Stream-BSGS: Memory-Efficient Tree-BSGS with Optimized Relinearization Scheduling for FHE Polynomial Evaluation

Zhongqi Wang[1][0009-0001-3459-5925], Shintaro Narisada[2][0000-0002-9399-9778], and Takashi Nishide[1][0000-0001-6518-8989]

[1] University of Tsukuba, Tsukuba, Japan
s2430135@u.tsukuba.ac.jp, nishide@risk.tsukuba.ac.jp
[2] KDDI Research, Inc., Fujimino, Japan
sh-narisada@kddi.com

Abstract. Polynomial evaluation is a fundamental primitive in RLWE-based fully homomorphic encryption (FHE) and underpins non-trivial applications such as bootstrapping and privacy-preserving machine learning. The dominant approach is the BSGS family of algorithms, including the Paterson–Stockmeyer (PS) method and tree-BSGS. In tree-BSGS, lazy relinearization reduces the number of expensive relinearization (key-switching) operations by deferring them across tree levels. While prior implementations were limited to a single level, the profitability of deeper deferral remains an open question. Extending relinearization beyond one level leads to enlarged intermediate ciphertexts and necessitates additional higher-order evaluation keys, transforming deeper lazy relinearization into a classic time–memory tradeoff.

This work advances lazy relinearization in tree-BSGS through two complementary contributions. First, we develop a noise-aware scheduling policy. Within the natural family of level-wise schedules, we prove that deferring single-step partial relinearization to the deepest noise-feasible level strictly minimizes the total relinearization count. Second, we propose Stream-BSGS, which fuses the leaf-generation and tree-reduction phases into a single streaming pass. By maintaining the same arithmetic structure as standard tree-BSGS, Stream-BSGS inherits the efficiency gains of our policy while substantially reducing the memory footprint, thereby significantly mitigating the memory overhead.

Experiments in OpenFHE across BFV, BGV, and CKKS show that Stream-BSGS achieves 1.66× to 1.91× wall-clock speedup over PS on high-degree polynomials, while reducing peak memory by 1.65× to 2.81×. With a smaller baby-step size, peak memory can be further reduced by 2.28× to 3.7×, while still retaining 1.49× to 1.90× speedup over PS.

Keywords: Fully homomorphic encryption · Polynomial evaluation · Baby-step giant-step · Lazy relinearization · Memory efficiency · BFV · BGV · CKKS.

1 Introduction

Fully homomorphic encryption (FHE) [16] enables arbitrary computation on encrypted data and is a foundational technology for privacy-preserving applications [2]. Among the three major RLWE-based FHE schemes, BFV [14], BGV [8], and CKKS [13], polynomial evaluation is a fundamental computational primitive. In BFV and BGV, non-linear functions over encrypted integers, including comparison, division, and modular reduction, must be realized through polynomial evaluation [19,15], since these schemes natively support only addition and multiplication. In CKKS, bootstrapping [11,26,7] requires the evaluation of a high-degree polynomial approximation of the modular reduction function; privacy-preserving machine learning [17,23,28,24] similarly relies on polynomial evaluation to approximate non-linear activation functions such as ReLU and sigmoid. Recent work on privacy-preserving transformer inference [31] further highlights the importance of efficient high-degree polynomial evaluation over encrypted data. The efficiency of polynomial evaluation therefore affects a wide range of FHE applications.

Polynomial evaluation in FHE is commonly based on the Baby-Step Giant-Step (BSGS) decomposition of Paterson and Stockmeyer [33]. For a polynomial of degree d, BSGS partitions the coefficients into $s \approx \sqrt{d}$ segments and computes baby-step powers of the encrypted input via approximately $b \approx \sqrt{d}$ ciphertext-ciphertext multiplications, with $b \cdot s \approx d$. Each segment is then evaluated as a leaf ciphertext through ciphertext-plaintext multiplications, and the leaves are combined using precomputed giant-step powers. The way leaves are combined distinguishes the main BSGS variants.

The standard approach, commonly referred to as the *PS algorithm* in FHE literature [20], precomputes all $s - 1$ giant-step powers and combines the leaves through a linear scan, requiring $s - 2$ additional ciphertext-ciphertext multiplications. *Tree-BSGS* [22,7,27] replaces this linear combination with a balanced binary tree reduction; only $\log_2 s$ giant-step powers need to be precomputed via repeated squaring, saving approximately $\sqrt{d}$ ciphertext-ciphertext multiplications.

1.1 Evaluation Dimensions and the Four-Dimensional Framework

In many settings, FHE polynomial evaluation algorithms are compared along three computational dimensions.

Ciphertext–ciphertext multiplication count. Ciphertext–ciphertext multiplication is typically among the most expensive basic operations in FHE. Each such multiplication incurs substantial computational cost and consumes noise budget, thereby reducing the remaining multiplicative depth available for subsequent operations. Reducing the total number of ciphertext–ciphertext multiplications is therefore a direct way to improve computational efficiency.

Multiplicative depth. The multiplicative depth of a polynomial evaluation algorithm determines the scale of the FHE parameters required to execute it. A larger depth generally requires a larger ciphertext modulus Q, and sometimes

Table 1. BSGS design space for FHE polynomial evaluation under degree d, baby-step size b, and leaf count s with $b \cdot s \geq d+1$ (balanced split: $b \approx s \approx \sqrt{d}$). Ct–ct mult and relin counts for tree-BSGS exclude the $\log_2 s - 1$ giant-step squarings shared with Phase 2 precomputation; PS's count already includes its $s-2$ giant-step multiplications.

Variant	ct–ct mults	Mult. depth	Relin. count	Peak working set
PS [33,20]	$b+2s-3$	$\lceil \log_2(d-\sqrt{d}) \rceil + 1$	$b+2s-3$	$b+2s-1$
Tree-BSGS [22,7,27]	$b+s-2$	$\log_2 d$	$b+s-2$	$b+s+\log_2 s$

a larger ring dimension N, to maintain both correctness and security. This increases ciphertext size and the cost of individual operations.

Relinearization count. Each ciphertext–ciphertext multiplication produces a higher-order ciphertext, with three or more polynomial components, which is usually reduced back to the standard 2-polynomial form through *relinearization*, also known as key switching. Relinearization is often one of the most expensive sub-operations in ciphertext multiplication, typically accounting for a large fraction of the total multiplication time. Reducing the number of relinearization operations can therefore yield substantial end-to-end speedup.

These three dimensions capture computation-side costs. Improvements along these dimensions often translate into better runtime or smaller FHE parameters, although tradeoffs between them may arise. Memory cost, however, has received less systematic treatment in comparisons of FHE polynomial evaluation algorithms. Recent work has shown that memory movement can be a major bottleneck in FHE computations [9], with ciphertext working sets often exceeding the last-level cache capacity of modern processors. Memory consumption is also a practical constraint when deploying FHE applications on commodity hardware [34,35].

Since leveled-FHE ciphertexts are large and structurally complex, a useful algorithm-level measure of memory usage during polynomial evaluation is the maximum number of ciphertexts that are simultaneously live during execution. We call this quantity the *peak working set*. Together with the three computational dimensions above, peak working set gives a *four-dimensional comparison framework* for polynomial evaluation algorithms: ciphertext–ciphertext multiplication count, multiplicative depth, relinearization count, and peak working set. This framework lets us compare algorithms from both computational and memory perspectives.

