Compiling Sparse Keys for Bootstrapping FHEs: Algorithms, Hardware Acceleration, and Beyond

Binwu Xiang[1], Songyu Wu[2], Baoyu Li[3], Xinwei Qiang[3], Benqiang Wei[4], and Yu Yu[1,3]

[1]East China Normal University
[2]Institute of Information Engineering, Chinese Academy of Sciences
[3]Shanghai Jiao Tong University
[4]China Telecom Quantum Information Technology Group Co., Ltd.

Abstract

Blind rotation is the dominant computational bottleneck in bootstrapping for bitwise FHE schemes such as TFHE. Existing constructions typically evaluate $O(n)$ sequential external products for an LWE secret of dimension n, incurring substantial latency and a large number of NTT/iNTT operations. In this work, we present a new framework for NTRU-based bootstrapping that reduces the sequential complexity of blind rotation for sparse binary LWE secrets. Inspired by Jain et al. (CRYPTO 2026), we use Cuckoo hashing to transform an n-dimensional binary LWE secret of Hamming weight h into extended buckets of one-hot representation. This structured representation reduces the sequential external products from $O(n)$ to $O(h)$ in blind rotation. We also design a modulus-switching method tailored to sparse secrets. We further explore an NTT-free variant that eliminates all online NTT/iNTT operations during blind rotation while supporting gate bootstrapping with lower parallel depth, offering a potentially useful building block for hardware-friendly FHE implementations. Empirically, we achieve state-of-the-art bootstrapping performance on both CPUs and GPUs. At comparable decryption failure rates and on a single CPU thread with AVX-512, our implementation executes Boolean gate, 4-bit, and 6-bit bootstrapping in 0.83, 1.75, and 2.65 ms, outperforming TFHE-rs by 3.31×, 4.18×, and 20.47×, respectively. On an RTX 4090 GPU, we attain a throughput of 154,739 gate bootstraps per second, corresponding to an amortized time of 6.46μs, and speedups of 86.7× over our CPU result and 13.6× over VeloFHE (Shen et al., TCHES 2025). As a concrete application, we develop the first NTRU-based 8-bit FHE instruction set, achieving up to over 10× speedup over Trama et al. (TCHES 2025) with over 100× smaller key size.

1 Introduction

Fully homomorphic encryption (FHE) [27] enables computation directly on encrypted data without revealing the underlying plaintexts. Since Gentry's seminal construction, more than a decade of research has substantially advanced both the theoretical foundations and practical performance of FHE. Existing FHE schemes can be broadly divided into two families. The first follows a *word-wise* paradigm, represented by BGV [13], BFV [12,24], and CKKS [18]. These schemes support homomorphic integer or approximate arithmetic and typically exploit ciphertext packing to process many plaintext values in parallel. The second follows a *bitwise* paradigm, represented by FHEW [22], TFHE [19], and their variants. In these schemes, bootstrapping is used not only to refresh ciphertexts, but also as a basic mechanism for evaluating Boolean gates and programmable lookup tables. Consequently, the performance of a bitwise FHE system is largely determined by the efficiency of its bootstrapping procedure.

Blind rotation is the dominant computational bottleneck in FHEW/TFHE-style bootstrapping. Standard constructions sequentially process the coefficients of an n-dimensional LWE secret, resulting in $O(n)$ CMux operations and external products, each involving multiple (i)NTTs. Because every update depends on the preceding accumulator, this $O(n)$ chain forms the critical path and can hardly be shortened by parallelizing the arithmetic within individual external products. Recently, NTRU-based bootstrapping [11,38,48] lowers the cost of each step by representing a ciphertext with one ring element rather than the two components of an RLWE ciphertext, thereby roughly halving the ciphertext size and reducing the polynomial-arithmetic cost of external products. However, they still require $O(n)$ sequential external products.

A complementary approach to reducing the cost of blind rotation is to encode exploitable structure directly into the secret-key distribution. Block-binary and related distributions impose a public partition on the secret and restrict the admissible coefficient patterns within each block, allowing multiple secret-dependent choices to be consolidated into fewer homomorphic operations [35]. Partial and shared-randomness secret keys similarly exploit structured correlations to improve upon noise growth, reduce evaluation-key size, or accelerate selected homomorphic primitives [9, 14, 25, 30]. Collectively, these works establish secret-key structure as a first-class de-

sign dimension for efficient bootstrapping.

Within this paradigm, sparse LWE is particularly attractive: its secret vector has small Hamming weight $h \ll n$, so only a small fraction of its coefficients contribute to secret-dependent computation. Sparse secrets have been exploited to accelerate bootstrapping in BGV/BFV [29, 39, 50] and CKKS [2, 6, 16, 17, 33]. More recently, they have also been incorporated into RLWE-based FHEW/TFHE bootstrapping, in both standard and amortized settings, to reduce the cost of blind rotation [28, 31]. Motivated by these advances, we investigate whether general sparse LWE secrets can be effectively and securely combined with the algebraic structure of NTRU to obtain analogous sparsity-driven improvements in NTRU-based bootstrapping.

1.1 Our Contributions

We present an NTRU-based bootstrapping framework that exploits general sparse LWE secret keys to reduce the number of sequential external products of blind rotation from $O(n)$ to $O(h)$. Building on this framework, we develop high-performance CPU and GPU implementations for both gate bootstrapping and high-precision programmable bootstrapping (PBS). We further introduce an NTT-free bootstrapping framework with a highly parallel structure that has the potential to facilitate hardware-friendly FHE implementations.

Probabilistic Batch Codes for NTRU blind rotation. We apply the canonically decodable probabilistic batch code (PBC) compiler of Jain et al. [31] to the NTRU/NGSW setting. Starting from a fixed-weight binary LWE secret $\mathbf{s} \in \{0,1\}^n$ of Hamming weight h, the compiler uses C public replication maps and a successful reverse-Cuckoo schedule to derive $h' = O(h)$ one-hot buckets. We design a compatible NTRU/NGSW bootstrapping procedure that merges all candidate monomial rotations in each bucket into a single NGSW selector ciphertext. Consequently, blind rotation requires one external product per bucket, reducing its sequential external-product complexity from $O(n)$ to $O(h)$ compared with conventional TFHE and NTRU-based constructions [11, 19]. In addition, we design a novel modulus-switching method tailored to sparse fixed-weight secrets, reducing rounding noise and enabling the use of more compact parameter sets.

CPU and GPU implementations. We implement our construction on CPUs. Using a single thread without AVX instructions, our Boolean gate bootstrapping takes 2.33 ms, achieving 5.88× and 4.73× speedups over TFHE in OpenFHE [40] and FINAL [11], respectively. With AVX-512 and comparable failure rates, our Boolean, 4-bit, and 6-bit instances require 0.83, 1.75, and 2.65 ms, corresponding to 3.31×, 4.18×, and 20.47× speedups over TFHE-rs [49], respectively. Multi-threading further reduces the best observed latencies of our fastest Boolean, 4-bit, and 6-bit settings to 0.16, 0.59, and 0.67 ms, respectively. We also develop a GPU implementation and evaluate it on an NVIDIA RTX 4090. Our implementation reaches 154,739 Boolean gate bootstrapping operations per second (OPS), outperforming VeloFHE [44] by 13.6×.

LUT-based NTT-free external products. We propose a lookup-table-based external product that replaces online NTT/FFT and multiplications with table lookups, negacyclic rotations, and vector additions or subtractions. Because each table entry is a fresh NGSW encryption of a digit-scaled message, decomposition digits no longer multiply the key error, reducing digit-dependent noise. This design removes the requirement for an NTT-compatible modulus and allows our sparse-secret NTRU bootstrapping framework to have lower parallel depth. Our NTT-free implementation executes Boolean gate bootstrapping in 3.47 ms using 6 CPU threads and achieves a batch-one latency of 0.47 ms on the RTX 4090. Therefore, the additive dataflow highlights the potential for hardware-friendly FPGA and ASIC acceleration without dedicated transform or modular-multiplication units.

Efficient FHE processor and Applications. Building on our blind rotation algorithm, we develop NSA, an NTRU-based 8-bit FHE instruction set that redesigns the tree-based framework of Trama et al. [45]. Our construction replaces their costly inter-level pipeline based on SampleExtract and PublicFunctionalKeyswitch with direct NTRU-domain packing, reducing the online evaluation-key footprint from 17,104 MiB in the original public artifact to 124.68 MiB. On the same CPU platform, NSA achieves 2.00×–41.98× speedups across the reported instruction classes, which cover 39 evaluated variants. For four representative encrypted programs, Maximum, Average, SquareSum, and BubbleSort, it achieves 4.22×–17.17× end-to-end speedups over encrypted arrays of five 8-bit integers. We also instantiate the AES framework of Belaid et al. [8] over NSA. Our single-threaded ten-round AES-128 evaluation takes 5.292 s for $\mathsf{DFR} < 2^{-32}$ and 7.430 s for $\mathsf{DFR} < 2^{-128}$, achieving speedups of 3.26× and 37.5×, respectively, over Hippogryph [8].

1.2 Technical Overview

Before presenting our construction, we briefly recall GINX blind rotation in FHEW/TFHE [19, 26] and the NTRU instantiation in FINAL [11]. Let $\mathbf{c} = (\mathbf{a}, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ be an LWE ciphertext under a binary secret $\mathbf{s} = (s_0, \ldots, s_{n-1}) \in \{0,1\}^n$. Define $\widetilde{a}_i = \lfloor 2Na_i/q \rceil$ and $\widetilde{b} = \lfloor 2Nb/q \rceil$ in $\mathbb{Z}_{2N}$ for illustration, and write $\widetilde{\mathbf{a}} = (\widetilde{a}_0, \ldots, \widetilde{a}_{n-1})$. Refer to Section 5.1 for improved modulus switching method. Given a test polynomial $r(X) \in R_Q$, blind rotation computes $\mathsf{NTRU}_f(X^{-\widetilde{b}+\langle \widetilde{\mathbf{a}}, \mathbf{s}\rangle} r(X))$.

For a binary secret, the target monomial factors as

$$X^{-\widetilde{b}+\langle \widetilde{\mathbf{a}}, \mathbf{s}\rangle} r(X) = X^{-\widetilde{b}} r(X) \prod_{i=0}^{n-1} \left(1 + s_i (X^{\widetilde{a}_i} - 1)\right).$$

For $1 \le i < n$, the bootstrapping key contains $\mathsf{BRK}_i = \mathsf{NGSW}_f(s_i)$. From the public rotation $\widetilde{a}_i$, the evaluator forms the GINX-style CMux selector

$$\widetilde{\mathsf{BRK}}_i = \mathfrak{g} + (X^{\widetilde{a}_i} - 1)\mathsf{BRK}_i = \mathsf{NGSW}_f\left(X^{\widetilde{a}_i s_i}\right),$$

where $\mathfrak{g}$ is the gadget vector. Starting from the public ring element $\mathsf{ACC}^{(0)} = X^{-\widetilde{b}} r(X)$, the first CMux selector follows the KDM-lifting idea in Xiang et al. [48]: set $\mathsf{BRK}_0 = \mathsf{NGSW}_f(s_0/f)$, instead of $\mathsf{NGSW}_f(s_0)$. Given an additional $\mathsf{CK} = \mathsf{NGSW}_f(1/f)$, we form

$$\widetilde{\mathsf{BRK}}_0 = \mathsf{CK} + (X^{\widetilde{a}_0} - 1)\mathsf{BRK}_0 = \mathsf{NGSW}_f\left(X^{\widetilde{a}_0 s_0}/f\right).$$

Then, for every $i \in [n]$, we have

$$\begin{aligned}\mathsf{ACC}^{(i+1)} &= \widetilde{\mathsf{BRK}}_i \odot \mathsf{ACC}^{(i)} \\ &= \mathsf{NTRU}_f\left(r(X) X^{-\widetilde{b} + \sum_{k=0}^{i} \widetilde{a}_k s_k}\right).\end{aligned}$$

Classical binary-key blind rotation therefore requires n sequential external products.

The block-LWE acceleration [35] groups the secret into public one-hot blocks. We partition the indices $[n]$ into several blocks $\{\mathcal{C}_j\}$. Assume the secret key $\mathbf{s}$ satisfies $\sum_{i \in \mathcal{C}_j} s_i = 1$, i.e., exactly one entry indexed by $\mathcal{C}_j$ equals 1, and the rest are 0. Then, the selectors in block $\mathcal{C}_j$ can be merged linearly as

$$\widetilde{\mathsf{BRK}}_j = \sum_{i \in \mathcal{C}_j} X^{\widetilde{a}_i} \mathsf{BRK}_i = \mathsf{NGSW}_f\left(X^{\sum_{i \in \mathcal{C}_j} \widetilde{a}_i s_i}\right),$$

which is the one-hot multi-input selector view of Joye and Paillier [32]. Blind rotation then uses one external product per block. However, the public one-hot constraint changes the secret distribution. Because concrete security must account for this extra structure, existing designs keep each block size conservative (typically 2 or 3) and obtain only a constant-factor reduction in the number of external products.

To harness the efficiency gains of sparsity and block-wise evaluation while avoiding the substantial security degradation caused by a direct block partition, we follow Jain et al. [31]: the PBC compiles the support of a general sparse secret into one-hot buckets. We instantiate this compiler with NTRU/NGSW ciphertexts and use the resulting bucket representation for NTRU blind rotation.

From General Sparse Keys to Bucket Blocks. For a fixed-weight secret $\mathbf{s}$, let $I_{\mathbf{s}} = \mathrm{supp}(\mathbf{s})$ be the indices with nonzero entries and $h = \mathsf{HW}(\mathbf{s}) = \#I_{\mathbf{s}}$. In the reverse-Cuckoo realization of Angel et al. [5], C public hash functions $\eta_1, \ldots, \eta_C : [n] \to [h']$ give every coordinate C candidate buckets, where $h' = O(h)$. Scheduling finds an injection

$$\tau : I_{\mathbf{s}} \hookrightarrow [h'], \quad \tau(i) \in \{\eta_1(i), \ldots, \eta_C(i)\}, \quad \forall i \in I_{\mathbf{s}},$$

such that no two nonzero secret coordinates are assigned to the same bucket. If scheduling fails, we resample the hash functions and layout for the same LWE secret.

