL transform, although the precise speedup depends on the transform schedule and key-switching parameters.

W-first is faster for n512p17 and n512p17-prec at every thread count, and X-first for n1024p17; for n256p17, W-first is faster at 1, 2 and 32 threads and X-first at 4 to 16 threads. All full bootstraps in Section 4.5 use the W-first order.

Table 2. Parameter sets and evaluation results for real-number bootstrapping reported in [6], and our implementation, at 1 to 32 computation threads. Precision is in bits; SHIP's values follow its published convention, and ours reports maximum precision. Latencies are seconds per ciphertext. The ratio rows give $T_{\text{SHIP}}/T_{\text{ours}}$ at each thread count.

Parameter set	N	SHIP $\log(PQ)$	Implementation	Precision (bits)	Latency (s) by computation threads					
					1	2	4	8	16	32
LL13	2^{13}	218	SHIP	4.45	3.02	1.59	0.80	0.45	0.29	0.22
			Ours	4.58	2.52	1.40	0.85	0.48	0.27	0.23
			$T_{\text{SHIP}}/T_{\text{ours}}$	–	1.20	1.14	0.94	0.94	1.07	0.96
LL14	2^{14}	437	SHIP	16.90	4.91	2.57	1.29	0.70	0.42	0.33
			Ours	17.20	4.19	2.32	1.37	0.77	0.50	0.38
			$T_{\text{SHIP}}/T_{\text{ours}}$	–	1.17	1.11	0.94	0.91	0.84	0.87
HT14	2^{14}	438	SHIP	4.55	13.20	7.30	3.62	2.01	1.49	1.09
			Ours	4.78	12.83	6.78	3.80	2.22	1.27	1.06
			$T_{\text{SHIP}}/T_{\text{ours}}$	–	1.03	1.08	0.95	0.91	1.17	1.03
HT15	2^{15}	881	SHIP	18.10	29.60	12.10	6.09	3.30	2.00	1.63
			Ours	18.53	22.48	11.87	6.80	4.04	2.43	2.06
			$T_{\text{SHIP}}/T_{\text{ours}}$	–	1.32	1.02	0.90	0.82	0.82	0.79

Table 4 repeats the measurement with rescaling after each stage. Per-stage rescaling costs up to 54% more latency for CKKS and up to 60% for n256p17; for the n512 sets it costs 4% to 24% more, and for n1024p17, where the X-axis matrix multiplication dominates, the two schedules are within 4% of each other. The one-rescale schedule saves one rescaling and carries one limb fewer through the first stage, but not the StC modulus bits.

4.5 Full Bootstrap Latency and Precision

Table 5 reports GL latency per slice, per matrix and per ciphertext at 1 to 32 threads.

At 32 threads, n256 and n1024 achieve per-slice speedups of 1.95× and 1.29× over one CKKS ciphertext at the same RLWE dimension and usable depth. In these experiments, n256 has 0.72 fewer maximum and 0.06 more mean precision bits; n1024 has 0.45 fewer maximum and 0.97 more mean precision bits. The four-digit n512 configuration achieves a 1.45× per-slice speedup, with 1.59 fewer maximum and 0.67 fewer mean bits. With four digits, the auxiliary modulus must cover a two-limb digit, so P takes 92 of the 430 bits and the product-tree factors are encoded at about 44 bits; the measured error is then dominated by product-tree rescaling and HMux rounding, and no four-digit allocation we tested closed the gap (Table 6). Eight single-limb digits reduce P to 52 bits and release 40 bits to the StC limb and the product tree, raising the factor scale to about 49 bits, at the cost of twice the key-switching digits. The n512p17-prec configuration improves maximum and mean precision over LL14-secure by 1.82 and 2.18 bits, respectively, with a 1.05× per-slice speedup.

To test whether a different four-digit allocation closes the gap, we screened 208 four-digit n512p17 configurations on 32 Y-slices each, varying γ, the Q/P allocation, the application scale, the StC split and order, and θ, and validated 31 of them with complete bootstraps. Table 6 lists ten representative validated sets with the default, the eight-digit set and the CKKS reference: a γ sweep from 64 to 256 at a 37-bit application scale and application scales of 38, 40 and 42 bits. Mean precision peaks at 18.84 bits for γ between 96 and 128 and falls on both sides. The best validated mean, 18.88 bits at a 38-bit application scale, is 0.57 bits below the CKKS mean at a higher per-slice latency; the best validated maximum, 16.12 bits, is 1.37 bits below the CKKS maximum. The remaining axes are flat: $\theta = 4$, 8, 16 and 32 at the default Q and P widths give 18.79, 18.83, 18.83 and 18.84 mean bits, the P splits 56/36, 54/38, 50/42 and 46/46 lie within 0.01 bits

Table 3. Two-stage GL and CKKS StC with one final rescaling, at 1 to 32 computation threads. GL latency is given per slice (equal to one CKKS ciphertext), per matrix and per ciphertext; bold marks the faster GL axis order at each thread count. The CKKS/GL rows give CKKS latency divided by GL per-slice latency.