Table 1 applies this framework to the main variants of the BSGS family. PS has a peak working set of $b+2s-1$ (baby powers, giant powers, and leaves), while tree-BSGS reduces the giant-power table from $s-1$ to $\log_2 s$ entries through repeated squaring, achieving a peak of $b+s+\log_2 s$. Tree-BSGS is computationally superior: it reduces the ciphertext–ciphertext multiplication count from $b+2s-3$ to $b+s-2$ and lowers the multiplicative depth from $\lceil \log_2(d-\sqrt{d}) \rceil + 1$ to $\log_2 d$.

1.2 Lazy Relinearization in Tree-BSGS

The tree structure of tree-BSGS provides an additional optimization opportunity: *lazy relinearization*—the deferral of relinearization across tree levels to reduce the total count of expensive relinearization (key-switching) operations.

Existing tree-BSGS implementations [22,7,27,37] apply this technique conservatively, deferring relinearization by only a single level. At this depth, intermediate ciphertexts expand to 3-polynomial form only briefly, and no evaluation keys beyond the standard relinearization key are required, keeping the memory overhead minimal.

However, the profitability of *deeper* deferral has remained unclear. Each additional level of deferral maintains intermediate ciphertexts at higher polynomial orders (K-poly form for $K > 3$), which necessitates higher-order evaluation keys for subsequent reduction to 2-poly form. These factors collectively drive up memory consumption: multiple un-relinearized higher-order ciphertexts must be held in memory during tree reduction, and larger evaluation keys increase the static memory footprint. Consequently, deeper deferral introduces a *time–memory tradeoff*—reducing the relinearization count at the cost of significantly increased memory requirements. § 5.3 quantifies this tradeoff through an ablation study.

1.3 Our Contributions

We extend lazy relinearization beyond a single level through two complementary contributions: a scheduling analysis determining the optimal deferral depth, and a streaming reformulation designed to minimize peak memory consumption.

Contribution 1: Noise-aware relinearization scheduling (§ 3). In tree-BSGS, relinearization following each ciphertext-ciphertext multiplication can be deferred to a later tree level, reducing the total number of expensive key-switching operations. Existing tree-BSGS work [27,7,37] defers relinearization by exactly one tree level; whether deeper deferral is beneficial depends on how the resulting higher-order ciphertexts are relinearized.

We show that deeper deferral can be profitable, and address two questions that arise when deferral goes beyond one level. The analysis uses a cost asymmetry between key switching and tensor-product multiplication: under the BFV, BGV, and CKKS parameter regimes considered in this work, the marginal cost of one key-switching step exceeds that of one additional tensor-product term by a factor of $\alpha > 2$ (Proposition 8).

- **How to relinearize.** At one-level deferral, the deferred ciphertext has only one extra polynomial component, so there is only one way to relinearize. From two-level deferral onward, more than one extra component has accumulated, and two strategies become possible: *full relinearization* removes all extra components at once, while *partial relinearization* removes one component per subsequent multiplication. We prove that partial relinearization is strictly more efficient (Proposition 11).

– **How deep to defer.** Under partial relinearization, the total tree-reduction cost is strictly decreasing in the deferral depth k (Theorem 12), so the deferral should be as deep as the noise budget allows. We define k_{noise} as the deepest level for which the schedule still produces a correct output upon decryption, and prove that $k = k_{\text{noise}}$ is the optimal feasible policy within the level-wise schedule family (Theorem 18). k_{noise} is determined by $O(\log d)$ trial evaluations at the target parameters.

This also helps explain why prior tree-BSGS work has used only one-level deferral: under *full* relinearization, increasing the deferral depth from one to two levels can increase the tree-reduction cost. We call this the *second-level barrier*. This barrier is caused by the cost of full relinearization, rather than by deeper deferral itself; it disappears under partial relinearization, where deeper deferral becomes beneficial up to the noise-feasibility limit.

At CKKS bootstrapping parameters, the resulting policy delivers up to 42.0% speedup in the tree-combination stage over the non-lazy baseline and 10.8% over the one-level deferral of prior work (§ 5.1).

Contribution 2: Stream-BSGS (§ 4). While Contribution 1 improves computational performance through optimized scheduling, deeper lazy relinearization can also increase the memory cost of higher-order ciphertexts, making the time-memory tradeoff more acute. To reduce this memory cost, we propose Stream-BSGS, which fuses the leaf generation and tree reduction phases of standard tree-BSGS into a single streaming pass—equivalently, a depth-first (post-order) evaluation of the reduction tree: each leaf ciphertext is immediately folded into the tree reduction upon generation, and intermediate results are merged and released as soon as their sibling nodes become available. The peak working set drops from $b + s + \log_2 s$ (for standard tree-BSGS) to $b + 2\log_2 s + 2$. Stream-BSGS *preserves all arithmetic properties of tree-BSGS*: the ciphertext–ciphertext multiplication count, multiplicative depth, ciphertext–plaintext operation count, output value, and noise profile are all identical. It also inherits Contribution 1's optimized relinearization scheduling through a shared fold subroutine. Thus, Stream-BSGS removes the $O(s)$ live-leaf term that arises in standard tree-BSGS from storing all leaf ciphertexts before reduction, while retaining the computational savings from lazy relinearization.

Comparison with PS and tree-BSGS at fixed (b, s). Under the four-dimensional comparison framework, at the same baby/giant-step parameter split (b, s), Stream-BSGS is no worse than PS and existing tree-BSGS along every computational dimension (ciphertext–ciphertext multiplications, multiplicative depth, and relinearization count under the same scheduling policy) and strictly reduces the peak working set from $b + s + \log_2 s$ to $b + 2\log_2 s + 2$. Contribution 1's scheduling applies equally to both tree-BSGS and Stream-BSGS; however, Stream-BSGS can realize its computational benefit without incurring a corresponding memory penalty (§ 5.3). Because Stream-BSGS preserves the arithmetic structure of tree-BSGS, it can replace tree-BSGS at the leaf-generation and tree-reduction

stages without modifying the surrounding baby-step or giant-step precomputation. This applies equally to BFV, BGV, and CKKS.

Flexibility: further memory-computation trade-off. Since Stream-BSGS's peak working set depends primarily on b rather than s, halving b (and doubling s) further reduces peak memory usage at the expense of a marginal increase in wall-clock time. This optimization is less attractive for PS or standard tree-BSGS, where reducing b necessitates an increase in s, thereby elevating peak memory. A detailed analysis of this trade-off is provided in § 4.3.

Experiments. Full experiments on OpenFHE across all three schemes confirm that Stream-BSGS achieves 1.66× to 1.91× wall-clock speedup over PS on high-degree polynomials (degree 2^{14} through 2^{16}) at standard parameters, with 1.65× to 2.81× peak memory reduction. By reducing $\log b$ by one, peak memory drops by up to 3.7× while retaining 1.49× to 1.90× speedup over PS. An ablation study isolates the individual effects of the two contributions and empirically validates the time-memory tradeoff of deeper lazy relinearization: at the tested parameters, the scheduling of Contribution 1 alone typically changes end-to-end time by only a few percent, and it is the streaming execution of Contribution 2 that converts the memory reduction into the reported end-to-end gains.