Let $\pi(j,u)$ denote the original coordinate copied to slot $u \in [m_j]$ of bucket j, and set $\widetilde{a}'_{j,u} = \widetilde{a}_{\pi(j,u)}$. A successful schedule selects one copy of every coordinate in $I_{\mathbf{s}}$ and defines selector bits $s'_{j,u}$. Adding the dummy selector $s'_{j,\perp} = 1 - \sum_u s'_{j,u}$ gives

$$s'_{j,\perp} + \sum_{u=0}^{m_j - 1} s'_{j,u} = 1, \quad \sum_{j=0}^{h'-1} \sum_{u=0}^{m_j - 1} \widetilde{a}'_{j,u} s'_{j,u} = \langle \widetilde{\mathbf{a}}, \mathbf{s} \rangle.$$

Thus the compiled key is one-hot in every bucket and preserves the LWE phase. Define its selected monomial by

$$\rho_j(X) = s'_{j,\perp} + \sum_{u=0}^{m_j - 1} s'_{j,u} X^{\widetilde{a}'_{j,u}} = X^{\sum_u \widetilde{a}'_{j,u} s'_{j,u}}.$$

It follows that $\prod_{j=0}^{h'-1} \rho_j(X) = X^{\langle \widetilde{\mathbf{a}}, \mathbf{s} \rangle}$. Therefore, via PBC, we transform a length-n sparse secret $\mathbf{s}$ into a length-$(Cn + h')$ secret $\mathbf{s}'$ with h' blocks. For every bucket $j \ge 1$, the key contains $\mathsf{BRK}_{j,u} = \mathsf{NGSW}_f(s'_{j,u})$ for $u \in [m_j] \cup \{\perp\}$. Given the public LWE mask, the evaluator merges the slot keys into

$$\widetilde{\mathsf{BRK}}_j = \mathsf{BRK}_{j,\perp} + \sum_{u=0}^{m_j - 1} X^{\widetilde{a}'_{j,u}} \mathsf{BRK}_{j,u} = \mathsf{NGSW}_f(\rho_j(X)).$$

The merges are independent across buckets. Directly adapting the RLWE-PBC technique of Jain et al. [31] to scalar NTRU ciphertexts, however, is not straightforward. In their RLWE setting, a public test polynomial can be used directly as a trivial ciphertext. This is not possible in the NTRU setting, because the corresponding encoding involves the secret-dependent factor f^{-1}. We therefore define the first-bucket evaluation keys as $\mathsf{BRK}_{0,u} = \mathsf{NGSW}_f\left(s'_{0,u} f^{-1}\right), u \in [m_0] \cup \{\perp\}$. Because the encrypted message depends on the secret key f, our construction relies on the KDM security of the underlying NTRU encryption, as in [11, 38, 48]. To the best of our knowledge, constructing an NTRU-based FHE scheme without such an assumption remains an open problem. Consequently, for any polynomial $r(X)$, its external product with $\mathsf{BRK}_{0,u}$ constitutes a valid scalar NTRU encryption of $s'_{0,u} r(X)$. For $j \in [h']$, we compute

$$\begin{aligned}\mathsf{ACC}^{(j+1)} &= \widetilde{\mathsf{BRK}}_j \odot \mathsf{ACC}^{(j)} \\ &= \mathsf{NTRU}_f\left(X^{-\widetilde{b}} r(X) \prod_{v=0}^{j} \rho_v(X)\right).\end{aligned}$$

Consequently, $\mathsf{ACC}^{(h')} = \mathsf{NTRU}_f(X^{-\widetilde{b} + \langle \widetilde{\mathbf{a}}, \mathbf{s} \rangle} r(X))$. The compiler reduces the chain from n coordinate external products to $h' = O(h)$ bucket external products. Angel et al. [5] experimentally evaluate $C = 3$ and $h' = \lceil 1.5h \rceil$ with negligible failure probability, whereas we follow Jain et al. [31] and use $C = 3$ and $h' = h + 3$ for efficiency, with sufficient scheduling success probability for our instantiations. A detailed discussion on PBC parameters is included in Appendix A.

NTT-Free Bucket Update. The standard NGSW external product is normally evaluated with NTT/FFT, whose transform-friendly arithmetic and staged butterfly dataflow impose nontrivial requirements on both software and hardware. Inspired by the idea of AP accumulator [3, 22], we explore a lookup-based external product that requires only vector additions and subtractions and admits rich parallel structure. Let $c = \mathsf{NTRU}_f(\mu)$ and $\mathbf{C} = \mathsf{NGSW}_f(\nu)$. Write $\mathfrak{g}_\beta^{-1}(c) = (c_0, \ldots, c_{\ell-1})$, where $c_t(X) = \sum_k c_{t,k} X^k$. Expanding $\mathbf{C} \odot c$ and exchanging the order of summation gives

$$\mathbf{C} \odot c = \sum_{t=0}^{\ell-1} \sum_{k=0}^{N-1} (c_{t,k} X^k) C_t = \sum_{k=0}^{N-1} X^k \left(\sum_{t=0}^{\ell-1} c_{t,k} C_t \right).$$

Provided a lookup table of NGSW ciphertexts $\mathsf{T}(\nu)$, where $\mathsf{T}_\xi(\nu) = \mathsf{NGSW}_f(\xi\nu)$ for $1 \le \xi \le \beta/2$, and $\mathsf{T}_{-\xi}(\nu) = -\mathsf{T}_\xi(\nu)$, $\mathsf{T}_0(\nu) = 0$, we define

$$\mathsf{T}(\nu) \odot_{\mathsf{lut}} c = \sum_{k=0}^{N-1} X^k \left(\sum_{t=0}^{\ell-1} \left(\mathsf{T}_{c_{t,k}}(\nu) \right)_t \right) = \mathsf{NGSW}(\nu \cdot \mu).$$

Although the general complexity is increased without NTT/FFT, this lookup-based external product is computed solely through vector negacyclic shifts, additions and subtractions, and thus admits low parallel depth. Meanwhile, it avoids the gadget-base noise variance amplification of $\beta^2/12$, enabling more compact parameters.

We apply this lookup-based external product to the PBC bucket selectors and blind rotation. The resulting online ciphertext path is purely additive after gadget decomposition, with no NTT/FFT or general polynomial multiplication. We study this design as a parallel-friendly exploration and hardware design point rather than assuming that digit indexing must outperform a highly optimized NTT backend. In CPUs and GPUs, its concrete performance remains governed by table storage and memory bandwidth.

2 Preliminaries

This section fixes notation and recalls the necessary background, including LWE, approximate gadget decomposition, NTRU/NGSW, and PBC scheduling.

2.1 Notation

We write $\mathbb{Z}$ for the set of integers, and $\mathbb{Z}_p$ for integers modulo p. We use $[a]_p$ for the unique integer in $[-p/2, p/2)$ congruent to a modulo p, and $\lfloor a \rfloor$, $\lceil a \rceil$, $\lfloor a \rceil$ for the floor, ceiling, and the nearest integer to a. For any positive integer d, write $[d] := \{0, 1, \ldots, d-1\}$. The cardinality of a finite set $\mathcal{S}$ is denoted by $\#\mathcal{S}$. All logarithms are base 2.

Vectors are written in bold lowercase. For a vector $\mathbf{u}$, we write $\|\mathbf{u}\|_2$ and $\|\mathbf{u}\|_\infty$ for its ℓ_2 and ℓ_∞ norms, respectively. We write $\mathrm{supp}(\mathbf{u})$ for the support (indices of nonzero entries) of $\mathbf{u}$, and its Hamming weight is $\mathsf{HW}(\mathbf{u}) = \#\mathrm{supp}(\mathbf{u})$.

Let N be a power of two. We work over the $2N$-th cyclotomic ring $R := \mathbb{Z}[X]/(X^N + 1)$ and its modular quotient $R_p := R/pR \cong \mathbb{Z}_p[X]/(X^N + 1)$ for any positive integer p. Throughout the paper, q denotes the LWE modulus and Q denotes the NTRU accumulator modulus. We identify a polynomial with its coefficient vector when applying norms and Hamming weight.

For a finite set $\mathcal{S}$, the notation $x \xleftarrow{\$} \mathcal{S}$ means that x is sampled uniformly from $\mathcal{S}$. For a distribution D, we write $x \leftarrow D$ for sampling according to D, and denote the uniform distribution over $\mathcal{S}$ by $\mathcal{U}(\mathcal{S})$.

2.2 Gaussian Errors and Noise Model

The discrete Gaussian distribution $D_{\mathbb{Z},\tau,c}$ with width $\tau > 0$ and center $c \in \mathbb{R}$ assigns to each $z \in \mathbb{Z}$ probability proportional to $\exp(-\pi(z-c)^2/\tau^2)$. For noise accounting, we parameterize errors by their coefficient standard deviation: $e \leftarrow \mathrm{SG}(\sigma) = D_{\mathbb{Z},\sqrt{2\pi}\sigma,0}$ denotes a centered discrete Gaussian with variance σ^2, and $\mathrm{SG}(\sigma)^N$ denotes independent coefficient-wise sampling of a vector or polynomial.

Noise model. Throughout the paper, we adopt the standard *Independent Gaussian Heuristic* used in FHE noise analyses. Namely, after each homomorphic operation, the coefficients of every error polynomial are treated as independent centered Gaussian variables with the tracked coefficient variance. Under this heuristic, a linear combination $\sum_i u_i e_i$ has estimated variance $\sum_i u_i^2 \sigma_i^2$ when each e_i has variance σ_i^2, and independent contributions are combined by adding their variances.

2.3 The LWE Problem and LWE Ciphertexts

The *Learning with Errors (LWE)* problem, introduced by Regev [43], asks to distinguish samples of the form $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ from uniformly random pairs in $\mathbb{Z}_q^{n+1}$, where $\mathbf{a} \xleftarrow{\$} \mathbb{Z}_q^n$, $\mathbf{s} \in \mathbb{Z}^n$ is the secret, and e is sampled from a narrow error distribution over $\mathbb{Z}$. The decisional LWE assumption states that this problem is computationally hard for appropriate parameters. An LWE ciphertext of $m \in \mathbb{Z}_t$ under $\mathbf{s}$ is

$$\mathsf{LWE}_{\mathbf{s}}(m, \sigma) := (\mathbf{a}, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q,$$

where $\mathbf{a} \xleftarrow{\$} \mathbb{Z}_q^n$ and $b = \langle \mathbf{a}, \mathbf{s} \rangle + e + \Delta m \pmod q$, with $e \leftarrow \mathrm{SG}(\sigma)$ and $\Delta = \lfloor q/t \rceil$. Decryption computes $\lfloor [b - \langle \mathbf{a}, \mathbf{s} \rangle]_q / \Delta \rceil \bmod t$ and recovers m when the error is sufficiently small relative to Δ.

In practice, binary and sparse-binary secrets are standard variants that enable specific accelerations. This work samples $\mathbf{s}$ from all $\{0,1\}^n$ vectors with fixed Hamming weight h.

2.4 Approximate Gadget Decomposition

Let $\beta \geq 2$ and $1 \leq \ell \leq \lceil \log_\beta Q \rceil$ be a decomposition base and length for a modulus Q. Define the gadget vector

$$\mathfrak{g} := (\mathfrak{g}_0, \mathfrak{g}_1, \ldots, \mathfrak{g}_{\ell-1}), \quad \mathfrak{g}_i = \lfloor Q/\beta^{\ell-i} \rceil.$$

The *signed base-β approximate decomposition* of an integer $a \in \mathbb{Z}_Q$ is a vector $\mathfrak{g}_\beta^{-1}(a) = (a_0, a_1, \ldots, a_{\ell-1}) \in \mathbb{Z}^\ell$ such that $a = \sum_{i=0}^{\ell-1} a_i \cdot \mathfrak{g}_i + \varepsilon_\mathfrak{g}$, where each digit $a_i \in (-\beta/2, \beta/2]$ and the approximate error $|\varepsilon_\mathfrak{g}| \leq \lceil Q/(2\beta^\ell) \rceil$. Due to the (pseudo)randomness of our cryptosystems, we assume the output digits and the approximate error from the decompositions are *uniformly distributed*. Therefore, the estimated variances of a digit and an error are $\beta^2/12$ and $\sigma_\mathfrak{g}^2 \lesssim Q^2/(12\beta^{2\ell})$, respectively. The decomposition is applied entry/coefficient-wise to vectors/polynomials.

2.5 NTRU-Based Encryption Schemes

We recall the NTRU-based primitives in prior NTRU-FHE constructions [11,38,48]: scalar NTRU ciphertexts, vectorized NTRU-GSW (NGSW) ciphertexts, and the external product.

NTRU Ciphertexts. Let $f \in R_Q$ be an invertible small secret typically from $\mathrm{SG}(\sigma)$. An NTRU ciphertext of $\mu \in R_Q$ is

$$c = \mathsf{NTRU}_f(\mu, \sigma) := (e + \mu) f^{-1} \in R_Q, \quad e \leftarrow \mathrm{SG}(\sigma)^N.$$

The decisional NTRU assumption [34] states that, for independently sampled small f and g with invertible f, the quotient gf^{-1} is indistinguishable from $\mathcal{U}(R_Q)$.

NGSW Ciphertexts. For the gadget parameters (β, ℓ) and vector $\mathfrak{g}$ of Section 2.4, an NGSW ciphertext of $\nu \in R_Q$ is

$$\mathbf{C} = \mathsf{NGSW}_f(\nu, \sigma) := \{e_t f^{-1} + \nu g_t\}_{t \in [\ell]} \in R_Q^\ell.$$

Here $e_t \leftarrow \mathrm{SG}(\sigma)^N$. We write the t-th component as C_t.

External Product. For $c = \mathsf{NTRU}_f(\mu, \sigma_\mu)$ and $\mathbf{C} = \mathsf{NGSW}_f(\nu, \sigma_\nu)$, let $\mathfrak{g}_\beta^{-1}(c) = (c_0, \ldots, c_{\ell-1})$, and define

$$\mathbf{C} \odot c := \langle \mathfrak{g}_\beta^{-1}(c), \mathbf{C} \rangle = \sum_{t=0}^{\ell-1} c_t C_t,$$

resulting in $\mathsf{NTRU}_f(\nu \cdot \mu, \sigma_\odot)$, an NTRU ciphertext of $\nu \cdot \mu$ with estimated noise variance

$$\sigma_\odot^2 \lesssim \|\nu\|_2^2 \sigma_\mu^2 + \frac{1}{12} N \ell \beta^2 \sigma_\nu^2 + \|\nu f\|_2^2 \sigma_\mathfrak{g}^2.$$

The same operation lifts a public ring element into NTRU form, with a modification to the NGSW encryption [48]. For $\mu \in R_Q$, let $\mathfrak{g}_\beta^{-1}(\mu) = (\mu_0, \ldots, \mu_{\ell-1})$ and $\mathbf{C} = \mathsf{NGSW}_f(\nu/f, \sigma_\nu)$. Then

$$\mathbf{C} \odot \mu = \sum_{t=0}^{\ell-1} \mu_t C_t = \mathsf{NTRU}_f(\nu \cdot \mu, \sigma_{\mathsf{lift}}).$$

Its error is $\sum_t \mu_t e_t - \nu \varepsilon_\mathfrak{g}$, with estimated noise variance

$$\sigma_{\mathsf{lift}}^2 \lesssim \frac{1}{12} N \ell \beta^2 \sigma_\nu^2 + \|\nu\|_2^2 \sigma_\mathfrak{g}^2.$$

Remark (KDM/circular security). In both $\mathsf{NTRU}_f(\cdot)$ and $\mathsf{NGSW}_f(\cdot/f)$, division by f applies to both the error and the message. When these ciphertexts are used homomorphically with key-dependent messages, security implicitly relies on circular-secure, or more generally KDM-secure, hardness of the underlying NTRU instance. This is standard in NTRU-based FHE constructions [11,38,48].

2.6 PBC and Cuckoo Scheduling

Canonical decoding. Following Angel et al. [5] and the canonically decodable formulation used by Jain et al. [31], an $(n, C, h, h', p_{\mathsf{succ}})$-PBC encodes n entries into h' buckets $\mathcal{C}_0, \ldots, \mathcal{C}_{h'-1}$ satisfying $m_j = \#\mathcal{C}_j$ and $\sum_{j=0}^{h'-1} m_j = Cn$, where each input entry is included in C buckets.