Parameter set	Order	Latency	Computation threads					
			1	2	4	8	16	32
LL13-secure	–	per ct (ms)	33.76	21.09	11.39	10.85	8.94	8.62
n256p17	$W \to X$	per slice (ms)	**9.18**	**6.71**	3.16	1.88	1.17	**1.02**
		per matrix (ms)	146.88	107.33	50.54	30.03	18.64	16.29
		total (s)	2.35	1.72	0.81	0.48	0.30	0.26
		CKKS/GL	3.68	3.14	3.60	5.77	7.64	8.45
	$X \to W$	per slice (ms)	12.32	7.84	**2.99**	**1.77**	**1.07**	1.07
		per matrix (ms)	197.17	125.50	47.81	28.38	17.07	17.15
		total (s)	3.15	2.01	0.76	0.45	0.27	0.27
		CKKS/GL	2.74	2.69	3.81	6.13	8.36	8.06
LL14-secure	–	per ct (ms)	85.64	50.54	27.63	15.20	18.30	18.98
n512p17	$W \to X$	per slice (ms)	**23.56**	**12.45**	**7.18**	**3.97**	**2.48**	**2.19**
		per matrix (ms)	754.03	398.28	229.61	126.89	79.25	70.05
		total (s)	12.06	6.37	3.67	2.03	1.27	1.12
		CKKS/GL	3.63	4.06	3.85	3.83	7.38	8.67
	$X \to W$	per slice (ms)	24.75	12.97	7.41	4.15	2.69	2.30
		per matrix (ms)	792.01	415.03	237.08	132.90	86.17	73.64
		total (s)	12.67	6.64	3.79	2.13	1.38	1.18
		CKKS/GL	3.46	3.90	3.73	3.66	6.80	8.25
n512p17-prec	$W \to X$	per slice (ms)	**24.86**	**13.38**	**7.25**	**4.28**	**2.45**	**2.10**
		per matrix (ms)	795.67	428.12	232.08	137.11	78.25	67.26
		total (s)	12.73	6.85	3.71	2.19	1.25	1.08
		CKKS/GL	3.44	3.78	3.81	3.55	7.47	9.04
	$X \to W$	per slice (ms)	26.85	14.23	7.55	4.45	2.65	2.35
		per matrix (ms)	859.23	455.22	241.65	142.42	84.81	75.15
		total (s)	13.75	7.28	3.87	2.28	1.36	1.20
		CKKS/GL	3.19	3.55	3.66	3.42	6.91	8.08
HT15-secure	–	per ct (ms)	351.07	180.57	94.56	61.54	47.00	51.83
n1024p17	$W \to X$	per slice (ms)	89.45	49.10	27.28	16.66	9.35	7.36
		per matrix (ms)	5724.62	3142.24	1746.03	1066.27	598.20	471.00
		total (s)	91.59	50.28	27.94	17.06	9.57	7.54
		CKKS/GL	3.92	3.68	3.47	3.69	5.03	7.04
	$X \to W$	per slice (ms)	**85.70**	**45.55**	**24.67**	**15.39**	**8.16**	**6.69**
		per matrix (ms)	5484.57	2915.18	1578.74	984.74	522.26	428.20
		total (s)	87.75	46.64	25.26	15.76	8.36	6.85
		CKKS/GL	4.10	3.96	3.83	4.00	5.76	7.75

Table 4. Two-stage StC with one final rescaling (Table 3) and with rescaling after each stage, which keeps the two stage primes as separate limbs, at 1 to 32 computation threads; the ratio rows divide the per-stage by the one-rescale latency. The starred sets have no NTT primes small enough to split their StC limb within the 128-bit budget, so their per-stage chains use two 20-bit primes and are insecure: they serve only this latency ablation.