2 Preliminaries and Related Work

2.1 FHE Schemes and Ciphertext Structure

We assume familiarity with the three major RLWE-based FHE schemes: BFV [14], BGV [8], and CKKS [13,12]. Let N be a power of two, $R = \mathbb{Z}[X]/(X^N + 1)$, and $R_q = R/qR$. A standard ciphertext is a pair $(c_0, c_1) \in R_q^2$ decrypting under a secret key s. BFV and BGV encrypt integers modulo a plaintext modulus t; CKKS encrypts approximate complex numbers with a scaling factor Δ. All three schemes share the same key-switching cost structure, which is analyzed in § 3.1.

A *K-polynomial ciphertext* has K components $(c_0, \ldots, c_{K-1})$. Multiplying a $(K-1)$-polynomial ciphertext by a (standard) 2-polynomial one via tensor product produces a K-polynomial ciphertext. Throughout the paper, $K_{\max}$ denotes the maximum polynomial count reachable during tree reduction; in a tree of height $H = \log_2 s$, this is $K_{\max} = H + 2$ (starting from 2-polynomial leaves, each of H levels adds one component). The maximum switch level is therefore H, but noise constraints invariably bind before this structural limit, as discussed in § 3.4.

2.2 Relinearization, Partial Relinearization, and Key-Switching Parameters

Full relinearization reduces a K-polynomial ciphertext back to 2-polynomial form by key-switching the higher components. We write $C_{\text{relin}}(K \to 2)$ for the cost of full relinearization from K-poly to 2-poly.

Single-step partial relinearization [37] reduces a K-polynomial ciphertext to $(K-1)$-polynomial form by key-switching only the highest component.

We write $C_{\text{mul}}(K)$ for the cost of an asymmetric no-relin multiplication $(K-1)$-poly $\times$ 2-poly $\to K$-poly, and define the marginal costs:

$$\Delta_{\text{mul}}(K) = C_{\text{mul}}(K) - C_{\text{mul}}(K-1), \quad \Delta_{\text{relin}}(K) = C_{\text{relin}}(K \to 2) - C_{\text{relin}}((K-1) \to 2).$$

§ 3.1 proves that Δ_{mul} and Δ_{relin} are structurally independent of K and establishes their ratio.

We write d_{num} for the number of key-switching digits and P for the number of special primes used in the hybrid key-switching procedure; L denotes the number of RNS primes in the ciphertext modulus q at the current level.

2.3 BSGS Polynomial Evaluation

For a polynomial $p(x) = \sum_{i=0}^{d} a_i x^i$ with $d+1 = b \cdot s$, BSGS uses baby-step size b and leaf count s (balanced split: $b \approx s \approx \sqrt{d}$). Tree-BSGS then proceeds in four phases.

Phase 1 (Baby step). Compute baby-power table $x, x^2, \ldots, x^b$ via balanced binary multiplication.

Phase 2 (Giant step). Compute giant-power table $z = x^b, z^2, z^4, \ldots, z^{2^{\log_2 s - 1}}$ by repeated squaring ($\log_2 s - 1$ squarings).

Phase 3 (Leaf generation). For each segment $j = 0, \ldots, s-1$, form a leaf ciphertext $y_j = \sum_{i=0}^{b-1} a_{jb+i} \cdot x^i$ via b ciphertext–plaintext multiplications.

Phase 4 (Tree reduction). Combine the s leaves bottom-up using giant powers in a balanced binary tree of height $H = \log_2 s$.

PS differs only in Phase 4, where the giant-step combination uses a linear scan instead of a tree, requiring $s-2$ additional ciphertext–ciphertext multiplications and resulting in a deeper multiplicative depth.

2.4 Peak Working Set

Definition 1 (Peak working set). *The* peak working set W_{peak} *of a ciphertext-domain algorithm is the maximum, over all program points during execution, of the number of ciphertexts that are simultaneously live. A ciphertext is* live *at a program point if it will be read again before it is freed.*

Proposition 2 (Peak working set of standard tree-BSGS). *Standard tree-BSGS with baby-step size b and leaf count s has peak working set*

$$W_{\text{peak}}^{\text{std}} = b + s + \log_2 s.$$

PS has peak working set $b + 2s - 1$: it requires $s-1$ precomputed giant powers in place of tree-BSGS's $\log_2 s$.

Proof. Phase 3 materializes all s leaf ciphertexts into a flat array before Phase 4 begins. The baby-power table holds b ciphertexts throughout Phases 3 and 4, and the giant-power table holds $\log_2 s$ ciphertexts (for tree-BSGS) or $s-1$ ciphertexts (for PS). At the program point immediately after the last leaf is formed and before any Phase 4 operation, all b baby powers, $\log_2 s$ giant powers, and s leaves are simultaneously live, giving a peak count of $b + s + \log_2 s$ for tree-BSGS. For PS, the $s - 1$ precomputed giant powers replace the $\log_2 s$ entry, giving $b + 2s - 1$. □

This $b + s + \log_2 s$ peak is not inherent to the arithmetic of the computation. Rather, it comes from *phase separation*: standard tree-BSGS first generates and stores all leaf ciphertexts, and only then starts the tree reduction. Since the reduction can consume each leaf as soon as it is generated, it is unnecessary to keep all s leaves live simultaneously. This observation motivates the streaming reformulation in § 4.

2.5 Related Work

Tree-BSGS and lazy relinearization. Han and Ki [22] introduced tree-structured BSGS for Chebyshev polynomial evaluation in CKKS. Bossuat et al. [7] adopted it for CKKS bootstrapping with one-level lazy relinearization. Lee et al. [27] further refined the bootstrapping pipeline, again deferring relinearization by one tree level. Wang et al. [37] applied the same one-level deferral strategy to bivariate function evaluation in BFV. These works use lazy relinearization as an optimization, but they do not analyze deeper deferral levels. They also do not study the peak working set of tree-BSGS, which is the memory cost addressed in this work.

BSGS in FHE applications. Beyond standalone polynomial evaluation, BSGS-family algorithms are used in many FHE applications. In privacy-preserving machine learning, polynomial evaluation via BSGS is used to approximate non-linear activation functions [17,23,28,24]. Recent privacy-preserving transformer inference frameworks [31] rely on high-degree polynomial evaluation for softmax, GELU, and LayerNorm approximations. CKKS and BFV/BGV bootstrapping [11,7,26,15,25,30,20,1] also rely on BSGS-based polynomial evaluation as a core subroutine. FHE-based private set intersection [10] and private information retrieval [3] further benefit from efficient polynomial evaluation. Thus improving the efficiency and reducing the memory footprint of BSGS algorithms affects many important FHE applications.

Asymmetric BSGS. Wang and Wang [36] proposed an asymmetric BSGS algorithm that reduces the relinearization count and the modulus-switching count from $O(\sqrt{d})$ to $O(d^{1/t})$ on PS's chain structure through base-B decomposition and chained tensor products, where $t \geq 3$ is a small constant trading tensor-product degree against switch reduction. Their experimental evaluation is conducted on BGV in HElib [21], with the BFV and CKKS cases discussed analytically in their appendices.