We write $\pi(j, u) \in [n]$ for the source position copied to slot $u \in [m_j]$ of bucket j. For a query $I \subseteq [n]$ with $\#I = h$, the scheduling returns either fail or a vector $\mathbf{u}^\star = (u_0^\star, \ldots, u_{h'-1}^\star)$, where $u_j^\star \in [m_j] \cup \{\perp\}$. Canonical decoding requires

$$\{\pi(j, u_j^\star) : u_j^\star \neq \perp\} = I, \quad \#\{j : u_j^\star \neq \perp\} = h.$$

Thus every queried position is read from exactly one bucket, no bucket supplies more than one entry, and the non-dummy outputs are a permutation of the query. For every fixed query I, the probability of scheduling success is bounded by

$$\Pr[\mathsf{Schedule}(I) = \mathsf{succ}] \geq p_{\mathsf{succ}},$$

where the probability is over the scheduling randomness.

Reverse-Cuckoo construction. We recall the Cuckoo PBC construction of Angel et al. [5]. The replication construction samples public hash functions $\eta_1, \ldots, \eta_C : [n] \to [h']$ and places one copy (i, r) in bucket $\eta_r(i)$ for every $(i, r) \in [n] \times [C]$, such that $\eta_r(i) \neq \eta_{r'}(i)$ for every i and $r \neq r'$. After fixing a public ordering within each bucket,

$$\mathcal{C}_j = \{(i, r) \in [n] \times [C] : \eta_r(i) = j\}, \quad \sum_{j=0}^{h'-1} m_j = Cn.$$

For a query I, scheduling succeeds if there exists an injection

$$\tau : I \hookrightarrow [h'], \quad \tau(i) \in \{\eta_1(i), \ldots, \eta_C(i)\}, \quad \forall i \in I.$$

Equivalently, define the candidate graph

$$\Gamma_\eta(I) = \left(I, [h']; \{(i, \eta_r(i)) : i \in I,\ r \in [C]\}\right),$$

then the schedule exists if this graph has a matching that covers I. We view PBC/Cuckoo construction as a transform from a general sparse length-n vector $\mathbf{s}$ with $h = \mathsf{HW}(\mathbf{s})$, to a

length-$(Cn+h')$ vector $\mathbf{s}'$ with h' bucket blocks, where each block has at most one nonzero entry.

Concrete regimes. After sampling a sparse secret key $\mathbf{s}$ with $\mathsf{HW}(\mathbf{s}) = h$, we independently sample a layout with Cuckoo hashing to "throw n indices into h' buckets, where each index is in C buckets". Then, we try to find a schedule. If failed, the hash functions and hence the layout are therefore rejected and resampled. Note that the publication of the accepted layout brings a $\log(1/p_{\mathsf{succ}})$-bit entropy loss to the secret space, thus leading to at most such security loss.

Angel et al. [5] experimentally evaluate $C = 3$ and $h' = \lceil 1.5h \rceil$, giving $Cn = 3n$ encoded slots. They estimate a failure probability of about 2^{-40} for $h \approx 200$ and closer to 2^{-20} for smaller queries. For simplicity of exposition and implementation, we directly follow Jain et al. [31] and use $C = 3$ and $h' = h + 3$, with sufficient success probability for our constructions. See Appendix A for further discussion on PBC parameters and their effect on security and efficiency.

3 PBC-Compiled NTRU Blind Rotation

In this section, we instantiate the PBC-compiled NTRU blind rotation outlined in Section 1.2.

3.1 PBC Compilation and Key Generation

Let $\mathbf{s} \in \{0,1\}^n$ be a binary sparse secret with support $I_{\mathbf{s}} = \mathrm{supp}(\mathbf{s})$ and Hamming weight $h = \mathsf{HW}(\mathbf{s})$. The offline procedure samples a public PBC replication layout and runs $\mathsf{Schedule}(I_{\mathbf{s}})$, resampling the layout whenever scheduling returns fail, according to Section 2.6. For a successful output $\mathbf{u}^\star = (u_0^\star, \ldots, u_{h'-1}^\star)$, define the compiled selector bits by

$$s'_{j,u} = \mathbf{1}[u_j^\star = u], \quad s'_{j,\perp} = 1 - \sum_{u=0}^{m_j-1} s'_{j,u} = \mathbf{1}[u_j^\star = \perp].$$

For any vector $\widetilde{\mathbf{a}} \in \mathbb{Z}_{2N}^n$, define $\widetilde{a}'_{j,u} = \widetilde{a}_{\pi(j,u)}$ and $\rho_j(X) = s'_{j,\perp} + \sum_u s'_{j,u} X^{\widetilde{a}'_{j,u}}$. Lemma 1 records the correctness of rotating in the exponent by one merged key per bucket, as a direct corollary of the properties of PBC scheduling.

Lemma 1 (Correctness of PBC compilation)**.** *For every successfully scheduled support, we have*

$$\sum_{j=0}^{h'-1} \sum_{u=0}^{m_j-1} \widetilde{a}'_{j,u} s'_{j,u} = \langle \widetilde{\mathbf{a}}, \mathbf{s} \rangle \in \mathbb{Z}_{2N}.$$

Moreover,

$$\rho_j(X) = X^{\sum_u \widetilde{a}'_{j,u} s'_{j,u}}, \quad \prod_{j=0}^{h'-1} \rho_j(X) = X^{\langle \widetilde{\mathbf{a}}, \mathbf{s} \rangle}.$$

The offline bootstrapping key encrypts these selectors. For each bucket $1 \le j < h'$ and $u \in [m_j] \cup \{\perp\}$, generate $\mathsf{BRK}_{j,u} \leftarrow \mathsf{NGSW}_f(s'_{j,u}, \sigma)$. For bucket $j = 0$ only, following the treatment of Xiang et al. [48], we set

$$\mathsf{BRK}_{0,u} = \mathsf{NGSW}_f(s'_{0,u}/f, \sigma)$$

to accommodate the public initialization of the accumulator.

3.2 Bucket-Key Merging

Given an input LWE ciphertext $\mathbf{c} = (\mathbf{a}, b)$, the online evaluator sets $\widetilde{a}_i = \lfloor 2Na_i/q \rceil$ and $\widetilde{a}'_{j,u} = \widetilde{a}_{\pi(j,u)}$, and writes $\widetilde{\mathbf{a}} = (\widetilde{a}_0, \ldots, \widetilde{a}_{n-1})$. For each bucket $j \in [h']$, it merges the NGSW selectors online:

$$\widetilde{\mathsf{BRK}}_j = \mathsf{BRK}_{j,\perp} + \sum_{u=0}^{m_j-1} X^{\widetilde{a}'_{j,u}} \mathsf{BRK}_{j,u}. \tag{1}$$

Each merged key has estimated variance $\widetilde{\sigma}^2_{\mathsf{BRK},j} \le (m_j+1)\sigma^2$, and encrypts $\rho_j(X)$ for $j \ge 1$ and $\rho_0(X)/f$ for $j = 0$. All h' merges depend on $\mathbf{a}$ but not on one another, so they can be evaluated in parallel before the sequential accumulator chain.

Equation (1) suggests the same cost of addition regardless of key form. For bucket j, merging the m_j real slots requires ℓm_j length-N vector additions. Since $\sum_j m_j = Cn$, all bucket merges require $Cn\ell$ vector additions in $\mathbb{Z}_Q^N$. For further use, we discuss three key preprocessing strategies, which differ only in the overhead for producing the shifted terms and preparing the merged keys for the subsequent external products.

Coefficient-domain keys. We first consider storing BRK in coefficient form. In this representation, multiplication by X^k is a negacyclic rotation with sign changes and no modular multiplication. Each merged key must subsequently undergo ℓ forward NTTs, preparing for external products.

NTT-domain keys. Alternatively, if we store BRK in NTT form, multiplication by X^k becomes a Hadamard product with each monomial X^k precomputed in NTT form. This adds $m_j\ell$ Hadamard products in $\mathbb{Z}_Q^N$ per bucket, or $Cn\ell$ summing over all buckets, but requires no key-domain NTT after merging.

Precomputed NTT-domain monomial shifts. Jain et al. [31, Appendix B.1] eliminate the Hadamard products by precomputing $X^k \mathsf{BRK}_{j,u}$ for every $k \in \mathbb{Z}_{2N}$ in NTT form. The online merge then incurs no arithmetic beyond the common additions, at the cost of a $2N$-fold increase in bootstrapping-key size. Equivalently, since $X^{k+N} = -X^k$ in R_Q, storing the N shifts for $k \in [N]$ reduces the expansion to a factor of N while retaining the same online arithmetic. The resulting bottleneck is primarily key storage and memory bandwidth.

A detailed cost analysis of the three key-preprocessing strategies and the subsequent external products in blind rotation is provided in Section 3.3 and summarized in Table 1. A concrete performance comparison is provided in Appendix B.

3.3 Blind Rotation

For an LWE ciphertext $\mathbf{c} = (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$ under $\mathbf{s}$, and a test polynomial $r(X) \in R_Q$, we aim to compute an NTRU ci-

phertext encrypting $r(X)X^{-\widetilde{b}+\langle\widetilde{\mathbf{a}},\mathbf{s}\rangle}$, where $\widetilde{\mathbf{a}} = \lfloor 2N\mathbf{a}/q \rceil$ and $\widetilde{b} = \lfloor 2Nb/q \rceil$ for illustration. Refer to Section 5.1 for improved modulus switching method. With all the ingredients in place, we present the PBC-based standard blind rotation procedure in Algorithm 1. We defer its correctness, noise analysis, and computational complexity to Theorem 1.

Algorithm 1: PBC-NTRU Blind Rotation

Input: An LWE ciphertext $\mathbf{c} = (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$ under $\mathbf{s}$; a test polynomial $r(X)$.

Auxiliaries : PBC layout and BRK from Section 3.1.

Output: $\mathsf{NTRU}_f(X^{-\widetilde{b}+\langle\widetilde{\mathbf{a}},\mathbf{s}\rangle} r(X), \sigma_{\mathsf{br}})$.

$(\widetilde{\mathbf{a}}, \widetilde{b}) \leftarrow \lfloor 2N\mathbf{c}/q \rceil$;

Set each $\widetilde{a}'_{j,u}$ from $\widetilde{\mathbf{a}}$ with the PBC layout;

Form each $\widetilde{\mathsf{BRK}}_j$ with Equation (1) in parallel;

$\mathsf{ACC}^{(0)} \leftarrow X^{-\widetilde{b}} r(X)$;

for $j = 0$ **to** $h' - 1$ **do**

  $\mathsf{ACC}^{(j+1)} \leftarrow \widetilde{\mathsf{BRK}}_j \odot \mathsf{ACC}^{(j)}$;

return $\mathsf{ACC}^{(h')}$;

Theorem 1. *Algorithm 1 correctly outputs* $\mathsf{ACC}^{(h')} = \mathsf{NTRU}_f\left(X^{-\widetilde{b}+\langle\widetilde{\mathbf{a}},\mathbf{s}\rangle} r(X), \sigma_{\mathsf{br}}\right)$, *where*

$$\sigma_{\mathsf{br}}^2 \lesssim \frac{1}{12}(Cn + h')N\ell\beta^2\sigma^2 + \left(1 + (h'-1)\|f\|_2^2\right)\sigma_{\mathfrak{g}}^2.$$

Moreover, Algorithm 1 performs h' standard external products. Table 1 summarizes the preprocessing-dependent key-merging arithmetic and the complete transform count.

Proof. The correctness follows from Lemma 1 and the properties of the NTRU external product. For the noise, the first update and every later update satisfy, respectively,

$$\sigma_1^2 \lesssim \frac{N\ell\beta^2}{12}(m_0 + 1)\sigma^2 + \sigma_{\mathfrak{g}}^2,$$
$$\sigma_{j+1}^2 \lesssim \sigma_j^2 + \frac{N\ell\beta^2}{12}(m_j + 1)\sigma^2 + \|f\|_2^2\sigma_{\mathfrak{g}}^2, \quad j \geq 1.$$

Iterating with $\sum_j m_j = Cn$ yields the stated bound. □

Computation and parallelism. Table 1 reports the cost of Algorithm 1 under different strategies in Section 3.2. Add and Mul refer to the number of vector additions/subtractions and Hadamard products in $\mathbb{Z}_Q^N$, respectively, *excluding* the operations inside complete length-N NTT procedure; NTT refers to the number of NTT/iNTT operations in both key-merging and external products. Either way, the online asymptotic complexity of Algorithm 1 is $O(C\ell nN + h'\ell N \log N)$ $\mathbb{Z}_Q$ operations, given that vector additions/subtractions and Hadamard products in $\mathbb{Z}_Q^N$ are both $O(N)$ operations, and NTT $O(N \log N)$.

Remark. For traditional non-sparse blind rotations, the online complexity is usually $O(\ell nN \log N)$ $\mathbb{Z}_Q$ operations, while other additions are often ignored since the $O(\ell n)$ NTTs dominate the cost. In contrast, the PBC-compiled blind rotation reduces the external product count from n to $h' = O(h)$, while the vector additions remain $O(\ell n)$. For a sufficiently sparse secret, the additions are no longer negligible, which is why we explicitly report them in Table 1.

Table 1: Main arithmetic in blind rotation under three preprocessing strategies.

Key preprocessing	Add	Mul	NTT
Coefficient domain	$Cn\ell$	0	$h'(2\ell+1)$
NTT domain	$Cn\ell$	$Cn\ell$	$h'(\ell+1)$
Precomputed shifts	$Cn\ell$	0	$h'(\ell+1)$

All bucket-key merges are independent, and therefore have parallel depth $\lceil \log m_{\max} \rceil$, where $m_{\max} = \max_j m_j = O(Cn/h')$; NTT-domain merging adds one Hadamard multiplication layer, while coefficient-domain keys add $\log N$ layers for merged-key NTTs. The h' external products remain sequential, while the ℓ forward NTTs and coefficient-wise operations within each external product are parallelizable.

Counting each radix-2 butterfly stage as one depth level, precomputed shifts give total parallel depth $\lceil \log m_{\max} \rceil + h'(2\log N + \lceil \log \ell \rceil + 1)$, while NTT-domain keys add 1 layer for extra Hadamard products, and coefficient-domain keys add $\log N$ layers for NTTs of the merged keys.

4 NTT-Free Blind Rotation

In this section, we instantiate the NTT-free bucket update outlined in Section 1.2.

4.1 Lookup-Based External Product

Let $c = \mathsf{NTRU}_f(\mu, \sigma_\mu)$ and $\mathbf{C} = \mathsf{NGSW}_f(\nu, \sigma_\nu)$. Write $\mathfrak{g}_\beta^{-1}(c) = (c_0, \ldots, c_{\ell-1})$, where $c_t(X) = \sum_{k=0}^{N-1} c_{t,k} X^k$. The standard external product expands as

$$\mathbf{C} \odot c = \sum_{t=0}^{\ell-1} c_t(X) C_t = \sum_{k=0}^{N-1} X^k \left(\sum_{t=0}^{\ell-1} c_{t,k} C_t \right).$$

The first expression uses ℓ polynomial multiplications, while the last expression exposes scalar products by the small decomposition digits. Generate a digit-indexed table of NGSW ciphertexts $\mathsf{T}(\nu)$, such that

$$\mathsf{T}_\xi(\nu) = \mathsf{NGSW}_f(\xi\nu, \sigma_\nu), \quad \forall 1 \leq \xi \leq \beta/2,$$

and $\mathsf{T}_{-\xi}(\nu) = -\mathsf{T}_{\xi}(\nu)$ by convention. We define the lookup-based NTRU external product as

$$\mathsf{T}(\nu) \odot_{\mathsf{lut}} c = \sum_{k=0}^{N-1} X^k \left(\sum_{t\in[\ell], c_{t,k}\neq 0} \left(\mathsf{T}_{c_{t,k}}(\nu)\right)_t \right).$$

It outputs $\mathsf{NTRU}_f(\nu\mu, \sigma_{\mathsf{lut}})$ with estimated noise variance

$$\sigma_{\mathsf{lut}}^2 \lesssim \|\nu\|_2^2\sigma_\mu^2 + W_c\sigma_\nu^2 + \|\nu f\|_2^2\sigma_{\mathfrak{g}}^2,$$

where $W_c = \#\{(t,k) : c_{t,k} \neq 0\}$. For a public μ evaluated with tables encrypting $\xi\nu/f$, the first term is absent and the last term becomes $\|\nu\|_2^2\sigma_{\mathfrak{g}}^2$. Each selected component therefore contributes one fresh-key error with coefficient one, removing the $\beta^2/12$ amplification in a standard external product.