Parameter set	Order	Latency	Computation threads					
			1	2	4	8	16	32
LL13-secure*	–	1 rescale, per ct (ms)	33.76	21.09	11.39	10.85	8.94	8.62
		2 rescales, per ct (ms)	45.36	26.82	15.83	10.38	13.04	13.24
		2 rescales / 1 rescale	1.34	1.27	1.39	0.96	1.46	1.54
n256p17*	$W \to X$	1 rescale, per slice (ms)	9.18	6.71	3.16	1.88	1.17	1.02
		2 rescales, per slice (ms)	12.89	7.47	4.35	2.25	1.48	1.26
		2 rescales / 1 rescale	1.40	1.11	1.38	1.20	1.26	1.24
	$X \to W$	1 rescale, per slice (ms)	12.32	7.84	2.99	1.77	1.07	1.07
		2 rescales, per slice (ms)	11.44	8.67	3.59	2.14	1.71	1.38
		2 rescales / 1 rescale	0.93	1.11	1.20	1.21	1.60	1.29
LL14-secure	–	1 rescale, per ct (ms)	85.64	50.54	27.63	15.20	18.30	18.98
		2 rescales, per ct (ms)	109.92	65.97	38.11	22.28	19.07	24.58
		2 rescales / 1 rescale	1.28	1.31	1.38	1.47	1.04	1.30
n512p17	$W \to X$	1 rescale, per slice (ms)	23.56	12.45	7.18	3.97	2.48	2.19
		2 rescales, per slice (ms)	27.21	14.66	7.95	4.91	2.82	2.48
		2 rescales / 1 rescale	1.15	1.18	1.11	1.24	1.14	1.13
	$X \to W$	1 rescale, per slice (ms)	24.75	12.97	7.41	4.15	2.69	2.30
		2 rescales, per slice (ms)	26.52	14.14	7.73	4.72	3.31	2.45
		2 rescales / 1 rescale	1.07	1.09	1.04	1.14	1.23	1.07
n512p17-prec	$W \to X$	1 rescale, per slice (ms)	24.86	13.38	7.25	4.28	2.45	2.10
		2 rescales, per slice (ms)	28.17	15.38	8.40	5.06	2.85	2.50
		2 rescales / 1 rescale	1.13	1.15	1.16	1.18	1.16	1.19
	$X \to W$	1 rescale, per slice (ms)	26.85	14.23	7.55	4.45	2.65	2.35
		2 rescales, per slice (ms)	29.51	15.76	8.63	5.08	3.18	2.79
		2 rescales / 1 rescale	1.10	1.11	1.14	1.14	1.20	1.19
HT15-secure	–	1 rescale, per ct (ms)	351.07	180.57	94.56	61.54	47.00	51.83
		2 rescales, per ct (ms)	403.52	220.72	121.06	78.30	59.81	70.66
		2 rescales / 1 rescale	1.15	1.22	1.28	1.27	1.27	1.36
n1024p17	$W \to X$	1 rescale, per slice (ms)	89.45	49.10	27.28	16.66	9.35	7.36
		2 rescales, per slice (ms)	88.85	48.10	28.26	17.35	9.42	7.10
		2 rescales / 1 rescale	0.99	0.98	1.04	1.04	1.01	0.96
	$X \to W$	1 rescale, per slice (ms)	85.70	45.55	24.67	15.39	8.16	6.69
		2 rescales, per slice (ms)	88.23	46.31	25.28	15.72	8.49	6.78
		2 rescales / 1 rescale	1.03	1.02	1.02	1.02	1.04	1.01

Table 5. Complex bootstrap latency at 1 to 32 computation threads, with the shared maximum and mean precision from the 32-thread runs, in bits. GL latency is given per slice (equal to one CKKS ciphertext), per matrix and per ciphertext. The CKKS/GL rows give the matched CKKS SHIP latency divided by the GL per-slice latency.

Parameter set	Latency	Computation threads						Max bits	Mean bits
		1	2	4	8	16	32		
LL13-secure	per ct (ms)	4817.80	2886.19	1833.06	883.98	477.85	360.12	4.60	6.46
n256p17	per slice (ms)	3996.72	1987.44	1049.43	604.37	305.96	185.11	3.88	6.52
	per matrix (s)	63.95	31.80	16.79	9.67	4.90	2.96		
	total (s)	1023.16	508.79	268.65	154.72	78.33	47.39		
	CKKS/GL	1.21	1.45	1.75	1.46	1.56	1.95		
LL14-secure	per ct (ms)	7620.97	4285.35	2453.54	1383.08	754.44	578.39	17.49	19.45
n512p17	per slice (ms)	6837.30	3521.46	1940.86	1146.49	593.92	400.21	15.90	18.78
	per matrix (s)	218.79	112.69	62.11	36.69	19.01	12.81		
	total (s)	3500.70	1802.99	993.72	587.00	304.09	204.91		
	CKKS/GL	1.11	1.22	1.26	1.21	1.27	1.45		
n512p17-prec	per slice (ms)	9714.73	4962.30	2694.73	1612.38	826.81	549.76	19.31	21.63
	per matrix (s)	310.87	158.79	86.23	51.60	26.46	17.59		
	total (s)	4973.94	2540.70	1379.70	825.54	423.32	281.48		
	CKKS/GL	0.78	0.86	0.91	0.86	0.91	1.05		
HT15-secure	per ct (ms)	39252.63	20327.45	11170.61	7083.28	4052.27	3276.28	18.73	20.48
n1024p17	per slice (ms)	38394.41	20284.34	10657.17	6125.13	3183.43	2546.07	18.28	21.45
	per matrix (s)	2457.24	1298.20	682.06	392.01	203.74	162.95		
	total (s)	39315.88	20771.16	10912.94	6272.13	3259.83	2607.17		
	CKKS/GL	1.02	1.00	1.05	1.16	1.27	1.29		

of each other, and alternative product trees and a 41-bit StC limb lie within 0.10 bits. The gap is set by the four-digit budget rather than by the allocation, and only the eight-digit set closes it.