Their construction is built on the chain-structured combination of PS, and the extension to tree-structured BSGS is left open. This paper targets the latter setting: balanced binary tree reduction in tree-structured BSGS. Wang and Wang [36] optimize the chain-structured combination used in PS, whereas our work optimizes scheduling and execution order in tree-structured BSGS. The two directions are therefore complementary rather than directly overlapping.

Other homomorphic polynomial-evaluation approaches. PEGASUS [29] takes a different route to non-polynomial functions, switching between CKKS and FHEW ciphertexts to evaluate them via look-up tables instead of high-degree polynomial approximation. Okada, Player, and Pohmann [32] exploit the Galois automorphisms of the BFV/BGV plaintext algebra to evaluate a polynomial of degree d (bounded by the ring degree) with only $O(\log d)$ ciphertext–ciphertext multiplications and automorphism evaluations, beating the $2\sqrt{d}$ multiplication bound that applies when automorphisms are not available. Both rest on evaluation structures fundamentally different from coefficient-domain tree-BSGS, so neither our relinearization scheduling nor the streaming execution of Stream-BSGS transfers to them, nor their techniques to the BSGS family. Our methods instead apply uniformly across BFV, BGV, and CKKS—in particular to CKKS bootstrapping, whose approximate plaintext space lacks the finite Galois structure that [32] requires and where BSGS-style polynomial evaluation remains the core primitive.

Memory challenges in FHE. De Castro et al. [9] demonstrated that FHE computations can be strongly constrained by memory movement: ciphertext working sets do not fit in last-level caches of current processors, making memory bandwidth an important performance bottleneck. Rovida and Leporati [34] showed that memory is a critical constraint for deploying FHE-based neural network inference even on commodity hardware, proposing encoding strategies to reduce the memory footprint. Ünay et al. [35] developed compiler-level techniques to manage rotation-key memory in complex FHE programs. These works address memory at the architecture, system, encoding, or key-management level; our work is complementary, reducing memory at the *algorithmic* level by reducing the peak number of simultaneously live ciphertexts during polynomial evaluation.

3 Optimal Relinearization Scheduling

This section presents Contribution 1. The scheduling theory applies to any BSGS variant built on a balanced binary tree reduction with asymmetric (2-polynomial right operand) multiplication. Both standard tree-BSGS and Stream-BSGS of § 4 are instances. §4.1 shows that Stream-BSGS inherits the scheduling policy through a shared fold subroutine.

3.1 Structural Cost Asymmetry

We bound the marginal cost of one tensor-product term and one key-switching step, and derive the asymmetry ratio $\alpha = \Delta_{\text{relin}}/\Delta_{\text{mul}}$. The bounds depend only on the operation definitions and apply to BFV, BGV, and CKKS.

Throughout, we use *limb operations* as the unit of algorithmic cost. A limb operation is a degree-N coefficient-wise polynomial multiplication or NTT over a single RNS prime, costing $\Theta(N)$.

Marginal Tensor-Product Cost Consider an asymmetric no-relin multiplication $(K-1)$-poly $\times$ 2-poly $\to K$-poly. Let the left operand be $\mathbf{c} = (c_0, \ldots, c_{K-2}) \in R_q^{K-1}$ and the right operand $\mathbf{c}' = (c'_0, c'_1) \in R_q^2$. The tensor product produces the K-polynomial result $\mathbf{e} = (e_0, \ldots, e_{K-1})$ with

$$e_i = \sum_{j+\ell=i} c_j \cdot c'_\ell, \qquad i = 0, \ldots, K-1, \tag{1}$$

requiring $2(K-1)$ ring multiplications in R_q.

Proposition 3 (Tensor-product marginal). *The marginal cost of the tensor product when the left operand increases from $(K-2)$-poly to $(K-1)$-poly is exactly 2 ring multiplications in R_q, independent of K.*

Proof. Adding the component c_{K-2} to the left operand introduces exactly two new terms: $c_{K-2} \cdot c'_0$ (contributing to e_{K-2}) and $c_{K-2} \cdot c'_1$ (producing e_{K-1}). All other terms in (1) are unchanged. Each ring multiplication in R_q costs L limb operations. □

The no-relin multiplication of BFV additionally requires a $\lfloor t/q \rceil$ scaling step (the Bajard–Eynard–Hasan–Zucca (BEHZ) [5] or Halevi–Polyakov–Shoup (HPS) [18] procedure) that is cryptographically inseparable from the tensor product: without it, the result is not a valid BFV ciphertext. This step performs a per-component basis extension ($L \to L+1$ RNS primes and back), adding at most $c_S \cdot L$ limb operations per marginal component, where $c_S \leq 1$. For BGV and CKKS, the tensor product alone constitutes the complete no-relin multiplication (ModSwitch and Rescale are separate, optional operations), so $c_S = 0$.

Definition 4 (Marginal multiplication cost). $\Delta_{\text{mul}} \coloneqq (2 + c_S) \cdot L$ *limb operations, where $c_S = 0$ for BGV/CKKS and $c_S \leq 1$ for BFV. By Proposition 3, Δ_{mul} is independent of K.*

Marginal Key-Switching Cost A single-step partial relinearization from K-poly to $(K-1)$-poly key-switches the highest component c_{K-1} using the hybrid key-switching procedure with d_{num} digits and P special primes. The procedure consists of three stages:

(i) *ModUp*: extend c_{K-1} from L to $L+P$ RNS primes via CRT basis conversion, costing $\Theta(P \cdot L)$ limb operations.

(ii) *Inner product*: for each of d_{num} digits, multiply by the 2-component evaluation key in R_{QP}, producing $2d_{\text{num}}$ ring multiplications each over $L{+}P$ limbs, totaling $2d_{\text{num}}(L{+}P)$ limb operations.
(iii) *ModDown*: reduce from $L{+}P$ back to L primes via CRT basis conversion, costing $\Theta(P \cdot L)$ limb operations.

Proposition 5 (Key-switching marginal). *The cost of one partial-relinearization step is independent of K. Its dominant term is $2d_{\text{num}}(L{+}P) + 2PL$ limb operations.*

Proof. The key-switching procedure operates solely on c_{K-1}, which is a single element of R_q regardless of K. The digit-decomposition structure, evaluation-key format, and ModUp/ModDown procedure are all determined by the parameters $(N, q, d_{\text{num}}, P)$ and do not depend on K. The inner-product cost is $2d_{\text{num}}(L{+}P)$; ModUp and ModDown each contribute PL from CRT basis conversion, plus lower-order NTT terms. □

Definition 6 (Marginal relinearization cost). $\Delta_{\text{relin}} \coloneqq 2d_{\text{num}}(L{+}P){+}2PL$ *limb operations (dominant terms). By Proposition 5, Δ_{relin} is independent of K.*

Remark 7. Key switching is a ciphertext-domain operation: it acts on R_q-elements via digit decomposition and evaluation-key multiplication, with no dependence on the plaintext encoding (the plaintext modulus t of BFV/BGV or the scaling factor Δ of CKKS). Consequently, Δ_{relin} is identical across all three schemes for the same ciphertext parameters (N, q, d_{num}, P).