For a uniformly distributed base-β digit, it is zero with probability $1/\beta$, and hence $\mathbb{E}[W_c] = \ell N(1-\beta^{-1})$. For simplicity, all subsequent noise and computation estimates use this expectation directly.

4.2 Table Bootstrapping Keys and Merging

The offline procedure applies the lookup construction to every compiled selector. For each bucket $j \in [h']$, slot $u \in [m_j] \cup \{\perp\}$, and $1 \le \xi \le \beta/2$, generate $\mathsf{TBRK}_{j,u}$ as a table ciphertext for $s'_{j,u}$, such that

$$\mathsf{TBRK}_{j,u,\xi} = \begin{cases} \mathsf{NGSW}_f(\xi s'_{0,u}/f, \sigma), & j = 0, \\ \mathsf{NGSW}_f(\xi s'_{j,u}, \sigma), & j \ge 1. \end{cases}$$

We use $\mathsf{TBRK}_{j,u,-\xi} = -\mathsf{TBRK}_{j,u,\xi}$ and $\mathsf{TBRK}_{j,u,0} = 0$ as notation. Given an LWE ciphertext, the online evaluator sets $\widetilde{a}_i = \lfloor 2Na_i/q \rceil$ and $\widetilde{a}'_{j,u} = \widetilde{a}_{\pi(j,u)}$, and merges the slot keys into $\widetilde{\mathsf{TBRK}}_j$ as

$$\widetilde{\mathsf{TBRK}}_{j,\xi} = \mathsf{TBRK}_{j,\perp,\xi} + \sum_{u=0}^{m_j-1} X^{\widetilde{a}'_{j,u}} \mathsf{TBRK}_{j,u,\xi}. \qquad (2)$$

Therefore, each $\widetilde{\mathsf{TBRK}}_{j,\xi}$ encrypts $\xi\rho_j$ for $j \ge 1$, and $\xi\rho_0/f$ for $j = 0$, respectively.

4.3 Lookup-Based Blind Rotation

We apply the lookup-based NTRU external product along with the table-key design to modify the PBC-NTRU standard blind rotation Algorithm 1, and present the PBC-based NTT-free blind rotation as in Algorithm 2. We defer its correctness and noise estimate to Theorem 2, and discuss its computational complexity and parallelism in Section 4.4.

Theorem 2. *Algorithm 2 correctly outputs* $\mathsf{ACC}^{(h')} = \mathsf{NTRU}_f\left(X^{-\widetilde{b}+\langle\widetilde{\mathbf{a}},\mathbf{s}\rangle} r(X), \sigma_{\mathsf{br}}\right)$, *where*

$$\sigma_{\mathsf{br}}^2 \lesssim (Cn+h')\ell N(1-\beta^{-1})\sigma^2 + \left(1+(h'-1)\|f\|_2^2\right)\sigma_{\mathfrak{g}}^2.$$

Proof. Similar to the proof of Theorem 1. □

Algorithm 2: PBC-NTRU NTT-Free Blind Rotation

Input: An LWE ciphertext $\mathbf{c} = (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$ under $\mathbf{s}$; a test polynomial $r(X)$.
Auxiliaries : PBC layout and TBRK from Section 4.2.
Output: $\mathsf{NTRU}_f(X^{-\widetilde{b}+\langle\widetilde{\mathbf{a}},\mathbf{s}\rangle} r(X), \sigma_{\mathsf{br}})$.

$(\widetilde{\mathbf{a}}, \widetilde{b}) \leftarrow \lfloor 2N\mathbf{c}/q \rceil$;
Set each $\widetilde{a}'_{j,u}$ from $\widetilde{\mathbf{a}}$ with the PBC layout;
Form each $\widetilde{\mathsf{TBRK}}_{j,\xi}$ with Equation (2) in parallel;
$\mathsf{ACC}^{(0)} \leftarrow X^{-\widetilde{b}} r(X)$;
for $j = 0$ **to** $h' - 1$ **do**
  $\mathsf{ACC}^{(j+1)} \leftarrow \widetilde{\mathsf{TBRK}}_j \odot_{\mathsf{lut}} \mathsf{ACC}^{(j)}$;
return $\mathsf{ACC}^{(h')}$;

4.4 Storage, Computation, and Parallelism

Key storage. The PBC standard bootstrapping key contains $(Cn+h')\ell$ ring elements. The lookup construction stores $\beta/2$ positive-digit tables for every selector and therefore contains $(Cn+h')\ell\beta/2$ ring elements, a factor of $\beta/2$ larger than the standard bootstrapping key.

Key merging. For each bucket j, Equation (2) requires $m_j\ell\beta N/2$ additions in $\mathbb{Z}_Q$. Therefore, key merging requires $Cn\ell\beta N/2$ additions in $\mathbb{Z}_Q$ in total. These additions are independent across buckets, components, and coefficients, giving an ideal parallel depth $\lceil \log m_{\max} \rceil$, where $m_{\max} = \max_j m_j$.

Lookup-based external products. Each bucket update selects $\ell N(1-\beta^{-1})$ table components on average. Summing the length-N ring elements over all h' buckets requires $h'\ell N^2(1-\beta^{-1})$ additions in $\mathbb{Z}_Q$. Within each bucket, the additions across coefficients and selected table components are parallelizable, while the h' lookup-based external products remain sequential, giving an ideal parallel depth $h' \left\lceil \log\left(\ell N(1-\beta^{-1})\right) \right\rceil$.

Overall complexity. Together, the two stages have $\mathbb{Z}_Q$-addition complexity $O(\ell\beta CnN + h'\ell N^2)$. Compared with the $O(C\ell nN + h'\ell N \log N)$ $\mathbb{Z}_Q$ operations in Section 3, the lookup construction increases the overall cost but reduces the ideal parallel depth to $\lceil \log m_{\max} \rceil + h' \left\lceil \log\left(\ell N(1-\beta^{-1})\right) \right\rceil$, roughly subtracting $h' \log N$ from the standard PBC depth, where $m_{\max} = O(Cn/h')$ is the largest bucket size. The lookup-based approach suggests potentially faster arithmetic with specialized software and hardware.

5 Sparse-Secret Modulus Switching and Bootstrapping

In this section, we first present a modulus-switching method for sparse secrets, inspired by the mean-compensation tech-

nique of de Ruijter et al. [21], and analyze its noise reduction. We then complete the bootstrapping pipeline by integrating the PBC-based blind rotation with the remaining standard bootstrapping operations, as illustrated below:

$$\mathbf{c} \xrightarrow{\mathsf{ModSwitch}_{Q\to 2N}} (\widetilde{\mathbf{a}}, \widetilde{b}) \xrightarrow{\text{Blind Rotate}} \mathsf{ACC}^{(h')} \xrightarrow{\text{Extract}} \mathsf{LWE}_{\mathbf{f}}(\cdot) \xrightarrow{\text{LWE KeySwitch}} \mathsf{LWE}_{\mathbf{s}}(\cdot) = \mathbf{c}_{\mathsf{out}}.$$

5.1 Sparse-Secret Modulus Switching

The blind-rotation exponents are obtained by switching the input LWE ciphertext from Q to $2N$. Let $\mathbf{c} = (\mathbf{a}, b) \in \mathbb{Z}_Q^{n+1}$ be encrypted under $\mathbf{s} \in \{0,1\}^n$ with $\mathsf{HW}(\mathbf{s}) = h$. Standard nearest-integer rounding contributes the drift

$$e_{\mathsf{ms}} = \left(\widetilde{b} - \frac{2N}{Q} b\right) - \sum_{i=0}^{n-1} \left(\widetilde{a}_i - \frac{2N}{Q} a_i\right) s_i,$$

where $\widetilde{a}_i = \lfloor 2Na_i/Q \rceil$ and $\widetilde{b} = \lfloor 2Nb/Q \rceil$, and e_{ms} has a variance of $(1+h)/12$. Let $\delta_i = \widetilde{a}_i - 2Na_i/Q \in [-1/2, 1/2]$, and set $D = \sum_i \delta_i$. The body compensates the conditional mean of the mask rounding as $\widetilde{b}' = \lfloor 2Nb/Q + hD/n \rceil$. The improved mean-compensate modulus switching then returns $(\widetilde{\mathbf{a}}, \widetilde{b}') \bmod 2N$. The procedure is detailed in Algorithm 3.

Algorithm 3: Fixed-weight compensated ModSwitch

Input: $\mathbf{c} = (\mathbf{a}, b) \in \mathbb{Z}_Q^{n+1}$, secret Hamming weight h
Output: $(\widetilde{\mathbf{a}}, \widetilde{b}') \in \mathbb{Z}_{2N}^{n+1}$

$\widetilde{a}_i \leftarrow \lfloor 2Na_i/Q \rceil$, $\widetilde{b} \leftarrow \lfloor 2Nb/Q \rceil$;
$\delta_i \leftarrow \widetilde{a}_i - 2Na_i/Q \in \mathbb{R}$;
$D \leftarrow \sum_{i=0}^{n-1} \delta_i$;
$\widetilde{b}' = \lfloor 2Nb/Q + hD/n \rceil$;
return $(\widetilde{\mathbf{a}}, \widetilde{b}') \bmod 2N$;

Denote $\eta = \widetilde{b}' - 2Nb/Q - hD/n \in [-1/2, 1/2]$, and

$$\gamma = \frac{h(n-h)}{n(n-1)} \left(\sum_{i=0}^{n-1} \delta_i^2 - \frac{D^2}{n}\right),$$

we present the analysis of Algorithm 3 as in Theorem 3.

Theorem 3. *Algorithm 3 satisfies*

$$\widetilde{b}' - \langle \widetilde{\mathbf{a}}, \mathbf{s} \rangle = 2N\left(b - \langle \mathbf{a}, \mathbf{s} \rangle\right)/Q + \widetilde{e}_{\mathsf{ms}} \pmod{2N},$$

where $\widetilde{e}_{\mathsf{ms}} = \eta + \sum_{i=0}^{n-1} \delta_i (h/n - s_i)$. *Therefore,* $\mathbb{E}_{\mathbf{s}}[\widetilde{e}_{\mathsf{ms}}] = \eta$ *and* $\mathrm{Var}_{\mathbf{s}}(\widetilde{e}_{\mathsf{ms}}) = \gamma$, *given that* $\mathbf{s} \xleftarrow{\$} \{\mathbf{s} \in \{0,1\}^n : \mathsf{HW}(\mathbf{s}) = h\}$. *Thus, we have* $\mathbb{E}_{\mathbf{s}}[\widetilde{e}_{\mathsf{ms}}^2] = \eta^2 + \gamma$.

Under the independent-uniform rounding heuristic (a natural corollary of pseudo-randomness of ciphertexts), i.e., η *and* $\{\delta_i\}_i$ *are independently uniform over* $[-1/2, 1/2)$, *we have*

$$\mathbb{E}_{\mathbf{c},\mathbf{s}}\left[\widetilde{e}_{\mathsf{ms}}^2\right] = \frac{1 + h - h^2/n}{12} < \frac{1+h}{12}.$$

Proof. We have $\mathbb{E}_{\mathbf{c}}[\eta] = \mathbb{E}_{\mathbf{c}}[\delta_i] = 0$, $\mathbb{E}_{\mathbf{c}}[\eta^2] = \mathbb{E}_{\mathbf{c}}[\delta_i^2] = 1/12$, and $\mathrm{Cov}[\delta_i, \delta_j] = 0$, $i \neq j$. Therefore, $\mathbb{E}_{\mathbf{c}}[\sum \delta_i^2] = n/12$, $\mathbb{E}_{\mathbf{c}}[D^2/n] = 1/12$, and thus $\mathbb{E}_{\mathbf{c}}[\gamma] = h(n-h)/(n(n-1)) \cdot (n-1)/12 = (h - h^2/n)/12$. Adding up, we have $\mathbb{E}_{\mathbf{c},\mathbf{s}}\left[\widetilde{e}_{\mathsf{ms}}^2\right] = \mathbb{E}_{\mathbf{c}}[\eta^2 + \gamma] = (1 + h - h^2/n)/12$. □

5.2 Bootstrapping Pipeline

Blind rotation. Under the standard encoding of messages and test polynomials [20, 22], the modulus-switched phase homomorphically selects the corresponding coefficient of $r(X)$ with either Algorithm 1 or Algorithm 2. The blind-rotation noise variance σ_{br}^2 is given by Theorem 1 or Theorem 2.

Extraction and LWEKS. As standard operations of classic FHEW/TFHE bootstrapping [20, 22, 37, 48], constant-term extraction gives $\mathsf{Ext}_0(\mathsf{ACC}^{(h')}) = \mathsf{LWE}_{\mathbf{f}}(\cdot)$, where an NTRU ciphertext is viewed as an RLWE ciphertext with $b = 0$, and standard LWE key switching returns $\mathsf{LWEKS}_{\mathbf{f}\to\mathbf{s}}(\mathsf{LWE}_{\mathbf{f}}(\cdot)) = \mathsf{LWE}_{\mathbf{s}}(\cdot) = \mathbf{c}_{\mathsf{out}}$. For simplicity, we directly use $q = Q$ for LWE key-switching, where Q is the bootstrapping modulus, with LWE error evaluated by the q/σ ratios. For base β_{ks}, length ℓ_{ks}, and error σ_{LWE}, the added variance is

$$\sigma_{\mathsf{ks}}^2 \lesssim \frac{1}{12} N \ell_{\mathsf{ks}} \beta_{\mathsf{ks}}^2 \sigma_{\mathsf{LWE}}^2 + \frac{\|f\|_2^2 Q^2}{12 \beta_{\mathsf{ks}}^{2\ell_{\mathsf{ks}}}}.$$

However, an n-dimensional RLWE key-switching can simultaneously accelerate the process to μs-level and shrink the key-switching-key size to under 1 MiB. We follow the view of Bergerat et al. [9], view the n-dimensional LWE secret also as an n-dimensional RLWE secret, and apply RLWE-style key-switching. Refer to [9, Algorithm 1] for details.