Table 7 reports the per-slice errors of the same runs, measured against the inverse-Y slice messages before reassembly, next to the full-output errors. The mean error grows by 4.06, 4.51, 4.59 and 5.06 bits, close to the $\frac{1}{2}\log_2 n$ bits implied by the exact $\sqrt{n}$ RMS amplification of Section 3.1; the maximum grows by 2.20 to 4.12 bits. Reconstruction checks using the decrypted slice errors leave a residual below $4 \cdot 10^{-13}$, so reassembly adds no error of its own.

4.6 Memory Footprint and Evaluation-Key Analysis

The dominant memory terms are the StC keys and the SHIP masked-column and HMux keys; the latter two together form the hybrid blind-rotation key. The logical HMux count depends on the coarse X-axis rotation range remaining after the θ-split and on the decomposition radix $\mathcal{B} = 4$. In contrast, dnum does not change this count; it determines the number of hybrid RNS partitions stored and accumulated within each HMux branch and therefore affects key memory. For CKKS, we use the value of θ optimized in the SHIP paper. For GL, we tune θ subject to the algebraic constraint $\theta \mid n$. Its masked-column count is $\sum_e |\mathcal{C}_e|$, rather than CKKS's fixed $4\theta h$ count in these configurations: a W-major public window can cross a W-block boundary and introduce candidates for multiple W exponents, so $|\mathcal{C}_e|$ varies with e. Consequently, GL can require more masks even with a smaller θ, as in LL14 (1504 versus 1116).

Table 6. Ten representative four-digit n512p17 allocations from the parameter search, validated with complete bootstraps at 32 threads, with the default n512p17 set, the eight-digit n512p17-prec set and the CKKS reference. Q widths list q_0, the StC limb and the application limb, then the five product-tree limbs; the StC column gives the axis order and the first/second stage scale bits. Rows 1 to 7 sweep γ at a 37-bit application scale; rows 8 to 10 use 38-, 40- and 42-bit application scales.

Set	Q widths (bits)	P widths	γ	θ	StC order, bits	Max bits	Mean bits	Per slice (ms)
LL14-secure	48,37,37 | 43,43,43,43,44	48,44	2048	–	direct, 37	17.49	19.45	578.39
n512p17	44,40,37 | 43,42,44,43,45	56,36	128	8	$W \to X$ 21/19	15.90	18.78	400.21
n512p17-prec	48,47,40 | 48,47,49,49,50	52	256	8	$W \to X$ 24/23	19.31	21.63	549.76
1	43,40,37 | 43,43,44,43,45	56,36	64.00	8	$W \to X$ 21/19	16.10	18.23	405.04
2	44,40,37 | 43,42,44,43,45	56,36	80.00	32	$W \to X$ 21/19	16.12	18.63	537.10
3	44,40,37 | 43,42,44,43,45	56,36	96.00	32	$W \to X$ 21/19	16.08	18.82	557.62
4	44,40,37 | 43,42,44,43,45	50,42	107.63	8	$X \to W$ 19/21	16.09	18.83	391.19
5	44,40,37 | 43,42,44,43,45	56,36	128.00	32	$W \to X$ 21/19	15.92	18.84	565.64
6	45,40,37 | 43,42,44,43,44	55,37	160.00	32	$W \to X$ 21/19	15.66	18.59	570.33
7	45,40,37 | 43,42,44,43,44	55,37	256.00	8	$W \to X$ 21/19	15.16	17.99	413.41
8	45,40,38 | 42,42,43,43,45	55,37	103.97	32	$W \to X$ 21/19	16.08	18.88	582.34
9	47,39,40 | 40,42,44,43,44	53,38	111.43	8	$W \to X$ 20/19	16.02	18.66	411.78
10	49,40,42 | 41,41,41,42,42	51,41	128.00	8	$W \to X$ 21/19	14.51	17.09	410.49

Table 7. Per-slice and full-output precision in bits from the 32-thread complex bootstraps. Slice errors are relative to the inverse-Y slice messages; full errors are relative to the original matrices.