Cost Asymmetry Combining Definitions 4 and 6 yields the cost asymmetry ratio used by our scheduling analysis. The form of this ratio follows from the standard cost model of hybrid key switching [22]; we restate it here in the form needed for the rest of the paper.

Proposition 8 (Structural cost asymmetry). *For any RLWE-based FHE scheme (BFV, BGV, or CKKS) using hybrid key switching with d_{num} digits and P special primes, the cost asymmetry ratio*

$$\alpha \;\coloneqq\; \frac{\Delta_{\text{relin}}}{\Delta_{\text{mul}}} \;\geq\; \frac{2d_{\text{num}}(L+P) + 2PL}{(2+c_S)\,L},$$

where $c_S \leq 1$. In particular:

(a) $\alpha > 1$ for all $d_{\text{num}} \geq 1$, $P \geq 1$.
(b) $\alpha > 2$ for all $d_{\text{num}} \geq 3$, $P \geq 2$ (equivalently, $d_{\text{num}} \geq 3$ with $P = d_{\text{num}}$, the default in hybrid key switching), which covers all practical parameter sets with multiplicative depth ≥ 5.

Proof. Taking the most conservative denominator ($c_S = 1$):

$$\alpha \;\geq\; \frac{2d_{\text{num}}(L{+}P) + 2PL}{3L} \;=\; \frac{2d_{\text{num}}}{3}\Big(1 + \frac{P}{L}\Big) + \frac{2P}{3}.$$

For $d_{\text{num}} \geq 3$, $P \geq 2$: $\alpha \geq 2(1 + 2/L) + 4/3 > 10/3 > 2$. For $d_{\text{num}} \geq 1$, $P \geq 1$: $\alpha \geq 2/3 \cdot (1 + 1/L) + 2/3 > 1$. □

Remark 9 (Other key-switching variants). The bound $\alpha \geq d_{\text{num}} + d_{\text{num}}^2/L$ (for $c_S = 0$, $P = d_{\text{num}}$) is a *lower* bound for hybrid key switching. Under GHS key switching ($d_{\text{num}} = 1$, $P \approx L$), the ratio becomes $\alpha \approx L \gg 2$. Under BV key switching, the number of digits is $d_{\text{num}} = \lceil \log q \rceil \gg 3$. Thus hybrid key switching actually yields the *smallest* α among the three standard variants; the cost asymmetry only strengthens under alternative key-switching methods.

Terminology. Throughout the remainder of this section, we use $c_{\text{TP}} \coloneqq \Delta_{\text{mul}}$ and $c_{\text{KS}} \coloneqq \Delta_{\text{relin}}$ to emphasize the structural constancy of the marginal costs. All cost expressions below are stated in terms of these two constants and the cost asymmetry ratio $\alpha = c_{\text{KS}}/c_{\text{TP}}$.

3.2 Strategy Framework and Optimal Relinearization Depth

In tree-BSGS, $s = 2^H$ leaf ciphertexts are combined bottom-up over H tree levels. At each level ℓ (with $\ell = 0$ at the leaves), adjacent nodes are paired; the right node is multiplied by a precomputed 2-polynomial giant-step power, and the result is added to the left node. We denote the number of pairs at level ℓ by $n_\ell = s/2^{\ell+1}$.

We parameterize the relinearization policy by a single integer k, the *switch level*:

> **Strategy** $H(k)$**:** Skip relinearization at tree levels $0, 1, \ldots, k-1$ (Stage I). Resume relinearization at every level from k onward (Stage II).

$H(0)$ is the non-lazy baseline; $H(1)$ is the one-level deferral used in prior work [27,7]. When relinearization resumes in Stage II, a second parameter arises: the *relinearization depth* $r \in \{1, \ldots, k\}$, specifying how many key-switching steps to perform ($r = 1$: partial, one step; $r = k$: full, all the way to 2-poly).

Proposition 10 (Polynomial-count propagation). *In strategy $H(k)$ with $k \geq 1$, the polynomial count at the output of level ℓ is*

$$P_\ell = \begin{cases} \ell + 3, & \text{if } 0 \leq \ell < k, \\ k + 2, & \text{if } k \leq \ell \leq H - 1. \end{cases}$$

That is, the polynomial count grows by one per level during Stage I, then remains permanently fixed at $(k+2)$ throughout Stage II, regardless of the relinearization depth $r \geq 1$.

Proof. Stage I (levels 0 to $k-1$, no relinearization): At level 0, both leaves are 2-polynomial; the multiplication $2 \times 2 \to 3$ and the addition $2+3 \to 3$ give $P_0 = 3$. At each subsequent level $\ell < k$, the right operand is $P_{\ell-1}$-polynomial, which when multiplied by a 2-polynomial giant power yields $(P_{\ell-1} + 1)$-polynomial; adding this to the $P_{\ell-1}$-polynomial left gives $P_\ell = P_{\ell-1} + 1 = \ell + 3$.

Stage II (levels k to $H - 1$, relinearization resumes): Base case ($\ell = k$): the right operand has count $k + 2$ before relinearization. After a depth-r relinearization it becomes $(k + 2 - r)$-polynomial; the subsequent multiplication

yields $(k+3-r)$-polynomial; adding to the $(k+2)$-polynomial left gives $P_k = \max(k+2, k+3-r) = k+2$ since $r \geq 1$. Induction gives $P_{\ell+1} = k+2$ whenever $P_\ell = k+2$. □

Optimal relinearization depth. Proposition 10 establishes that every Stage II right operand arrives at relinearization with exactly $(k+2)$ polynomial components. The per-pair Stage II cost is a function of relinearization depth $r \in \{1, \ldots, k\}$: one performs r key-switching steps (cost $r \cdot c_{\mathrm{KS}}$) to reduce the polynomial count, then multiplies at the reduced count (cost $C_{\mathrm{mul}}(k-r+3)$).

Proposition 11 (Optimality of single-step relinearization). *Under cost asymmetry ($\alpha > 1$, guaranteed by Proposition 8), the per-pair Stage II cost is strictly increasing in r. That is, $r = 1$ (single-step partial relinearization) is strictly optimal and $r = k$ (full relinearization) is strictly the worst.*

Proof. Each additional key-switching step saves c_{TP} on the subsequent multiplication (by lowering the polynomial count by one) but costs c_{KS}. Since $c_{\mathrm{KS}} > c_{\mathrm{TP}}$ (i.e., $\alpha > 1$), each additional step incurs a net cost of $c_{\mathrm{KS}} - c_{\mathrm{TP}} > 0$. □

The second-level barrier. When full relinearization ($r = k$) is used and the deferral is pushed from $k = 1$ to $k = 2$, the end-to-end tree-reduction cost increases rather than decreases. This phenomenon, which we call the *second-level barrier*, is a direct consequence of Proposition 11: full relinearization is the worst strategy in the relinearization-depth dimension, and its penalty grows with k, causing $H(2)$ under full relin to become more expensive than $H(1)$. Under partial relinearization ($r = 1$), the barrier disappears, as the following section shows.