Noise and DFR. Based on the preceding analysis, the PBS noise variance is bounded by

$$\sigma_{\mathsf{out}}^2 \lesssim \left(\frac{2N}{Q}\right)^2 (\sigma_{\mathsf{br}}^2 + \sigma_{\mathsf{ks}}^2) + \frac{1 + h - h^2/n}{12}.$$

Following Guimarães and Pereira [28], the decryption failure rate (DFR) for a κ-bit PBS is estimated as

$$\mathsf{DFR} = \mathrm{erfc}\left(\frac{2N/2^{\kappa+2}}{\sqrt{2}\sigma_{\mathsf{out}}}\right).$$

Gate bootstrapping uses this DFR for $\kappa = 1$, with modified variance for the Boolean-gate operation

$$\sigma_{\mathsf{out}}^2 \lesssim \left(\frac{2N}{Q}\right)^2 (2\sigma_{\mathsf{br}}^2 + \sigma_{\mathsf{ks}}^2) + \frac{1 + h - h^2/n}{12},$$

following Algorithm 3 and the analysis of Lee et al. [37].

6 Evaluation

In this section, we provide a theoretical comparison with related works, specify our concrete parameter sets, and report experimental results on both CPU and GPU platforms.

6.1 Analysis and Comparison

For asymptotic complexity, we compare PBC-NTRU blind rotation in Algorithm 1 and the NTT-free design in Algorithm 2 with standard dense-key RLWE and NTRU bootstrapping algorithms, represented by TFHE-rs [20, 49] and FINAL [11]. We consolidate these baselines into a single representative category, since they differ only by a constant factor.

The comparison counts each (i)NTT as a single transform, excluding its internal computations from the tally of other $\mathbb{Z}_Q$-operations (i.e., additions and multiplications). A $\mathbb{Z}_Q^N$-vector addition or Hadamard product is counted as N individual $\mathbb{Z}_Q$-operations. To evaluate the total complexity, each transform is assigned a cost of $O(N \log N)$ $\mathbb{Z}_Q$-operations. Finally, the parallel depth assumes ideal parallelism, as summarized in Table 2, where C and ℓ are considered constant and $h' = \Theta(h)$.

Algorithm 1 reduces the sequential external-product chain from $O(n)$ to $O(h)$. Meanwhile, Algorithm 2 removes NTTs and the $\beta^2/12$ amplification in noise variance. Furthermore, its purely additive accumulation structure significantly enhances parallelism and reduces the overall parallel depth, at the cost of an increased total $\mathbb{Z}_Q$-operation count.

6.2 Parameters

We provide the LWE and NTRU parameter sets for our constructions, all of which provide at least 128 bits of security. Specifically, we estimate the security level λ using the Lattice Estimator [1], which already considers the latest attacks against sparse secrets and the DSD attack [23] for NTRU. However, the PBC scheduling reveals information about the secret key, and decreases the security level by at most $\log(1/p_{\mathsf{succ}})$. We ensure the security levels given by the Lattice Estimator are still above 128 bits with such estimated security loss. Refer to Appendix A for detailed discussion.

Our first-layer LWE instances use fixed-weight sparse binary keys compatible with PBC scheduling. The set LWE_1 targets gate bootstrapping with $n = 1024$ and $h = 58$, while LWE_2 and LWE_3 target 4-bit and 6-bit PBS with lower LWE errors, respectively. Table 3 specifies these parameter sets, where Dist. stands for distribution, and q, β_{ks}, and ℓ_{ks} denote the modulus and gadget parameters for LWE key-switching.

The NTRU parameter sets in Table 4 instantiate the accumulator used by Algorithm 1, where Q represents NTT-friendly prime moduli[1]. For the NTT-free Algorithm 2, we instantiate gate bootstrapping with LWE_1 where ℓ_{ks} is modified to 10, and NTRU parameters $N = 1024$, $Q = 2^{18}$, $\sigma = 0.5$ with $\lambda = 144.1$, over 128-bit security.

For a fair comparison, we report parameter sets whose DFRs match the recommended parameters of corresponding baselines. For Boolean gate bootstrapping, we give $\mathsf{P}_1 = \mathsf{LWE}_1\&\mathsf{NTRU}_1$ for $\mathsf{DFR} < 2^{-32}$ as TFHE in OpenFHE[2] and FINAL[3], and $\mathsf{P}'_1 = \mathsf{LWE}_1\&\mathsf{NTRU}_2$ for $\mathsf{DFR} < 2^{-64}$ target of TFHE-rs 1.7.0[4]. For 4-bit PBS, we present $\mathsf{P}_4 = \mathsf{LWE}_2\&\mathsf{NTRU}_3$ with $\mathsf{DFR} < 2^{-128}$ and compare with the TFHE-rs setting at the same target. Since the 4-bit parameter set is the default recommendation in TFHE-rs and underlies its high-level integer benchmarks, we focus on this practically relevant setting. We additionally report a 6-bit programmable-bootstrapping instance, $\mathsf{P}_6 = \mathsf{LWE}_3\&\mathsf{NTRU}_3$, against the precision-matched TFHE-rs parameter set, with both achieving $\mathsf{DFR} < 2^{-40}$. We omit higher precisions, for which stronger DFR targets require larger rings, e.g., $N \geq 4096$, and incur substantial computational and key-size overheads. The DFRs of our parameter sets are estimated as described in Section 5.2, whereas those of the baselines are taken from their corresponding sources.

6.3 CPU Evaluation

Experimental setup. All CPU experiments, including the baseline measurements, are performed on the same machine: an AMD Ryzen 7 9800X3D with 32 GB RAM, running Ubuntu 22.04 LTS with full AVX-512 support. Each reported latency is averaged over 1,000 executions after warm-up, excluding key generation and one-time preprocessing. Unless otherwise specified, CPU results are single-threaded. When comparing with TFHE in OpenFHE and FINAL, we disable AVX in our implementation because these baselines do not use AVX optimizations. When comparing with TFHE-rs, both implementations are compiled with AVX-512 enabled.

Performance comparison. Table 5 reports the single-threaded CPU performance of bootstrapping algorithms and corresponding key sizes. At comparable DFRs, P_1 takes 2.33 ms without AVX, achieving 5.88× and 4.73× speedups over TFHE (OpenFHE) and FINAL, respectively. With AVX-512, P'_1 takes 0.83 ms and achieves a 3.31× speedup while reducing the BRK size by 1.31×. For 4-bit PBS, P_4 requires 1.75 ms and a 49.55 MiB BRK, achieving a 4.18× speedup over TFHE-rs with a comparable BRK size. For 6-bit PBS, P_6 requires 2.65 ms and a 96.80 MiB BRK, which is 20.47× faster than TFHE-rs and reduces the BRK size by 4.78×. We omit Jain et al. [31] from Table 5 because their code is not publicly available. But, benefiting from NTRU ciphertexts and our sparse-secret modulus switching, our non-AVX

[1] We use the following three NTT-friendly moduli: $Q = 536{,}856{,}577 \approx 2^{28.9999}$, $Q = 134{,}215{,}681 \approx 2^{26.9999}$, and $Q = 4{,}294{,}955{,}009 \approx 2^{31.9999}$.

[2] https://github.com/openfheorg/openfhe-development

[3] https://github.com/KULeuven-COSIC/FINAL

[4] https://github.com/zama-ai/tfhe-rs

[4] Our source code, including CPU, GPU, and applications, will be made publicly available upon acceptance.

Table 2: Asymptotic online arithmetic.

Method	NTTs	Other $\mathbb{Z}_Q$ ops.	Total $\mathbb{Z}_Q$-complexity	Parallel depth
TFHE [20, 49] FINAL [11]	$O(n)$	$O(nN)$	$O(nN\log N)$	$1+n\,(2\log N+\lceil\log \ell\rceil+1)$
PBC standard NTRU	$O(h)$	$O(nN)$	$O(hN\log N+nN)$	$\lceil\log m_{\max}\rceil+h'\,(2\log N+\lceil\log \ell\rceil+1)$
PBC NTT-free NTRU	0	$O(hN^2+nN)$	$O(hN^2+nN)$	$\lceil\log m_{\max}\rceil+h'\,\lceil\log\left(\ell N(1-\beta^{-1})\right)\rceil$

Table 3: LWE parameter sets.

Set	n	Dist.	h	q/σ	λ	ℓ_{ks}	β_{ks}
LWE_1	1024	Bin.	58	$2^{13.0}$	131.1	14	2
LWE_2	1024	Bin.	96	$2^{19.8}$	132.1	8	2^2
LWE_3	2048	Bin.	48	$2^{20.3}$	129.1	8	2^2

Table 4: NTRU parameter sets for Algorithm 1.

Set	N	Q	σ	λ	ℓ	β
NTRU_1	1024	$\approx 2^{28.99}$	14.0	128.0	2	2^9
NTRU_2	1024	$\approx 2^{26.99}$	7.0	128.0	3	2^6
NTRU_3	2048	$\approx 2^{31.99}$	0.5	137.8	2	2^{10}

implementation achieves 2.33 ms at over 128-bit security, corresponding to a 2.49× speedup over their reported 5.8 ms result and a 1.93× speedup even over their 4.5 ms instance of 120-bit security.

Multithreaded Evaluation. Multithreading reduces the best observed latencies of P_1, P'_1, P_4, and P_6 to 0.16, 0.24, 0.59, and 0.67 ms using 5, 4, 4, and 7 threads, respectively. Compared with the single-threaded results in Table 5, these correspond to speedups of 3.42×, 3.46×, 2.98×, and 3.96×. Complete scaling results are reported in Appendix C.

6.4 GPU Evaluation

We implement Algorithm 1 for all four parameter sets on an NVIDIA GeForce RTX 4090 24 GiB, using CUDA 12.6. Batch-one latency is averaged over 20 executions after warm-up, while peak throughput is measured over a logical batch of 16,384 bootstraps, matching the setting of VeloFHE [44]. For a fair comparison with other schemes, all reported results are evaluated under 128-bit security with $\mathsf{DFR} < 2^{-32}$. Table 6 compares our results with those reported by prior FPGA, ASIC, and GPU accelerators. Since these results were obtained under different hardware and measurement settings, cross-platform comparisons are provided for context. We primarily compare against implementations evaluated on the same RTX 4090 GPU model. Further GPU design details are provided in Appendix D.

On the RTX 4090, P_1 and P'_1 achieve batch-one latencies of 0.280 ms and 0.350 ms, respectively, together with throughputs of 154,739 and 108,833 OPS. In particular, P_1 provides 13.6× and 4.64× the throughput reported by VeloFHE and Maverick, respectively, on the same GPU model. At 4-bit precision, P_4 achieves a latency of 1.137 ms and a throughput of 51,401 OPS, corresponding to throughput improvements of 36.5× over VeloFHE and 10.8× over Maverick. At 6-bit precision, P_6 achieves 0.692 ms batch-one latency and 41,449 OPS. Its lower batch-one latency than P_4 is consistent with the smaller secret weight, which shortens the sequential external-product chain. At large batches, however, its larger LWE and ring dimensions incur more selector construction, arithmetic, and memory traffic, resulting in lower saturated throughput.

Table 5: CPU bootstrapping performance, single-threaded.

Construction	Parameters	BRK (MiB)	Time (ms)
Boolean gate bootstrapping without AVX instructions			
TFHE [19, 40]	–	26.53	13.69
FINAL [11]	–	13.01	11.03
Ours	P_1	24.48	2.33
Ours	P'_1	36.71	3.37
Boolean gate bootstrapping with AVX-512			
TFHE-rs [49]	–	48.09	2.75
Ours	P_1	24.48	0.56
Ours	P'_1	36.71	0.83
4-bit PBS with AVX-512			
TFHE-rs [49]	–	54.13	7.31
Ours	P_4	49.55	1.75
6-bit PBS with AVX-512			
TFHE-rs [49]	–	462.50	54.24
Ours	P_6	96.80	2.65

6.5 Evaluation of NTT-free Variant

For the NTT-free Algorithm 2, we use LWE_1 with $\ell_{\mathsf{ks}} = 10$ and NTRU parameters $(N, Q, \sigma, \beta, \ell) = (1024, 2^{18}, 0.5, 16, 3)$. This instance has a 293.72 MiB BRK and achieves $\mathsf{DFR} < 2^{-40}$. Our CPU implementation achieves a latency of 3.47 ms, while our RTX 4090 implementation reduces it to 0.47 ms, achieving a 7.38× speedup. Although this is a cross-platform comparison, our result is also 1.72× faster than the 0.81 ms

Table 6: Cross-platform bootstrapping performance.

Scheme	Platform	Prec. (bit)	Lat. (ms)	Thr. (OPS)
FPT [7]	FPGA	1	0.66	28,400
NFP [38]	FPGA	1	0.29	3,448
Quatorze-bis [10]	FPGA	1	1.02	31,250
Strix [42]	ASIC(28 nm)	1	0.44	21,104
Morphling [41]	ASIC(28 nm)	1	0.16	98,930
VeloFHE [44]	GPU(4090)	1	–	11,378
VeloFHE [44]	GPU(4090)	4	–	1,408
Maverick [47]	GPU(4090)	1	–	33,333
Maverick [47]	GPU(4090)	4	–	4,762
TFHE-rs [4]	GPU(H100)	1	0.81	28,620
TFHE-rs [4]	GPU(H100)	4	0.96	24,170
TFHE-rs [4]	GPU(H100)	6	9.9	–
Ours (P_1)	GPU(4090)	1	0.28	154,739
Ours (P'_1)	GPU(4090)	1	0.35	108,833
Ours (P_4)	GPU(4090)	4	1.14	51,401
Ours (P_6)	GPU(4090)	6	0.69	41,449

reported by TFHE-rs on the H100 [4]. Beyond its concrete performance, Algorithm 2 demonstrates that online TFHE-style blind rotation can be implemented without polynomial transforms or modular multipliers. The computation reduces to vector additions and subtractions, with power-of-two modular reduction realized by the natural truncation. Future FPGA or ASIC implementations could exploit wide adder arrays, multi-operand accumulation, and near-memory processing to improve both latency and energy efficiency.

7 Applications

In this section, we present NSA, a general-purpose 8-bit FHE processor, and evaluate it in end-to-end applications.

7.1 NSA: 8-bit instruction set on PBC-NTRU

First, we design an NTRU-based instruction set architecture, NSA, a general-purpose 8-bit FHE processor. Our design builds upon the TFHE Processor framework of Trama et al. [45], which represents each encrypted byte by two LWE ciphertexts encrypting its high and low nibbles: $x = 16x_{\mathrm{H}} + x_{\mathrm{L}}, x_{\mathrm{H}}, x_{\mathrm{L}} \in \mathbb{Z}_{16}$. We adopt their primary LUT interfaces in this framework: SimpleBoot for one encrypted nibble, LUTeval for two encrypted nibbles, and MVLUTeval for multiple LUT evaluations sharing the first tree level.

Refined Tree-Based LUT Evaluation. Our two-level construction eliminates the costly inter-level bridge inherent in the tree-based framework of Trama et al. [45]. As shown in Figure 1(a), their approach applies SampleExtract to each first-level RLWE row ciphertext, followed by a coordinate-wise PublicFunctionalKeyswitch to pack the resulting LWE ciphertexts into an RLWE ciphertext, which serves as the initial accumulator for the second blind rotation. Unfortunately, PublicFunctionalKeyswitch incurs prohibitive computational overhead and requires an excessively large key size.