Parameter set	Slice max	Slice mean	Full max	Full mean
n256p17	6.08	10.58	3.88	6.52
n512p17	19.73	23.29	15.90	18.78
n512p17-prec	21.78	26.22	19.31	21.63
n1024p17	22.40	26.51	18.28	21.45

Table 8 measures and counts the StC keys and the hybrid blind-rotation keys, with the latter comprising the masked-column and HMux keys. The total also includes relinearization, conjugation, and dense-to-sparse key-switching keys. We sum the stored 64-bit RNS coefficients and report decimal gigabytes ($1\,\mathrm{GB} = 10^9$ bytes). The count columns in Table 8 report masked-column ciphertexts, logical HMux branches, and StC automorphism keys, respectively; for GL, the StC count distinguishes the small W-rotation keys from the one large conjugate transpose key.

Our GL bootstrapping reduces the number of logical automorphism keys required by StC: the X-axis uses a native GL plaintext–ciphertext matrix multiplication, and only the W-axis uses a diagonal BSGS transform. However, the large conjugate transpose key used for the X-axis contains n times as many coefficients as a small rotation key. Consequently, GL uses fewer logical StC keys but not necessarily less StC key memory.

The large conjugate transpose key is used by the X-axis inversion. We generate this key at the maximum supported multiplicative depth, so its polynomial payload covers the RNS limbs available at that depth. The n512p17 configuration supports one multiplication level, whereas n1024p17 supports eight. This larger depth coverage explains the sharp increase in GL StC key memory from n512p17 to n1024p17.

Table 8. Counts and stored polynomial sizes in decimal GB for the StC and hybrid blind-rotation keys. The StC count for GL is shown as the number of small W-rotation keys plus one large conjugate transpose key. The total includes all evaluation keys.

Parameter set	N	Scheme	θ	w	Masked column	HMux	StC keys	StC (GB)	Hybrid blind rotation (GB)	All keys (GB)
LL13-secure	2^{13}	CKKS	6	384	744	372	158	0.12	7.90	8.04
n256p17		GL	16		3288	248	$6+1$	0.41	8.56	8.98
LL14-secure	2^{14}	CKKS	9	576	1116	372	190	0.20	10.73	10.94
n512p17		GL	8		1504	372	$6+1$	1.35	11.74	13.11
n512p17-prec		GL	8		1504	372	$6+1$	1.62	17.59	19.24
HT15-secure	2^{15}	CKKS	17	1056	2108	372	318	1.00	50.64	51.71
n1024p17		GL	4		720	496	$6+1$	22.57	46.70	69.33

Table 9. Published GL comparison with Crockett *et al.* [12]; ours is n1024p17. $n/\varphi(p)$ identifies the native matrix size and count per ciphertext. Times are seconds; each slice contains 16,384 slots, and the per-matrix latency divides by the matrices per ciphertext. Our precision entries give maximum/mean bits.

Source	Threads	$n/\varphi(p)$	Q/PQ bits	Output modulus	Precision (bits)	Total/ ct (s)	Per slice (s)	Per matrix (s)
[12]	1	64/256	748/868	100	17.61	419.99	6.56	1.64
Ours	1	1024/16	682/868	440	18.28/21.45	39315.88	38.39	2457.24
Ours	32	1024/16	682/868	440	18.28/21.45	2607.17	2.55	162.95

4.7 Comparison with Concurrent GL Bootstrapping

Crockett *et al.* [12] use the GL parameter set $(n, p) = (64, 257)$, with RLWE dimension $N = 32768$, Q748/P120, sparse-secret weight 128 and output modulus $\log_2 Q' = 100$. Their polynomial EvalMod has degree 49. They report 17.61 bits of output precision without specifying whether it is based on maximum or mean error. Our n1024p17 configuration has a 440-bit output modulus, compared with their 100 bits, and achieves 18.28 maximum and 21.45 mean precision bits. Their usable application multiplication depth is not specified.

Table 9 reports their latency per ciphertext, per slice and per matrix, normalized from their batch of four ciphertexts. They report only single-thread timings.

5 Related Work

5.1 Homomorphic Matrix Computations

Prior work reduces homomorphic matrix multiplication to a small number of plaintext modular matrix multiplications. Bae *et al.* address plaintext–ciphertext multiplication, while Park extends this approach to ciphertext–ciphertext multiplication for large matrices [2, 26]. These methods benefit from linear algebra libraries such as OpenBLAS [33] but remain coefficient-oriented CKKS constructions: their matrix layouts are constrained by the underlying ring, and alternating matrix operations with slot-domain SIMD requires repeated CtS and StC conversions. GL instead provides native slot-encoded matrix arithmetic through its multivariate ring, avoiding this representation mismatch [16].

A closely related concurrent CKKS-based matrix scheme uses a *Slots-in-Coefficient* (SinC) encoding to batch many matrix-multiplication instances [9]. This construction supports small and medium matrix dimensions without tying the matrix dimension or batch size directly to the ring dimension, and it substantially improves batched matrix-multiplication performance over earlier reduction-based methods. SinC

therefore addresses an important dimensional constraint of those methods, while retaining the coefficient-oriented representation limitation described above. Our work provides bootstrapping for GL's native matrix encoding.