All subsequent analysis is conducted under the optimal $r = 1$. The per-pair Stage II cost under this choice is

$$A = c_{\mathrm{KS}} + C_{\mathrm{mul}}(k+2),$$

and the total tree-reduction cost of strategy $H(k)$ is

$$C^{r=1}(k) = \underbrace{\sum_{\ell=0}^{k-1} n_\ell \cdot C_{\mathrm{mul}}(\ell+3)}_{\text{Stage I}} + A \cdot (2n_k - 1) + k \cdot c_{\mathrm{KS}},$$

where $n_\ell = s/2^{\ell+1}$ and the last term accounts for the root relinearization (k partial steps from $(k+2)$-poly back to 2-poly).

3.3 Switch-Level Monotonicity under $r = 1$

Since the marginal costs are structural constants (Definitions 4 and 6), we write c_{TP} and c_{KS} in place of Δ_{mul} and Δ_{relin} throughout. The marginal cost of deepening the switch level from k to $k+1$ under $r = 1$ is:

$$\Delta C^{r=1}(k) = n_k \cdot (2\, c_{\mathrm{TP}} - c_{\mathrm{KS}}) + (c_{\mathrm{KS}} - c_{\mathrm{TP}}). \tag{2}$$

Theorem 12 (Switch-level monotonicity under $r = 1$). *Under cost asymmetry ratio $\alpha > 2$ (guaranteed by Proposition 8 for all $d_{\text{num}} \geq 3$), $C^{r=1}(k)$ is strictly decreasing in k for all n_k exceeding the threshold*

$$n_k^* = \frac{\alpha - 1}{\alpha - 2}.$$

In particular, for $\alpha \geq 4$ (the typical regime): $n_k^* = 3/2$, so $C^{r=1}(k)$ is strictly decreasing for every tree with $s \geq 4$ ($n_k \geq 2$).

Proof. In (2), the coefficient of n_k is $2\,c_{\text{TP}} - c_{\text{KS}} = c_{\text{TP}}(2 - \alpha) < 0$ since $\alpha > 2$. The constant term is $c_{\text{KS}} - c_{\text{TP}} = c_{\text{TP}}(\alpha - 1) > 0$. Setting $\Delta C^{r=1}(k) = 0$ and solving: $n_k^* = (\alpha - 1)/(\alpha - 2)$. For $n_k > n_k^*$, the negative n_k term dominates and $\Delta C^{r=1}(k) < 0$. □

Remark 13. Equation (2) is the constant-marginal specialization of a more general formula that allows Δ_{mul} and Δ_{relin} to vary with K. In the general case, the monotonicity condition becomes

$$2\Delta_{\text{relin}}(k{+}2) - \Delta_{\text{relin}}(k{+}3) > 2\Delta_{\text{mul}}(k{+}3), \qquad (\star)$$

which reduces to $c_{\text{KS}} > 2\,c_{\text{TP}}$ (i.e., $\alpha > 2$) when the marginals are constant.

Remark 14 (Optimality of the threshold structure). The threshold structure is provably optimal within the class of level-wise schedules. By Theorem 12, extending Stage I is always profitable. Conversely, skipping relinearization at any level $\ell \geq k$ within Stage II permanently elevates the polynomial count from $k{+}2$ to $k{+}3$ (Proposition 10), requiring one extra c_{KS} per pair at every subsequent level and at the root—a cascade cost of $c_{\text{KS}} \cdot n_\ell$. Since the saving at level ℓ is only $n_\ell(c_{\text{KS}} - c_{\text{TP}})$, the net effect is $-n_\ell\, c_{\text{TP}} < 0$: the skip is strictly dominated.

3.4 Noise-Aware Optimal Policy

Deeper deferral increases the polynomial count of intermediates during Stage I, which raises the noise accumulated through tensor products. In CKKS, this manifests as continuous loss of output precision; in BFV and BGV, it manifests as outright decryption failure when the noise budget is exhausted. In both cases, a noise ceiling limits the feasible deferral depth. This section introduces a noise-feasibility constraint which is applicable for all three schemes and combines it with the cost monotonicity of §3.3 to obtain the optimal policy.

Proposition 15 (Noise monotonicity). *For all three schemes, the noise (or precision loss) incurred by strategy $H(k)$ is monotonically non-decreasing in k.*

Proof. $H(k{+}1)$ defers relinearization one level deeper than $H(k)$, so Stage I contains one additional tree level of un-relinearized (higher-polynomial-count) multiplication. Each such multiplication introduces a multiplicative noise growth factor strictly greater than 1, since the tensor product of noisy polynomials

amplifies the error. Therefore the noise at the boundary between Stage I and Stage II is strictly larger under $H(k{+}1)$ than under $H(k)$, and Stage II operates on the same tree structure in both strategies. For BFV/BGV, this means the noise budget remaining after $H(k{+}1)$ is strictly smaller. For CKKS, the accumulated L_∞ decryption error is strictly larger. □

Definition 16 (Noise-feasible switch level). *For any FHE scheme with parameter set Π, target degree d, and tree size s, define*

$$k_{\text{noise}}(\Pi, d, s) \;\coloneqq\; \max\big\{k \geq 0 : H(k) \text{ is noise-feasible}\big\},$$

where $H(k)$ is noise-feasible *if it produces correct decryption (BFV/BGV) or output precision above an application-specific threshold B (CKKS).*

Corollary 17. *By Proposition 15, the noise-feasible set $\{k : H(k)$ is acceptable$\}$ is an initial segment $\{0, 1, \ldots, k_{\text{noise}}\}$.*

Determination of k_{noise} The value of k_{noise} depends on the scheme, the parameter set, and the target polynomial degree d (which determines how much noise budget is consumed by the baby-step and giant-step precomputation phases before tree reduction begins). It is determined by a *trial evaluation* at the target parameters:

- *BFV and BGV*: evaluate $H(k)$ for $k = H, H{-}1, \ldots$ until the first noise-feasible k.
- *CKKS*: evaluate $H(k)$ for $k = H, H{-}1, \ldots$ until the first k whose output precision exceeds the application-specific threshold B.

Since $H = O(\log d)$, this requires at most $O(\log d)$ trial evaluations, each being a single polynomial evaluation at the target degree.

Theorem 18 (Noise-aware optimal policy). *Under $\alpha > 2$ (guaranteed by Proposition 8 for $d_{\text{num}} \geq 3$), the $H(k)$ policy that minimizes the total tree-reduction cost subject to noise feasibility is $H(k^*)$ with $r = 1$, where*

$$k^* = k_{\text{noise}}.$$

Proof. By Theorem 12, $C^{r=1}(k)$ is strictly decreasing in k for all feasible tree sizes, so the cost-minimizing k in any feasible subset is its maximum element. By the corollary of Proposition 15, the noise-feasible set is the initial segment $\{0, \ldots, k_{\text{noise}}\}$, with $k_{\text{noise}} < H$ at any practical FHE parameter set (noise constraints bind before the tree-height limit, since higher-order key switching and tensor products (no-relin multiplications) accumulate noise that strictly exceeds the fresh-ciphertext baseline). The cost-optimal feasible k is therefore $k^* = k_{\text{noise}}$. □

Corollary 19 (Practical value). *The optimal scheduling is fully characterized by two results that hold for all RLWE-based FHE parameter sets with hybrid key switching ($d_{\text{num}} \geq 3$): (1) partial relinearization $r = 1$ is strictly optimal (Proposition 11); (2) maximal deferral to the noise ceiling within the threshold schedule is strictly optimal (Theorem 12; see also Remark 14). The remaining parameter k_{noise} is determined by $O(\log d)$ lightweight trial evaluations.*

Algorithm 1 TreeReduceHybrid: policy $H(k^*)$ with single-step partial relinearization.