To address these bottlenecks, we propose a novel and efficient tree-based LUT evaluation framework. Our design completely removes the expensive PublicFunctionalKeyswitch step, achieving significantly lower computational complexity and drastically reduced key storage, as illustrated in Figure 1(b). The core insight is that, following the first blind rotation and multiplication by the public row polynomials, the candidate row values already appear as the constant coefficients of the resulting NTRU ciphertexts. Leveraging this observation, we introduce *NTRU RevHomTrace Packing* (Algorithm 4, detailed in Appendix F), which packs these ciphertexts directly into the initial NTRU accumulator for the second blind rotation, thereby eliminating both SampleExtract and PublicFunctionalKeyswitch. [5] This optimization reduces the total key size from 17,104 MiB in Trama et al. [45] to 124.68 MiB, a 137.18× reduction. Moreover, the same inter-level bridge naturally extends to MVLUTeval: the first blind rotation is shared across multiple tables, while the second-level branches are evaluated independently.

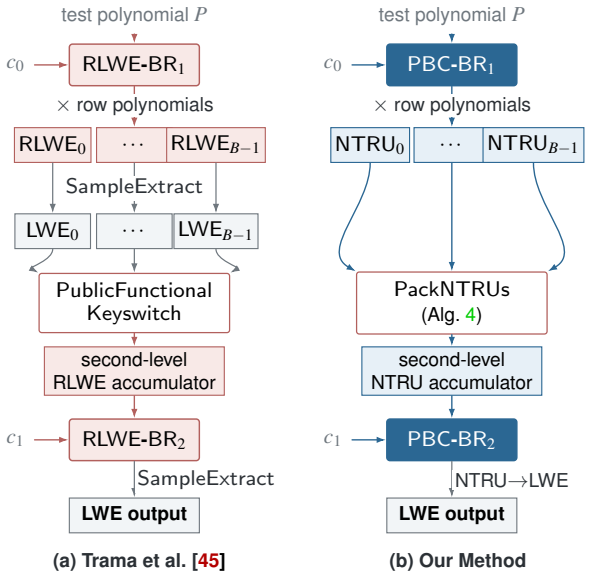

Figure 1: Inter-level bridge for LUTeval and MVLUTeval.

In Table 7, we compare representative instruction latencies of NSA with TFHE Processor [45], both evaluated single-threaded on the same CPU platform described in Section 6.3. One can see that NSA achieves speedups ranging from 2.00×

[5] We use the parameter set P_4 for NSA but adjust ℓ to 3 for tree-based LUT, achieving DFR $< 2^{-40}$ as in [45].

to 41.98× across different instructions. In Appendix E.1, we provide a detailed description of the instruction set.

Table 7: Benchmark of 8-bit instructions.

Instruction	Latency (ms) [45]	Latency (ms) NSA	Speedup
ANDi/ORi/XORi	56.79	3.61	15.73×
ADDi/SUBi	114.94	10.69	10.75×
MULi	116.15	10.95	10.61×
DIVi	136.18	10.48	12.99×
MOD4i	117.66	8.32	14.15×
SHLi/SHRi	85.07	10.88	7.82×
ROLi/RORi	111.85	14.33	7.81×
EQi	90.50	8.30	10.91×
LTEi	89.54	2.13	41.98×
NEG	200.53	10.99	18.24×
ABS	191.42	14.12	13.56×
MINi/MAXi	189.58	10.61	17.86×
AND/OR/XOR	229.10	17.11	13.39×
SHL/SHR	419.20	209.80	2.00×
ROL/ROR	620.63	161.88	3.83×
EQ	367.16	19.98	18.37×
LT/GT/GTE	574.94	24.62	23.35×
ADD/SUB	442.52	24.46	18.09×
MUL	629.47	227.33	2.77×
MULM	2212.49	415.88	5.32×
DIV	6415.40	1089.34	5.89×
MOD	5990.45	1112.96	5.38×
MIN/MAX	1048.00	118.77	8.82×
CSEL	632.48	72.84	8.68×

7.2 End to End Applications

Encrypted Programs. As a small-scale end-to-end evaluation, we consider four representative programs over an encrypted array of five 8-bit integers: Maximum, Average, and SquareSum compute the largest element, the arithmetic mean, and the sum of the squares of the array, respectively, and BubbleSort sorts the array using compare-and-swap operations. As shown in Table 8, our implementation achieves end-to-end speedups of 11.35×, 17.17×, 4.22×, and 9.42×, respectively, over TFHE Processor [45].

Table 8: Benchmark of 8-bit programs.

Program	[45]	NSA	Speedup
Maximum	4411.98	388.85	11.35×
Average	2212.82	128.89	17.17×
SquareSum	5107.75	1211.46	4.22×
BubbleSort	15648.82	1660.98	9.42×

Homomorphic AES-128 Evaluation. Transciphering reduces communication and storage costs by homomorphically converting symmetric ciphertexts into FHE ciphertexts. We instantiate this application with a full ten-round homomorphic evaluation of AES-128. [6]

In Table 9, we compare our method with two classes of TFHE-like constructions: the PBS-based Hippogryph construction [8][7], and the circuit-bootstrapping-based Refined LHE and Refined Flex.LHE constructions [46][8]. For DFR $< 2^{-32}$, our construction evaluates one AES-128 block in 5.292 s, achieving speedups of 3.26×, 1.69×, and 1.56× over Hippogryph, Refined LHE, and Refined Flex.LHE, respectively. For DFR $< 2^{-128}$, our construction requires 7.430 s, corresponding to speedups of 37.5×, 2.00×, and 1.85×, respectively. Appendix E.2 provides complete step-by-step details, with Figure 3 depicting the per-round dataflow.

Table 9: Benchmark of homomorphic AES-128.

DFR	Scheme	Latency (s)	Speedup
2^{-32}	Hippogryph [8]	17.269	3.26×
	Refined LHE [46]	8.940	1.69×
	Refined Flex.LHE [46]	8.265	1.56×
	Ours	**5.292**	1.00×
2^{-128}	Hippogryph [8]	278.340	37.5×
	Refined LHE [46]	14.886	2.00×
	Refined Flex.LHE [46]	13.748	1.85×
	Ours	**7.430**	1.00×

8 Conclusion

In this work, we explored how to further reduce the computational costs of gate and functional bootstrapping in NTRU-based FHE. Our results, spanning algorithmic improvements, hardware acceleration, and concrete applications, demonstrate the potential of NTRU-based FHE. We observe that modular additions overwhelmingly dominate the computational workload in both our NTT-based implementation and its NTT-free variant. This suggests that a dedicated architecture optimized for high-throughput modular addition could further reduce bootstrapping latency and improve hardware efficiency.

Acknowledgments

We thank ChatGPT for assistance with English language polishing. All technical content was reviewed and verified by the authors.

[6] We use the parameter set P_4 for AES but adjust ℓ to 3 for tree-based LUT evaluation at a full-round DFR $< 2^{-32}$ and to 4 at a full-round DFR $< 2^{-128}$.

[7] https://github.com/CryptoExperts/Hippogryph

[8] https://github.com/KAIST-CryptLab/refined-tfhe-lhe

References

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

[2] Andreea Alexandru, Andrey Kim, and Yuriy Polyakov. General functional bootstrapping using CKKS. In *Advances in Cryptology – CRYPTO 2025*, volume 16002, pages 304–337. Springer Nature Switzerland, Cham, 2025.

[3] Jacob Alperin-Sheriff and Chris Peikert. Faster bootstrapping with polynomial error. In *Advances in Cryptology – CRYPTO 2014*, volume 8616, pages 297–314. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[4] Pedro Geraldo M. R. Alves, Beka Barbakadze, Enzo Di Maria, Jakub Klemsa, Agnes Leroy, and Guillermo Oyarzun. Sub-millisecond TFHE bootstrapping on GPU. Poster presented at the 5th Annual FHE.org Conference on Fully Homomorphic Encryption, March 2026. Accessed: Aug. 16, 2026.

[5] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 962–979, San Francisco, CA, May 2018. IEEE.

[6] Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, and Damien Stehlé. Bootstrapping bits with CKKS. In *Advances in Cryptology – EUROCRYPT 2024*, volume 14652, pages 94–123. Springer Nature Switzerland, Cham, 2024.

[7] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Furkan Turan, and Ingrid Verbauwhede. FPT: A fixed-point accelerator for torus fully homomorphic encryption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 741–755. ACM, 2023.

[8] Sonia Belaïd, Nicolas Bon, Aymen Boudguiga, Renaud Sirdey, Daphné Trama, and Nicolas Ye. Further improvements in AES exe cution over TFHE. *IACR Commun. Cryptol.*, 2(1):39, 2025.

[9] Loris Bergerat, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, Adeline Roux-Langlois, and Samuel Tap. New secret keys for enhanced performance in (T)FHE. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024*, pages 2547–2561. ACM, 2024.

[10] Jonas Bertels, Hilder V. L. Pereira, and Ingrid Verbauwhede. FINAL bootstrap acceleration on FPGA using dsp-free constant-multiplier ntts. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(3):293–316, 2025.

[11] Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. FINAL: Faster FHE instantiated with NTRU and LWE. In *Advances in Cryptology – ASIACRYPT 2022*, volume 13792, pages 188–215. Springer Nature Switzerland, Cham, 2022.

[12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Advances in Cryptology – CRYPTO 2012*, volume 7417, pages 868–886. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[13] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325, Cambridge Massachusetts, January 2012. ACM.

[14] Chunling Chen, Zhihao Li, Qingyun Niu, Xianhui Lu, Ruida Wang, Lutan Zhao, and Rui Hou. Sub-millisecond gate bootstrapping. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, pages 1431–1444, Bangalore India, June 2026. ACM.

[15] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient homomorphic conversion between (ring) LWE ciphertexts. In *Applied Cryptography and Network Security*, volume 12726, pages 460–479. Springer International Publishing, Cham, 2021.

[16] Jung Hee Cheon, Wonhee Cho, Jaehyung Kim, and Damien Stehlé. Homomorphic multiple precision multiplication for CKKS and reduced modulus consumption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 696–710. ACM, 2023.

[17] 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, pages 398–428. Springer Nature Switzerland, Cham, 2025.

[18] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology – ASIACRYPT 2017*, volume 10624, pages 409–437. Springer International Publishing, Cham, 2017.

[19] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In

Advances in Cryptology – ASIACRYPT 2016, volume 10031, pages 3–33. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020.

[21] Thomas de Ruijter, Jan-Pieter D'Anvers, and Ingrid Verbauwhede. Don't be mean: Reducing Approximation Noise in TFHE through Mean Compensation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(1):82–104, January 2026.

[22] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology – EUROCRYPT 2015*, volume 9056, pages 617–640. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[23] Léo Ducas and Wessel van Woerden. NTRU fatigue: How stretched is overstretched? In *Advances in Cryptology – ASIACRYPT 2021*, volume 13093, pages 3–32. Springer International Publishing, Cham, 2021.

[24] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012.

[25] Jingwei Feng, Baofeng Wu, Dongdai Lin, and Binwu Xiang. Accelerating NTRU-based bootstrapping with block key distributions. In *Information Security and Cryptology*, volume 16408, pages 43–62. Springer Nature Singapore, Singapore, 2026.

[26] Nicolas Gama, Malika Izabachène, Phong Q. Nguyen, and Xiang Xie. Structural lattice reduction: Generalized worst-case to average-case reductions and homomorphic cryptosystems. In *Advances in Cryptology – EUROCRYPT 2016*, volume 9666, pages 528–558. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[27] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009.

[28] Antonio Guimarães and Hilder V. L. Pereira. Fast amortized bootstrapping with small keys and polynomial noise overhead. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 2967–2981, Taipei Taiwan, November 2025. ACM.

[29] Shai Halevi and Victor Shoup. Faster homomorphic linear transformations in HElib. In *Advances in Cryptology – CRYPTO 2018*, volume 10991, pages 93–120. Springer International Publishing, Cham, 2018.

[30] Intak Hwang, Seonhong Min, and Yongsoo Song. Carousel: Fully homomorphic encryption with bootstrapping over automorphism group. In *Advances in Cryptology – ASIACRYPT 2025*, volume 16251, pages 131–164. Springer Nature Singapore, Singapore, 2026.

[31] Aayush Jain, Huijia Lin, Zeyu Liu, and Sagnik Saha. New techniques for fast and shallow FHE bootstrapping and beyond. In *Advances in Cryptology – CRYPTO 2026*, volume 16801, pages 428–460. Springer Nature Switzerland, Cham, 2026.

[32] Marc Joye and Pascal Paillier. Blind rotation in fully homomorphic encryption with extended keys. In *Cyber Security, Cryptology, and Machine Learning*, volume 13301, pages 1–18. Springer International Publishing, Cham, 2022.

[33] Charanjit S. Jutla and Nathan Manohar. Sine series approximation of the mod function for bootstrapping of approximate HE. In *Advances in Cryptology – EUROCRYPT 2022*, volume 13275, pages 491–520. Springer International Publishing, Cham, 2022.

[34] Paul Kirchner and Pierre-Alain Fouque. Revisiting lattice attacks on overstretched NTRU parameters. In *Advances in Cryptology – EUROCRYPT 2017*, volume 10210, pages 3–26. Springer International Publishing, Cham, 2017.

[35] Changmin Lee, Seonhong Min, Jinyeong Seo, and Yongsoo Song. Faster TFHE bootstrapping with block binary keys. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, pages 2–13, Melbourne VIC Australia, July 2023. ACM.

[36] Kang Hoon Lee and Ji Won Yoon. Homomorphic field trace revisited : Breaking the cubic noise barrier. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(1):105–134, January 2026.

[37] Yongwoo Lee, Daniele Micciancio, Andrey Kim, Rakyong Choi, Maxim Deryabin, Jieun Eom, and Donghoon Yoo. Efficient FHEW bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption. In *Advances in Cryptology – EUROCRYPT 2023*, volume 14006, pages 227–256. Springer Nature Switzerland, Cham, 2023.

[38] Zhihao Li, Xianhui Lu, Zhiwei Wang, Ruida Wang, Ying Liu, Yinhang Zheng, Lutan Zhao, Kunpeng Wang, and Rui Hou. Faster NTRU-based bootstrapping in less than 4 ms. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(3):418–451, July 2024.

[39] Shihe Ma, Tairong Huang, Anyu Wang, and Xiaoyun Wang. Faster BGV bootstrapping for power-of-two cyclotomics through homomorphic NTT. In *Advances in*

Cryptology – ASIACRYPT 2024, volume 15484, pages 143–175. Springer Nature Singapore, Singapore, 2024.

[40] OpenFHE Development Team. OpenFHE: Open-source fully homomorphic encryption library. https://github.com/openfheorg/openfhe-development. Software library, accessed August 10, 2026.

[41] Prasetiyo, Adiwena Putra, and Joo-Young Kim. Morphling: A throughput-maximized TFHE-based accelerator using transform-domain reuse. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 249–262, Edinburgh, United Kingdom, March 2024. IEEE.

[42] Adiwena Putra, Prasetiyo, Yi Chen, John Kim, and Joo-Young Kim. Strix: An end-to-end streaming architecture with two-level ciphertext batching for fully homomorphic encryption with programmable bootstrapping. In *56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1319–1331, Toronto ON Canada, October 2023. ACM.

[43] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, pages 84–93, Baltimore MD USA, May 2005. ACM.

[44] Shiyu Shen, Hao Yang, Zhe Liu, Ying Liu, Xianhui Lu, Wangchen Dai, Lu Zhou, Yunlei Zhao, and Ray C. C. Cheung. VeloFHE: GPU acceleration for FHEW and TFHE bootstrapping. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(3):81–114, June 2025.