5.2 Bootstrapping

CKKS bootstrapping. Classical CKKS bootstrapping research primarily targets improvements in precision, latency, and multiplicative depth within the polynomial-EvalMod paradigm [4, 5, 18]. SHIP instead targets a shallower and more parallelizable regime, and its low-depth design objective is the starting point for our work [6]. Whereas SHIP operates on standard CKKS ciphertexts, our construction adapts this objective to the multivariate GL slice algebra while preserving GL's matrix-native representation.

Cheon *et al.* [11] provide a multidimensional bootstrapping construction for tensor-style CKKS packing, which the GL paper identifies as a possible starting point for GL bootstrapping [16]. That construction extends the classical polynomial-approximation regime to multidimensional CKKS and requires rotation material for its linear transforms. Our work instead targets native (X, W) GL slices and the low-depth SHIP regime. The two approaches therefore differ in both their underlying representation and their depth objective.

Batched CKKS bootstrapping. A recent line of work focuses on improving CKKS bootstrapping throughput by batching ciphertexts. In this direction, Cheon *et al.* introduce MRFHE, a CKKS-type scheme over a mixed-radix cyclotomic ring with degree $N = 2^{w_2}3^{w_3}$ [7]. Its CRT representation exposes the slots as an $n_1 \times n_2$ matrix, with $n_1 n_2 = N/2$, and factors the global DFT into radix-2 and radix-3 transforms of the form

$$\mathbf{D}_1 \mathbf{M} \mathbf{D}_2^{\mathsf{T}}.$$

For a batch of ciphertexts, MRFHE packs the inputs into two auxiliary extension-ring representations and evaluates the left and right DFT factors through reduction-based plaintext matrix multiplication. This replaces the rotation-heavy transform for each ciphertext with an $O(1)$ amortized number of key-switching operations per ciphertext. Their proof-of-concept experiments report 1.42×–1.73× throughput improvements for the complete bootstrapping pipeline with batches of 64–256 ciphertexts.

MRFHE provides a two-dimensional slot layout with independent row and column rotations and supports reduction-based batched matrix multiplication. Unlike GL, it does not provide a native matrix product on matrix-encoded ciphertexts.

Concurrent GL bootstrapping. Concurrent work by Crockett *et al.* [12] develops an alternative, classical EvalMod-based bootstrapping construction for GL. Their X-axis linear transform, like ours, uses native GL plaintext–ciphertext matrix multiplication and a conjugate transpose. For the W axis, however, they introduce an auxiliary variable V and work over

$$R_{\text{aux}} = \mathbb{Z}[I, X, W, V]/\langle I^2 + 1,\, X^n - I,\, \Phi_p(W),\, \Phi_p(V)\rangle.$$

Given slice ciphertexts $\mathsf{ct}_j = (b_j, a_j)$ modulo Q_L, they form one packed ciphertext $\mathsf{ct}^{\text{ext},V} \in (R^{\text{aux}}_{Q_L})^2$, where $R^{\text{aux}}_{Q_L} = R_{\text{aux}}/Q_L R_{\text{aux}}$:

$$\mathsf{ct}^{\text{ext},V} = \left(\sum_j b_j V^j,\, \sum_j a_j V^j \right).$$

The coefficient of V^j therefore stores the j-th slice ciphertext; because the secret $s(X, W)$ is independent of V, this packing preserves decryption. They next apply a conjugate transpose that conjugates the Gaussian coefficients, inverts the X-axis, and exchanges the inverted W- and V-axes. Because this transformation changes the secret-key representation, they key-switch the result back to the original secret $s(X, W)$. To evaluate the W-axis DFT, they encode its matrix in a polynomial with a second temporary variable U, multiply the transposed ciphertext by this polynomial, and apply Tr_U. At the slot level, this trace computes the inner-product sum over the U-index and removes U from the result; it does not remove the packing variable

V. After reversing the conjugate transpose, they extract each coefficient of V^j to recover the transformed slice ciphertexts. Their optimized ModRaise/CtS procedure uses three distinct switching keys: one small dense-to-sparse key over the (X, W) ring, one big conjugate transpose key over the full GL ring, and one big conjugate transpose key over R_{aux}. The full-ring key is applied once and the auxiliary-ring key twice, giving three big key-switching operations. This count excludes the relinearization material used by EvalMod.