Require: Node list nd[0..|nd| − 1], giant powers $G[0..H-1]$, switch level k^*, noise ceiling $K_{\max}$
Ensure: Reduced ciphertext in standard 2-polynomial form

$\ell \leftarrow 0$
while |nd| > 1 **do**
  nd′ ← [ ]
  for all consecutive pair (left, right) in nd **do**
    $K_r \leftarrow$ PolyCount(right)
    if ($\ell \geq k^*$ and $K_r > 2$) or ($K_r + 1 > K_{\max}$ and $K_r > 2$) **then**
      if $K_r = 3$ **then**
        right ← Relinearize(right)
      else
        PartialRelinInPlace(right) ▷ $r = 1$
      end if
    end if
    prod ← EvalMultNoRelin(right, $G[\ell]$)
    nd′.append(EvalAdd(left, prod))
  end for
  if |nd| is odd **then**
    nd′.append(nd[last])
  end if
  nd ← nd′; $\ell \leftarrow \ell + 1$
end while
while PolyCount(nd[0]) > 2 **do**
  if PolyCount(nd[0]) = 3 **then**
    nd[0] ← Relinearize(nd[0])
  else
    PartialRelinInPlace(nd[0])
  end if
end while
return nd[0]

Pseudocode. Algorithm 1 formalizes the combined strategy $H(k^*)$ with $r = 1$.

4 Stream-BSGS

§3 analyzed relinearization scheduling to reduce the cost of tree reduction. However, deeper lazy relinearization amplifies the memory cost: higher-order ciphertexts occupy more memory, and the tree-level dependencies force them to remain simultaneously live. This section introduces Stream-BSGS, which removes the s-scale term from the peak working set while preserving the arithmetic properties and scheduling of tree-BSGS.

Algorithm 2 PushAndFold: stack-based fold with inherited partial lazy relinearization.

Require: Pending stack stk, new leaf newLeaf, giant powers $G[0..H-1]$, switch level k^*, noise ceiling $K_{\max}$

```
cur ← newLeaf;  curLevel ← 0
while stk is non-empty and top of stk has level curLevel do
    (·, left) ← pop(stk)
    K_r ← PolyCount(cur)
    if (curLevel ≥ k* and K_r > 2) or (K_r + 1 > K_max and K_r > 2) then
        if K_r = 3 then
            cur ← Relinearize(cur)
        else
            PartialRelinInPlace(cur)
        end if
    end if
    prod ← EvalMultNoRelin(cur, G[curLevel])
    cur ← EvalAdd(left, prod)
    curLevel ← curLevel + 1
end while
push(stk, (curLevel, cur))
```

4.1 Algorithm

Key idea. We fuse Phases 3 and 4 of standard tree-BSGS into a single streaming pass. As each leaf ciphertext is generated, it is immediately passed to a stack-based fold routine that combines it with any previously produced leaves at the matching tree level. The stack holds at most one ciphertext at each tree level simultaneously, so its depth never exceeds $H + 1$ where $H = \log_2 s$.

PushAndFold routine. Algorithm 2 describes the fold routine. It inherits the scheduling policy of Algorithm 1: when a pair is formed at tree level ℓ, the right operand is relinearized (if $\ell \geq k^*$ and $K_r > 2$, or if the noise ceiling would be exceeded) using single-step partial relinearization. This is the mechanism by which Stream-BSGS automatically inherits Contribution 1's optimal scheduling.

Main streaming loop. Algorithm 3 is the top-level Stream-BSGS procedure. Phases 1 and 2 (baby and giant powers) are unchanged. The fused Phase 3/4 replaces the flat materialization of all leaves with streaming: generate the next leaf, call PushAndFold, repeat. After all s leaves have been streamed, the stack holds exactly one entry at level $\log_2 s$, which is the root.

Relation to classical streaming reductions. The stack-based binary-counter fold pattern is a classical streaming reduction technique in parallel computing [6]; in tree terms, Stream-BSGS performs a depth-first (post-order) evaluation of the reduction tree. Our contribution is to apply this pattern to FHE BSGS leaf generation, where standard tree-BSGS first generates and stores all leaf ciphertexts

Algorithm 3 StreamBSGS: streaming polynomial evaluation.

Require: Coefficients $a_0, \ldots, a_d$, encrypted input ct_x, baby-step size b, leaf count s with $b \cdot s = d + 1$, switch level k^*, noise ceiling $K_{\max}$
Ensure: Ciphertext encrypting $p(x) = \sum_{i=0}^{d} a_i x^i$

Phase 1: Compute baby powers $\mathrm{bp}[1] = \mathrm{ct}_x, \ldots, \mathrm{bp}[b]$
Phase 2: Compute giant powers $G[0] = \mathrm{bp}[b], G[1] = G[0]^2, \ldots, G[\log_2 s - 1]$
Fused Phase 3/4: $\mathrm{stk} \leftarrow \emptyset$
for $j = 0$ to $s - 1$ **do**
    $\mathrm{leaf}_j \leftarrow \sum_{i=0}^{b-1} a_{jb+i} \cdot \mathrm{bp}[i]$ ▷ b ct–pt multiplications
    PushAndFold$(\mathrm{stk}, \mathrm{leaf}_j, G, k^*, K_{\max})$
end for
Let $(\log_2 s, \mathrm{root})$ be the single remaining entry of stk
while PolyCount(root) > 2 **do**
    if PolyCount(root) $= 3$ **then**
        root $\leftarrow$ Relinearize(root)
    else
        PartialRelinInPlace(root)
    end if
end while
return root

before a level-order reduction. This yields the peak-working-set bound proved below and allows the relinearization scheduling of Contribution 1 to be used within the same fold subroutine.

4.2 Correctness, Peak Working Set, and Properties

Proposition 20 (Correctness). *For any coefficients and any choice of $(k^*, K_{\max})$, Stream-BSGS produces a ciphertext that decrypts to the same value as standard tree-BSGS with Algorithm 1 in Phase 4.*

Proof (Sketch). The two algorithms perform the same arithmetic on the same operands; only the execution order differs. Induction on H: after pushing 2^i leaves, PushAndFold has combined them into a single ciphertext at level i matching the corresponding subtree root of Algorithm 1. □

Theorem 21 (Peak working set of Stream-BSGS). *Stream-BSGS on $s = 2^{\log_2 s}$ leaves with baby-step size b has peak working set*

$$W_{\mathrm{peak}}^{\mathrm{Stream}} \leq b + 2\log_2 s + 2.$$

Proof. The baby-power table contributes b live ciphertexts throughout the fused pass (they are freed only after the last leaf is generated). The giant-power table $G[0..\log_2 s - 1]$ contributes $\log_2 s$ live ciphertexts throughout the fused pass. When j leaves have been pushed, the stack holds one entry per set bit of j; the stack size equals the popcount of j, which is at most $\log_2 s$ for $j < s$. During the inner loop of PushAndFold, at most two additional ciphertexts are simultaneously live: the popped left operand and the current fold result. The total peak is at most $b + \log_2 s + \log_2 s + 2 = b + 2\log_2 s + 2$. □