[45] Daphné Trama, Aymen Boudguiga, Pierre-Emmanuel Clet, Renaud Sirdey, and Nicolas Ye. Designing a general-purpose 8-bit (T)FHE processor abstraction. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(2):535–578, March 2025.

[46] Ruida Wang, Jincheol Ha, Xuan Shen, Xianhui Lu, Chunling Chen, Kunpeng Wang, and Jooyoung Lee. Refined TFHE leveled homomorphic evaluation and its application. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS 2025, Taipei, Taiwan, October 13-17, 2025*, pages 2309–2323. ACM, 2025.

[47] Zhiwei Wang, Haoqi He, Lutan Zhao, Qingyun Niu, Dan Meng, and Rui Hou. Maverick: Rethinking TFHE bootstrapping on GPUs via algorithm-hardware Co-design. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1168–1184, Pittsburgh PA USA, March 2026. ACM.

[48] Binwu Xiang, Jiang Zhang, Yi Deng, Yiran Dai, and Dengguo Feng. Fast blind rotation for bootstrapping FHEs. In *Advances in Cryptology – CRYPTO 2023*, volume 14084, pages 3–36. Springer Nature Switzerland, Cham, 2023.

[49] Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. https://github.com/zama-ai/tfhe-rs.

[50] Jiachen Zhao, Jiang Zhang, Binwu Xiang, Songyu Wu, Yi Deng, and Dengguo Feng. Faster polynomial evaluations for SIMD FHEs and application to BGV in HElib. In *Advances in Cryptology – CRYPTO 2026*, volume 16801, pages 461–491. Springer Nature Switzerland, Cham, 2026.

A Discussion on PBC Parameters, Security, and Efficiency

This appendix examines how the replication factor C and the bucket count h' affect three aspects of the PBC compiler: scheduling success, leakage from publishing an accepted layout, and online evaluation cost.

Implemented configuration and scheduling success. Our implementation uses $C = 3$ and $h' = h + 3$, following the aggressive concrete configuration of Jain et al. [31]. For ideal independent uniform hash functions, the scheduling probability depends only on h: any two supports of size h are related by a permutation of $[n]$, which preserves the distribution of the reverse-Cuckoo graph.

We estimate p_{succ} by Monte Carlo simulation for the three LWE parameter sets of Table 3. Each trial samples a fresh support of size h and fresh candidate buckets, and then tests whether the resulting graph has a matching that covers the support. Table 10 reports the success rates over 30,000 independent trials per point.

Table 10: PBC scheduling success rate for $C = 3$.

Set	h	$h' = h+2$	$h' = h+3$	$h' = h+4$
LWE_1	58	21.6%	37.0%	53.4%
LWE_2	96	3.0%	7.3%	14.7%
LWE_3	48	34.0%	51.9%	68.0%

Leakage from layout rejection sampling. For a fixed secret, key generation samples layouts independently until scheduling succeeds. It publishes only the accepted replication layout; the matching $u^\star$ is used solely to generate the encrypted selector bits and is never revealed. This distinction is essential: publishing $u^\star$ would reveal the support through

the canonical-decoding relation $\{\pi(j,u_j^\star) : u_j^\star \neq \perp\} = \mathrm{supp}(\mathbf{s})$. The following lemma bounds the leakage caused solely by rejection sampling.

Lemma 2 (Leakage of an accepted PBC layout)**.** *Let $\mathbf{s} \leftarrow \mathcal{D}_h$ be a fixed-weight secret, and let $L \leftarrow \mathcal{L}$ be a PBC layout sampled independently of $\mathbf{s}$. Define*

$$\mathsf{Acc}(\mathbf{s},L) \in \{0,1\}, \quad p_{\mathbf{s}} := \Pr_{L \leftarrow \mathcal{L}}[\mathsf{Acc}(\mathbf{s},L) = 1].$$

Suppose that $p_{\mathbf{s}} \geq p_{\mathsf{succ}} > 0$ for every $\mathbf{s} \in \mathrm{supp}(\mathcal{D}_h)$, and let $L^\star$ be the first accepted layout. Then

$$D_\infty(P_{\mathbf{s},L^\star} \,\|\, P_{\mathbf{s}} \times P_L) \leq \log \frac{1}{p_{\mathsf{succ}}}.$$

Consequently, for every event $\mathcal{E}$,

$$\Pr[\mathcal{E}(\mathbf{s},L^\star)] \leq \frac{\Pr[\mathcal{E}(\mathbf{s},L)]}{p_{\mathsf{succ}}},$$

where L is independent of $\mathbf{s}$. Moreover,

$$I(\mathbf{s};L^\star) \leq \log \frac{1}{p_{\mathsf{succ}}}$$

and

$$\widetilde{H}_\infty(\mathbf{s} \mid L^\star) \geq H_\infty(\mathbf{s}) - \log \frac{1}{p_{\mathsf{succ}}}.$$

Proof. Let $\mu(\ell) = \Pr[L = \ell]$. For every fixed secret $\mathbf{s}$, rejection sampling gives

$$\Pr[L^\star = \ell \mid \mathbf{s}] = \frac{\mu(\ell)\mathbf{1}[\mathsf{Acc}(\mathbf{s},\ell) = 1]}{p_{\mathbf{s}}}.$$

Hence

$$\frac{P_{\mathbf{s},L^\star}(\mathbf{s},\ell)}{P_{\mathbf{s}}(\mathbf{s})\mu(\ell)} = \frac{\mathbf{1}[\mathsf{Acc}(\mathbf{s},\ell) = 1]}{p_{\mathbf{s}}} \leq \frac{1}{p_{\mathsf{succ}}}.$$

This proves the max-divergence bound, and summing the same pointwise inequality over $\mathcal{E}$ proves the event bound. Furthermore,

$$D(P_{\mathbf{s},L^\star} \| P_{\mathbf{s}} \times P_L) = \mathbb{E}_{\mathbf{s}}\left[\log \frac{1}{p_{\mathbf{s}}}\right] \leq \log \frac{1}{p_{\mathsf{succ}}}.$$

Because

$$D(P_{\mathbf{s},L^\star} \| P_{\mathbf{s}} \times P_L) = I(\mathbf{s};L^\star) + D(P_{L^\star} \| P_L),$$

the mutual-information bound follows. Finally,

$$\begin{aligned} P_{\mathsf{guess}}(\mathbf{s} \mid L^\star) &= \sum_\ell \max_{\mathbf{s}} P_{\mathbf{s},L^\star}(\mathbf{s},\ell) \\ &\leq \frac{1}{p_{\mathsf{succ}}} \sum_\ell \mu(\ell) \max_{\mathbf{s}} P_{\mathbf{s}}(\mathbf{s}) \\ &= \frac{P_{\mathsf{guess}}(\mathbf{s})}{p_{\mathsf{succ}}}, \end{aligned}$$

which proves the average conditional min-entropy bound. □

Concrete security accounting. Lemma 2 is attack-agnostic. In particular, if a key-recovery attack succeeds with probability at most $2^{-\lambda}$ when the public layout is independent of the secret, its success probability with an accepted layout is at most

$$\frac{2^{-\lambda}}{p_{\mathsf{succ}}} = 2^{-(\lambda - \log(1/p_{\mathsf{succ}}))}.$$

Thus rejection sampling costs at most $\log(1/p_{\mathsf{succ}})$ bits in this generic accounting.

For the implemented choice $h' = h+3$, the point estimates in Table 10 give losses of 1.44, 3.78, and 0.95 bits for LWE_1, LWE_2, and LWE_3, respectively. Since these probabilities are estimated, we additionally use one-sided Clopper–Pearson lower bounds. With a per-instance confidence failure probability of 2^{-66}, the three bounds hold simultaneously with probability greater than $1 - 2^{-64}$ by a union bound. Even using the smallest success counts consistent with the one-decimal percentages in Table 10 gives conservative lower bounds of approximately 34.4%, 5.95%, and 49.2%. These correspond to losses of at most 1.54, 4.08, and 1.03 bits. Subtracting them from the estimates in Table 3 leaves approximately 129.56, 128.02, and 128.07 bits, respectively. All three parameter sets therefore remain above 128 bits under this generic leakage bound and the stated statistical confidence. This calculation accounts only for publication of the accepted layout; the encrypted selector material is covered by the underlying cryptographic assumptions.

Two-copy layouts. The replication factor exposes a direct bandwidth–latency tradeoff. Reducing C from 3 to 2 decreases the replicated slots from $3n$ to $2n$ and reduces the number of independent merge additions by roughly one third. To retain a useful scheduling probability, however, it requires more buckets and hence more sequential external products. Table 11 reports representative $C = 2$ choices, again estimated from 30,000 independent trials per point.

Table 11: Representative $C = 2$ PBC configurations.

		$\approx 10\%$		$\approx 25\%$		$\approx 50\%$	
Set	h	h'	p_{succ}	h'	p_{succ}	h'	p_{succ}
LWE_1	58	76	9.4%	82	25.0%	90	49.6%
LWE_2	96	135	10.6%	144	25.3%	156	49.5%
LWE_3	48	62	11.5%	66	24.4%	73	50.0%

Relative to the implemented $C = 3$, $h' = h+3$ configuration, these choices increase the number of NTT-based bucket products by approximately $1.2\times$–$1.6\times$. They are therefore useful when merge bandwidth or available SIMD parallelism is the bottleneck, whereas $C = 3$ remains preferable when minimizing the sequential blind-rotation chain is the primary objective.

B PBC Key-Preprocessing Strategies

We compare the three key-preprocessing strategies in Section 3.1: coefficient-domain, NTT-domain, and precomputed-shifts representations of the standard-PBC bootstrapping key. All parameter sets use $C = 3$ candidate hash positions. For LWE dimension n, ring degree N, and gadget length ℓ, the precomputed-shifts table requires

$$M_{\text{shift}} = (3n + h')\ell N^2 \cdot 4 \quad \text{bytes.}$$

with 32-bit ring coefficients. The resulting table sizes are 24.48, 36.71, 99.09, and 193.59 GiB for the four parameter sets, respectively.

To give a fair comparison with such memory requirement, we evaluate all three representations on an Intel Xeon Platinum 8358 at 2.60 GHz with 2 TiB RAM and Ubuntu 22.04. Results for our different parameter sets are given in Table 12

Table 12: Single-threaded latency under different key representations (ms).

Set	Coeff.	NTT	Pre. Shifts
P_1	**2.08**	2.95	2.39
P'_1	**3.10**	4.39	4.21
P_4	8.54	17.90	**7.58**
P_6	**11.66**	34.10	12.05

Coefficient-domain preprocessing is 1.42×, 1.42×, 2.10×, and 2.92× faster than NTT-domain preprocessing for the four parameter sets. Precomputed shifts outperform the coefficient-domain method only for $\mathsf{LWE}_2\&\mathsf{NTRU}_3$, reducing its latency from 8.54 to 7.58 ms. For the other parameter sets, the additional memory traffic offsets the saved computation. We therefore use coefficient-domain preprocessing for the primary CPU results.

C Multithreaded CPU Evaluation

We parallelize the coefficient-domain key-merging stage using a producer–consumer pipeline. For a total of $T \geq 2$ threads, $T - 1$ producers dynamically process the unequal-length Cuckoo buckets. Each producer merges the rotated BRK rows over all ℓ gadget levels, applies lazy reduction, and transforms the result to the NTT domain. One consumer then applies the merged keys in bucket order, since the external products form a sequential accumulator dependency. This design overlaps bucket merging with external-product evaluation. All workspaces are allocated before measurement, and the threads are pinned to distinct physical cores.

As shown in Figure 2, the best observed latencies of P_1, P'_1, P_4, and P_6 decrease from 0.56, 0.83, 1.75, and 2.65 ms to 0.16, 0.24, 0.59, and 0.67 ms using 5, 4, 4, and 7 threads, respectively. These correspond to speedups of 3.42×, 3.46×, 2.98×, and 3.96×, respectively.

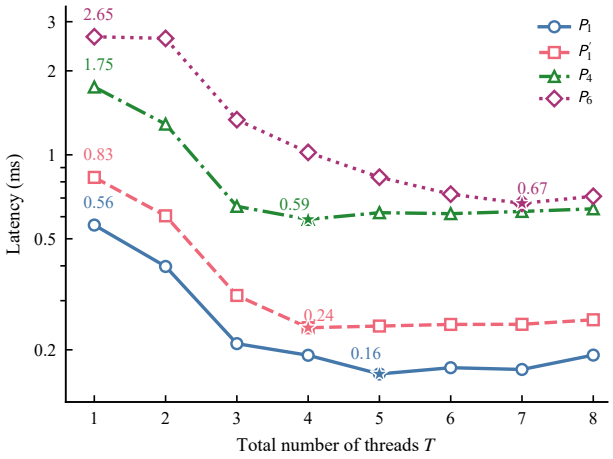

Figure 2: Latency versus total thread count T.

D GPU Dataflow and Kernel Organization

Evaluation-domain selector construction. We store BRKs in the NTT evaluation domain and construct PBC selectors directly therein. For each bucket $j \in [h']$ and each NTT frequency $f \in [N]$,

$$\widehat{\widetilde{\mathsf{BRK}}}_j[f] = \widehat{\mathsf{BRK}}_{j,\perp}[f] + \sum_{u=0}^{m_j-1} \psi^{(2f+1)\tilde{a}'_{j,u}} \cdot \widehat{\mathsf{BRK}}_{j,u}[f] \mod Q,$$

where the hat denotes the NTT representation, and ψ is the $2N$-th root of unity modulo Q. We precompute the $2N$ powers of ψ and evaluate each merging with them and modular multiplication.

GPU mapping. Parallelism is exposed across ciphertexts, buckets, gadget levels, and NTT frequencies. Each CUDA block handles one bucket–level pair for a ciphertext or a small ciphertext tile, while threads partition the N frequencies and sequentially accumulate the slots assigned to that bucket. The ψ table is staged in shared memory, and neighboring threads access consecutive BRK coefficients. For batched execution, each loaded BRK coefficient can also be reused across several ciphertext-specific accumulators. We autotune this ciphertext tile size because its benefit depends on both register pressure and cache reuse.

Modular arithmetic. We keep modular reduction out of the selector hot loop whenever possible. For P_1 and P'_1, a host-side bound verifies that the maximum bucket sum fits in a 64-bit accumulator, so threads use ordinary 64-bit multiply–adds and reduce only after completing the bucket. For $Q = 2^{32} - 12287$,used by P_4 and P_6, unreduced products cannot be accumulated safely. We exploit its pseudo-Mersenne form,

$2^{32} \equiv 12287 \pmod{Q}$: each product is folded once before accumulation, and the remaining folds are deferred until the bucket completes. A generic overflow-safe Barrett path is retained as fallback.

Fused external-product chain. After selector construction, one CUDA block executes the complete h'-bucket chain for each bootstrap. The evaluation-domain accumulator remains on chip across bucket iterations. Each iteration performs an inverse NTT, signed approximate gadget decomposition, forward NTTs of the ℓ digits, and pointwise multiply–accumulation with the merged BRK rows. A final inverse NTT produces the output. Fusing the chain removes inter-bucket kernel launches and global-memory round trips of the accumulator.