For the W-axis key material, their construction requires one big switching key over R_{aux}. A polynomial over the (X, W) RLWE ring has $n\varphi(p)$ Gaussian coefficients, whereas a polynomial over R_{aux} has $n\varphi(p)^2$. Thus, for the same RNS basis and gadget decomposition, this auxiliary key contains as many coefficients as $\varphi(p)$ small RLWE-ring switching keys. Our W-axis StC instead uses a BSGS decomposition with $O(\sqrt{\varphi(p)})$ small W-rotation keys and requires no auxiliary-ring key. This is a comparison of the W-axis key sizes; both constructions use separate key material for their X-axis transforms. In their reported instance $(n, p) = (64, 257)$, R_{aux} contains $64 \cdot 256 \cdot 256$ coefficients; the paper reports that operations in this ring are approximately eight times slower than operations in its $64 \cdot 64 \cdot 256$-coefficient three-variable ring, partly because the prime-length dimensions use Rader transforms. In this work, we algorithmically optimize the Rader NTT to substantially reduce its arithmetic count and close the performance gap with the standard one-dimensional radix-2 Cooley–Tukey NTT used by CKKS, as reflected by our experiments.

The constructions target different bootstrapping regimes. Crockett *et al.* accelerate the transforms around polynomial EvalMod, which dominates their reported runtime. We instead compute decryption in the exponent and use a shallow sine approximation. Section 4.7 compares the published workloads, input/output moduli and precision with our measurements. Their reported GL dimension is 2^{15}; our smaller instances demonstrate a low-depth alternative.

6 Conclusion

We presented a low-depth bootstrapping framework that operates directly on GL ciphertexts. The construction combines a two-stage GL StC transform with a SHIP-style half-bootstrap derived for the (X, W) slice algebra, while preserving GL's native matrix representation. The construction uses X-only hidden HMux selection, algebraically reduces the number of online plaintext encodings, and accelerates the W-axis transform using a fused Rader NTT. At matched RLWE dimensions and usable application depths, with parameter sets targeting 128-bit security, we demonstrate six-level complex bootstrapping for $\log_2 N \in \{13, 14, 15\}$. Across the evaluated configurations, the faster GL axis order gives amortized per-slice StC speedups of 3.44×–4.10× at one thread and 7.75×–9.04× at 32 threads over the matched CKKS baselines. For full complex bootstrapping, the CKKS-to-GL per-slice latency ratios are 0.78–1.21 at one thread and 1.05–1.95 at 32 threads.

Directions for future research include evaluating alternative prime-length NTT implementations, such as Bluestein's algorithm, and investigating whether Winograd NTT algorithms can reduce the number of modular multiplications and improve latency. Further work includes applying GL and the proposed bootstrapping framework to private neural-network inference, including transformer models, and exploring GPU acceleration for the computational primitives that dominate GL bootstrapping and matrix multiplication.

Acknowledgments

This work has been supported by NSF Award #2239334.

References

1. Aikata Aikata and Sujoy Sinha Roy. Secure and efficient outsourced matrix multiplication with homomorphic encryption. In *Progress in Cryptology – INDOCRYPT 2024*, pages 51–74. Springer, 2024.
2. Youngjin Bae, Jung Hee Cheon, Guillaume Hanrot, Jai Hyun Park, and Damien Stehlé. Plaintext-ciphertext matrix multiplication and FHE bootstrapping: Fast and fused. In *Advances in Cryptology – CRYPTO 2024*, pages 387–421. Springer, 2024.

3. Jean-Philippe Bossuat, Rosario Cammarota, Ilaria Chillotti, Benjamin R. Curtis, Wei Dai, Huijing Gong, Erin Hales, Duhyeong Kim, Bryan Kumara, Changmin Lee, et al. Security guidelines for implementing homomorphic encryption. *IACR Communications in Cryptology*, 1(4), 2025.
4. Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In *Advances in Cryptology – EUROCRYPT 2021*, pages 587–617. Springer, 2021.
5. Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *Advances in Cryptology – EUROCRYPT 2018*, pages 360–384. Springer, 2018.
6. Jung Hee Cheon, Guillaume Hanrot, Jongmin Kim, and Damien Stehlé. SHIP: A shallow and highly parallelizable CKKS bootstrapping algorithm. In *Advances in Cryptology – EUROCRYPT 2025*, volume 15603 of *Lecture Notes in Computer Science*, pages 398–428. Springer, 2025.
7. Jung Hee Cheon, Seungwan Hong, Minsik Kang, Jonghyun Kim, Taeseong Kim, Changmin Lee, and Junho Lee. MRFHE: Mixed-radix fully homomorphic encryption with better batch bootstrapping. Cryptology ePrint Archive, Paper 2026/853, 2026.
8. Jung Hee Cheon, Minsik Kang, Taeseong Kim, Junyoung Jung, and Yongdong Yeo. Batch inference on deep convolutional neural networks with fully homomorphic encryption using channel-by-channel convolutions. *IEEE Transactions on Dependable and Secure Computing*, 22(2):1674–1685, 2025.
9. Jung Hee Cheon, Minsik Kang, and Junho Lee. Fast batch matrix multiplication in ciphertexts. In *Advances in Cryptology – CRYPTO 2026*, volume 16801 of *Lecture Notes in Computer Science*, pages 558–590. Springer, 2026.
10. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437. Springer, 2017.
11. Jung Hee Cheon, Andrey Kim, and Donggeon Yhee. Multi-dimensional packing for HEAAN for approximate matrix arithmetics. Cryptology ePrint Archive, Paper 2018/1245, 2018.
12. Eric Crockett, Craig Gentry, Hyojun Kim, Yeongmin Lee, and Yongwoo Lee. Efficient bootstrapping in fully homomorphic encryption for matrix arithmetic. In *Advances in Cryptology – CRYPTO 2026*, volume 16801 of *Lecture Notes in Computer Science*, pages 492–524. Springer, 2026.
13. Desilo Inc. The DESILO FHE library. https://fhe.desilo.dev/latest/, 2026. Accessed: July 13, 2026.
14. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, pages 169–178, 2009.
15. Craig Gentry, Shai Halevi, and Nigel P. Smart. Better bootstrapping in fully homomorphic encryption. In *Public Key Cryptography – PKC 2012*, pages 1–16. Springer, 2012.
16. Craig Gentry and Yongwoo Lee. Fully homomorphic encryption for matrix arithmetic. In *Advances in Cryptology – CRYPTO 2026*, volume 16801 of *Lecture Notes in Computer Science*, pages 363–395. Springer, 2026.
17. Kyoohyung Han, Minki Hhan, and Jung Hee Cheon. Improved homomorphic discrete Fourier transforms and FHE bootstrapping. *IEEE Access*, 7:57361–57370, 2019.
18. Kyoohyung Han and Dohyeong Ki. Better bootstrapping for approximate homomorphic encryption. In *Topics in Cryptology – CT-RSA 2020*, pages 364–390. Springer, 2020.
19. Lucjan Hanzlik, Yang Zhang, Kathrin Grosse, Ahmed Salem, Maximilian Augustin, Michael Backes, and Mario Fritz. MLCapsule: Guarded offline deployment of machine learning as a service. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 3300–3309, 2021.
20. Seonghak Kim, Minji Park, Jaehyung Kim, Taekyung Kim, and Chohong Min. EvalRound algorithm in CKKS bootstrapping. In *Advances in Cryptology – ASIACRYPT 2022*, pages 161–187. Springer, 2022.
21. Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, et al. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access*, 10:30039–30054, 2022.
22. Long Li, Jianxin Lai, Peng Yuan, Tianxiang Sui, Yan Liu, Qing Zhu, Xiaojing Zhang, Linjie Xiao, Wenguang Chen, and Jingling Xue. ANT-ACE: An FHE compiler framework for automating neural network inference. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, pages 193–208, 2025.
23. Lawrence Lim, Vikas Kalagi, Divyakant Agrawal, and Amr El Abbadi. Tricycle: Private transformer inference with tricyclic encodings. Cryptology ePrint Archive, Paper 2025/1200, 2025.
24. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010.
25. Alexander May. How to meet ternary LWE keys. In *Advances in Cryptology – CRYPTO 2021*, volume 12826 of *Lecture Notes in Computer Science*, pages 701–731. Springer, 2021.

26. Jai Hyun Park. Ciphertext-ciphertext matrix multiplication: Fast for large matrices. In *Advances in Cryptology – EUROCRYPT 2025*, pages 153–180. Springer, 2025.
27. Charles M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56(6):1107–1108, 1968.
28. Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1310–1321, 2015.
29. Zejiu Tan, Junping Wan, Zoe L. Jiang, Jingjing Fan, Man Ho Au, and Siu Ming Yiu. High-precision homomorphic modular reduction for CKKS bootstrapping. In *Information Security and Privacy – ACISP 2025*, pages 232–251. Springer, 2025.
30. Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction APIs. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 601–618. USENIX Association, 2016.
31. Runhua Xu, Nathalie Baracaldo, and James Joshi. Privacy-preserving machine learning: Methods, challenges and directions. *arXiv preprint*, (arXiv:2108.04417), 2021.
32. Jiawen Zhang, Jian Liu, Lipeng He, Xinpeng Yang, Wen-jie Lu, Yinghao Wang, Kejia Chen, Xiaoyang Hou, Kui Ren, and Xiaohu Yang. Secure transformer inference made non-interactive. Cryptology ePrint Archive, Paper 2024/136, 2024.
33. Xianyi Zhang, Qian Wang, and Yunquan Zhang. Model-driven level 3 BLAS performance optimization on Loongson 3a processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 684–691. IEEE, 2012.