Warp-level NTT. For $N = 1024$, the 1024 threads are viewed as a 32×32 warp–lane array. The first five butterfly stages use warp shuffles, followed by one shared-memory transpose, after which the remaining five stages again become intra-warp. A stride-33 transpose avoids bank conflicts, so each transform requires only two block-wide synchronizations rather than one per stage. For $N = 2048$, each thread holds two virtual NTT positions; the first ten stages apply the same scheme to both streams and the final butterfly is register-local. For the three-level P'_1 configuration, we interleave the three independent digit NTTs within each thread to increase ILP and share their twiddle accesses and transpose synchronization.

Autotuning. We autotune batch-one latency and steady-state throughput independently. Throughput searches evaluation-domain merge tiles of 1–16 ciphertexts, 256/512/1024 merge threads, resident batches of 64–2048, and both BRK representations. All throughput winners use evaluation-domain BRKs and 256-thread merge blocks: $\mathsf{P}_1/\mathsf{P}'_1$ select tile 12 and resident batch 2048, while $\mathsf{P}_4/\mathsf{P}_6$ select tile 13 and resident batch 1024. Batch-one latency disables cross-ciphertext tiling; only P_4 selects the alternative coefficient-domain merge–NTT fusion.

Evaluation protocol. Batch-one latency uses resident and logical batch size one; each run is warmed up for 5,000 executions and measured over 20 executions. Throughput executes a logical workload of 16,384 bootstraps as repeated autotuned resident batches with reused buffers. After selecting the latency and throughput winners independently, we rerun each in five fresh processes and report the median; each throughput process measures 20 logical workloads.

E Application Details: NSA and AES

E.1 Details of 8-bit Instruction Set

Table 13 summarizes the semantics of the 39 instruction variants evaluated in Section 6.3. All operands and outputs are 8-bit values. Unless otherwise stated, arithmetic wraps modulo 2^8. Arithmetic, comparison, minimum, and maximum instructions use unsigned operands, except for `NEG` and `ABS`, which use 8-bit two's-complement semantics. Division and remainder instructions return zero when the divisor is zero. For a family denoted by `INSTR(i)`, `INSTR` takes two encrypted operands, whereas `INSTRi` takes one encrypted operand and one public immediate. Comparison instructions return an encrypted byte in $\{0,1\}$. The instruction nomenclature follows TFHE Processor [45], whereas their homomorphic realization uses our PBC backend.

E.2 Details of Homomorphic AES-128

Figure 3 details the per-round dataflow of the ten-round AES-128 evaluation of Section 7. Each round processes a 16-byte state that alternates between nibble and bit representations. Open boxes denote operations that require no bootstrapping, while filled boxes mark the three points at which a programmable bootstrapping is needed: the recomposition step, SubBytes, and the MVPBS bit decomposition. Consequently, bootstrapping is confined to the S-boxes and the two representation conversions, as stated in Section 7.

Within a round, the initial AddRoundKey is evaluated as a set of LWE additions. Recomposition is a programmable bootstrapping that maps the parity-bit state back to nibbles while performing a Horner refresh. SubBytes follows the 4-bit tree construction: each byte is split into two nibbles, and a single MVLUTeval produces both output nibbles of the S-box while sharing the first-level blind rotation. MVPBS then applies the reverse bit decomposition, returning the state to bit-LWE ciphertexts over $\mathrm{GF}(2)$, from which the affine layers ShiftRows (an index permutation), MixColumns, and AddRoundKey cost only LWE additions. During rounds $r = 1, \ldots, 9$ the state loops back from AddRoundKey to the recomposition step; in the final round $r = 10$ the dashed shortcut skips MixColumns, and the parity bits are output directly.

Bootstrapping budget. Our AES-128 evaluation requires $\approx 2{,}080$ programmable bootstrappings, as the Hippogryph implementation [8]. The initial recomposition costs $32 \times 4 = 128$; each full round costs 16×3 (S-box trees) $+ 32 \times 4$ (nibble recompositions) $+ 32$ (one shared rotation per MVPBS decomposition) $= 208$; and the final round costs $16 \times 3 + 32 = 80$. We optimize every component in isolation: a faster PBC-compiled NTRU blind rotation and an optimized NTRU packing technique, therefore, the resulting speedup over Hippogryph is component-level rather than structural.

Table 13: Semantics of the 39 evaluated 8-bit instruction variants.

Class	Instruction	Semantics
Arithmetic	`ADD(i)`	Addition modulo 256.
	`SUB(i)`	Subtraction modulo 256.
	`MUL(i)`	Least-significant byte of the unsigned product.
	`MULM(i)`	Most-significant byte of the 16-bit unsigned product of two 8-bit operands.
	`DIVi`	Unsigned division of an encrypted byte by a public byte.
	`DIV`	Unsigned division of one encrypted byte by another.
	`MOD4i`	Remainder of an encrypted byte modulo a public 4-bit nibble.
	`MOD`	Remainder of one encrypted byte modulo another encrypted byte.
Bitwise	`AND(i)`	Bitwise conjunction.
	`OR(i)`	Bitwise disjunction.
	`XOR(i)`	Bitwise exclusive-or.
	`SHLi`, `SHL`	Logical left shift by a public or encrypted index, respectively; indices greater than or equal to eight produce zero.
	`SHRi`, `SHR`	Logical right shift by a public or encrypted index, respectively; indices greater than or equal to eight produce zero.
	`ROL(i)`	Eight-bit rotation to the left, with the index reduced modulo 8.
	`ROR(i)`	Eight-bit rotation to the right, with the index reduced modulo 8.
Test	`EQ(i)`	Equality test.
	`LTEi`	Test whether an encrypted byte is less than or equal to a public byte.
	`LT`	Test whether the first encrypted byte is less than the second.
	`GT`	Test whether the first encrypted byte is greater than the second.
	`GTE`	Test whether the first encrypted byte is greater than or equal to the second.
Other	`MIN(i)`	Unsigned minimum of two encrypted bytes, or of an encrypted byte and a public byte.
	`MAX(i)`	Unsigned maximum of two encrypted bytes, or of an encrypted byte and a public byte.
	`CSEL`	Select the first encrypted byte when the encrypted predicate is one, and the second when it is zero.
	`NEG`	Two's-complement negation of an encrypted signed byte.
	`ABS`	Absolute value under 8-bit two's-complement semantics; in particular, `ABS(0x80)` returns `0x80`.

F NTRU Packing Modulo NTT Primes

As a state-of-the-art construction for homomorphic packing, the RevHomTrace method of Lee and Yoon [36] achieves both low noise growth and efficient amortized evaluation. Based on Chen et al. [15], it replaces the one-shot normalization by N^{-1} with a division by 2 per trace level, and evaluates HomTrace [15] in reverse order to eliminate the random noise introduced by modulus-raising from earlier divisions by 2. When applied to packing, this gives $O(N)$ external products to pack messages using N-dimensional RLWE, with noise variance at most $O(N \log N)$.

Our construction, however, requires an NTRU realization of RevHomTrace over an NTT-friendly prime modulus Q, rather than a power-of-two modulus. Directly switching an NTRU ciphertext to a power-of-two modulus is undesirable, because enforcing invertibility of f modulo a power of two restricts the admissible secret-key space and incurs at most a one-bit security loss under rejection sampling. Moreover, a power-of-two accumulator modulus would require an additional FFT backend solely for reverse-trace packing besides our NTT backend. We therefore give an NTRU-native reverse-trace packing algorithm that operates directly over R_Q and reuses the existing NTT backend.

Let $\tau_g : X \mapsto X^g$ denote the R_Q-automorphism. For each g, let $\mathsf{ATK}_g = \mathsf{NGSW}_f(f(X^g)/f, \sigma)$ and define $\mathsf{EvalAuto}_g(c) = \mathsf{ATK}_g \odot \tau_g(c)$, which evaluates τ_g and key-switches the resulting ciphertext from $f(X^g)$ back to $f(X)$. We present our NTRU RevHomTrace packing as in Algorithm 4, where each division by 2 and modulus raising in [36, Algorithm 6] is replaced by multiplication by $2^{-1} \bmod Q = (Q+1)/2$. Although multiplication by $2^{-1} \bmod Q$ may enlarge the centered noise of the intermediate ciphertext, this amplification is exactly canceled by the subsequent addition of its automorphism image, so the original noise is propagated with coefficient one. Each input of Algorithm 4 satisfies $(c_j f)_0 = u_j$, where we assume $u_j \in \mathbb{Z}_Q$ is a noisy term including the input error for simplicity; the remaining coefficients of $c_j f$ are arbitrary. For the convenience of proof, we use a non-recursive form of this algorithm.

Under the standard heuristic that the relevant input-error coefficients, decomposition errors, and effective errors of distinct EvalAuto calls are mutually independent, Theorem 4 gives the correctness and noise growth of Algorithm 4.

Theorem 4. *For any $0 \le k \le \log N$, Algorithm 4 correctly outputs $c = \mathsf{NTRU}(v, \sigma_{\mathsf{pack}})$, where $v_{jN/2^k} = u_j$ for $j \in [2^k]$,*

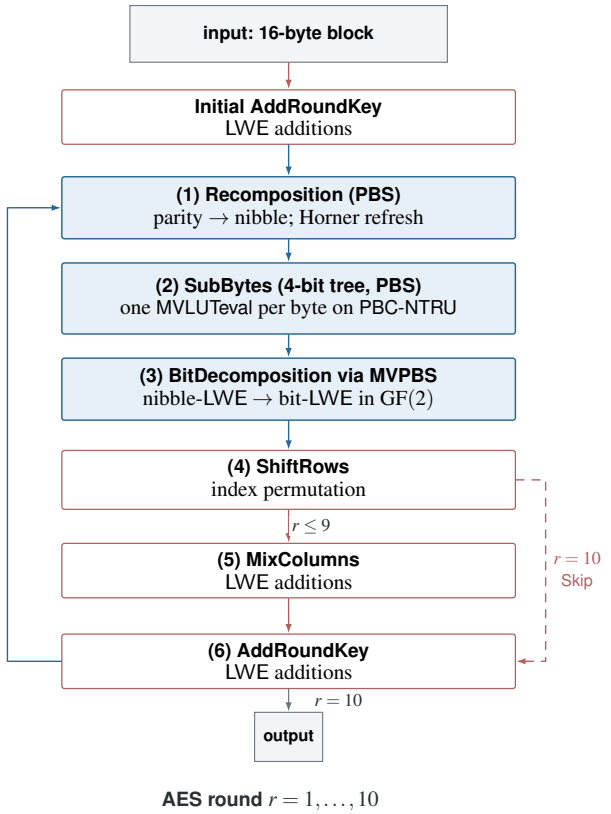

Figure 3: Per-round dataflow of our NSA-based AES evaluation, whose components are instantiations of the PBC-compiled NTRU PBS. Filled boxes mark the three PBS-based steps (Recomposition, SubBytes, and MVPBS BitDecomposition); all remaining layers cost only LWE additions. In the final round $r = 10$, the dashed shortcut skips MixColumns.

the remaining coefficients of v are arbitrary, and

$$\sigma^2_{\mathsf{pack}} \lesssim \sigma^2_{\mathsf{in}} + k \cdot \sigma^2_{\mathsf{auto}}.$$

In particular, $v = \sum_{j=0}^{N-1} u_j X^j$ for $k = \log N$. Here $\sigma^2_{\mathsf{auto}} = N\ell_{\mathsf{at}}\beta^2_{\mathsf{at}}\sigma^2/12 + \|f\|_2^2\sigma^2_{\mathfrak{g},\mathsf{at}}$.

Proof. For simplicity, we directly prove the full-packing case $k = \log N$; the same induction applies to any $k \le \log N$. We prove that after level d, for every $j \in [2^{k-d}]$ and $r \in [2^d]$,

$$(c_j^{(d)} f)_{rN/2^d} = u_{j+r2^{k-d}} + \delta_{j,r}^{(d)},$$

where $\delta_{j,r}^{(d)}$ contains one EvalAuto error coefficient from each of the first d levels. The claim for $d = 0$ is exactly $(c_j f)_0 = u_j$.

Suppose it holds after level $d-1$, and set $s = N/2^d$. For the EvalAuto call producing $c_j^{(d)}$, write

$$\mathsf{EvalAuto}_{2^d+1}(\widetilde{c}_j^{(d)}) f = \tau_{2^d+1}(\widetilde{c}_j^{(d)} f) + \varepsilon_j^{(d)}.$$

Algorithm 4: PackNTRUs

Input: NTRU ciphertexts c_j with $(c_j f)_0 = u_j$ and input-noise variance σ^2_{in}, for $j \in [2^k]$.

Auxiliaries : Automorphism keys ATK_{2^d+1} for $1 \le d \le k$.

Output: $c = \mathsf{NTRU}(v(X))$, where $v_{jN/2^k} = u_j$ for $j \in [2^k]$; for $k = \log N$, $v(X) = \sum_{j=0}^{N-1} u_j X^j$.

$c_j^{(0)} \leftarrow c_j$ for $j \in [2^k]$;
for $d = 1$ **to** k **do**
  for $j \in [2^{k-d}]$ **do**
    $\widetilde{c}_j^{(d)} \leftarrow 2^{-1}\left(c_j^{(d-1)} - X^{N/2^d} c_{j+2^{k-d}}^{(d-1)}\right)$;
    $c_j^{(d)} \leftarrow$
    $\widetilde{c}_j^{(d)} + \mathsf{EvalAuto}_{2^d+1}(\widetilde{c}_j^{(d)}) + X^{N/2^d} c_{j+2^{k-d}}^{(d-1)}$;
return $c = c_0^{(k)}$;

For $r \in [2^{d-1}]$, the induction hypothesis gives

$$(c_j^{(d-1)} f)_{2rs} = u_{j+2r2^{k-d}} + \delta_{j,r}^{(d-1)},$$
$$(c_{j+2^{k-d}}^{(d-1)} f)_{2rs} = u_{j+(2r+1)2^{k-d}} + \delta_{j+2^{k-d},r}^{(d-1)}.$$

The automorphism is a signed permutation of the coefficient basis, and

$$\tau_{2^d+1}(X^{2rs}) = X^{2rs}, \quad \tau_{2^d+1}(X^{(2r+1)s}) = -X^{(2r+1)s}.$$

Hence these coefficients are not mixed with any other coefficient. Since Q is odd, multiplication by 2^{-1} is exact, and the update gives

$$(c_j^{(d)} f)_{2rs} = (c_j^{(d-1)} f)_{2rs} + (\varepsilon_j^{(d)})_{2rs},$$
$$(c_j^{(d)} f)_{(2r+1)s} = (c_{j+2^{k-d}}^{(d-1)} f)_{2rs} + (\varepsilon_j^{(d)})_{(2r+1)s}.$$

These are precisely the claimed coefficients for indices $2r$ and $2r+1$. In particular, each old error is passed to the next level with coefficient one, while the current level adds one fresh EvalAuto error coefficient. This proves the induction. At $d = k$, only $c_0^{(k)}$ remains and

$$(c_0^{(k)} f)_{rN/2^k} = u_r + \delta_{0,r}^{(k)}, \quad r \in [2^k],$$

which proves correctness; for $k = \log N$, these are all coefficients.

Let σ_d^2 be the noise variance of a target coefficient after level d. The coefficient recurrence above gives

$$\sigma_d^2 \lesssim \sigma_{d-1}^2 + \sigma^2_{\mathsf{auto}}, \quad \sigma_0^2 = \sigma^2_{\mathsf{in}}.$$

Hence $\sigma_k^2 \lesssim \sigma^2_{\mathsf{in}} + k\sigma^2_{\mathsf{auto}}$, as claimed. □