Table 2. Four-dimensional comparison at the same (b, s) split ($b \approx s \approx \sqrt{d}$). Under $H(k^*)$ with r=1, the tree-reduction relin count is $(s/2^{k^*} - 1) + k^*$; the exact value depends on k^* and s. Ct–ct mult and relin counts exclude Phase 2 precomputation ($\log_2 s - 1$ squarings for tree-BSGS/Stream-BSGS; $s - 2$ multiplications for PS). Bold entries indicate dimensions where Stream-BSGS strictly improves over all other variants.

	ct–ct mults	Depth	Relin. count	Peak working set
PS	$b + 2s - 3$	$\lceil \log_2(d - \sqrt{d}) \rceil + 1$	$b + 2s - 3$	$b + 2s - 1$
Tree-BSGS	$b + s - 2$	$\log_2 d$	$b + s - 2$	$b + s + \log_2 s$
Tree-BSGS + $H(k^*)$	$b + s - 2$	$\log_2 d$	$b + O(s/2^{k^*})$	$b + s + \log_2 s$
Stream-BSGS + $H(k^*)$	$b + s - 2$	$\log_2 d$	$\mathbf{b + O(s/2^{k^*})}$	$\mathbf{b + 2\log_2 s + 2}$

Preservation of algorithmic properties. As an exact reordering of standard tree-BSGS under the $H(k^*)$ policy, Stream-BSGS preserves the ciphertext–plaintext operation count $(d + 1)$, ciphertext–ciphertext multiplication count $(b + s - 2)$, multiplicative depth $(\log_2(d + 1))$, output value (Proposition 20), and noise-growth profile. The optimal scheduling of Contribution 1 is inherited through the shared relinearization branch in Algorithms 1 and 2.

4.3 Four-Dimensional Comparison and Adaptability

Improvement. Table 2 summarizes the comparison under the same baby/giant-step split (b, s). Under the same relinearization scheduling policy, Stream-BSGS has the same ciphertext–ciphertext multiplication count, multiplicative depth, and relinearization count as tree-BSGS, while reducing the peak working set from $b + s + \log_2 s$ to $b + 2\log_2 s + 2$. The "Tree-BSGS + $H(k^*)$" row separates the effect of Contribution 1 from that of Stream-BSGS: the scheduling policy $H(k^*)$ reduces the relinearization count for both tree-BSGS and Stream-BSGS. However, applying $H(k^*)$ to phase-separated tree-BSGS does not reduce its peak working set, which remains $b+s+\log_2 s$. Moreover, the higher-order intermediate ciphertexts and the corresponding key-switching material required by deeper lazy relinearization can increase the measured memory consumption, as observed in the ablation study (§ 5.3).

Against PS, Stream-BSGS improves all four dimensions. Against tree-BSGS under the same scheduling policy, the ciphertext–ciphertext multiplication count, depth, and relinearization count are identical, while the peak working set is reduced from $b + s + \log_2 s$ to $b + 2\log_2 s + 2$.

Flexibility: b-halved variant. In Stream-BSGS, the peak working set depends mainly on b with only a logarithmic dependence on s. This enables a further time-memory tradeoff: reducing $\log b$ by one (halving b, doubling s). The baby-step phase then requires approximately $\sqrt{d}/2$ fewer multiplications, but the doubled leaf count adds approximately $\sqrt{d}$ tree-reduction multiplications, for a net increase of approximately $\sqrt{d}/2$. Because Contribution 1's scheduling reduces

the relinearization overhead of the additional multiplications, the actual wall-clock increase is small. Under PS or standard tree-BSGS, the same reduction of b would cause s to grow, but since their peak working set includes s (or $2s$ for PS) as a dominant term, the memory would increase rather than decrease. Stream-BSGS breaks this coupling.

Adaptability across schemes. Stream-BSGS preserves the arithmetic structure of tree-BSGS. It can therefore be integrated into an FHE implementation by replacing only the leaf-generation and tree-reduction stages with the streaming loop, while leaving the baby-step and giant-step precomputation unchanged. This applies to BFV, BGV, and CKKS.

5 Experimental Evaluation

5.1 Scheduling Validation: CKKS Chebyshev Tree Combination

The Chebyshev tree-combination subroutine is a major cost component within the modular-reduction step of CKKS bootstrapping [11,7,26,30]. This subroutine satisfies both structural assumptions of our scheduling policy (balanced binary tree reduction with asymmetric multiplication) and provides a natural setting for evaluating Contribution 1. We validate the policy using bootstrapping-realistic CKKS parameters ($N = 65536$, depth 20, scaling modulus size 59, first modulus size 60); switch levels $k = 0$ through $k = 4$ (polynomial counts up to $K = 6$) are tested. The end-to-end validation for all three schemes follows in §5.2. Tree sizes range from $s = 16$ to $s = 1024$.

Result 1: the second-level barrier under full relinearization. Column "r=2 vs $H(1)$" of Table 3 shows that $H(2)$ under full relinearization ($r = k = 2$) is consistently slower than $H(1)$ across all tested tree sizes. The slowdown is 27.5% at $s = 16$ and stabilizes at approximately 30% for $s \geq 32$. This provides empirical evidence for the second-level barrier discussed in §3.2: naively deepening lazy relinearization from $k = 1$ to $k = 2$ without switching to partial relinearization produces a strict cost increase. This is consistent with Proposition 11, which shows that full relinearization is the least efficient among the relinearization-depth options analyzed in this work.

Result 2: partial relinearization eliminates the barrier. Under $r = 1$, the last two columns of Table 3 show $H(2)$ achieving 42.0% speedup over $H(0)$ and 10.8% over $H(1)$ at convergence ($s \geq 256$), turning $H(2)$ from "30% worse than $H(1)$" into "10% better than $H(1)$" — a swing of approximately 40% attributable solely to the relinearization-depth choice.

Result 3: precision ceiling binds at $k = 2$. Table 4 reports time and output precision for $H(0)$ through $H(4)$ at $s = 256$.

$H(3)$ and $H(4)$ yield negligible additional time savings (under 2%) but lose 5 and 14 bits of precision, respectively. The practical optimum is $k^* = k_{\text{noise}} = 2$,

Table 3. Tree-reduction time on the Chebyshev tree subroutine. $H(2)$ is reported under both full relinearization (r=k=2, demonstrating the second-level barrier) and partial relinearization (r=1, demonstrating its elimination). Negative speedup denotes slowdown. CKKS, $N = 65536$, depth 20.

	Time (s)				Speedup of $H(2)$		
s	$H(0)$	$H(1)$	$H(2)_{r=2}$	$H(2)_{r=1}$	r=2 vs $H(1)$	r=1 vs $H(0)$	r=1 vs $H(1)$
16	1.988	1.344	1.714	1.268	−27.5%	+36.2%	+5.7%
32	4.072	2.687	3.514	2.495	−30.8%	+38.7%	+7.1%
64	8.222	5.416	7.106	4.909	−31.2%	+40.3%	+9.4%
128	16.670	10.875	14.142	9.734	−30.0%	+41.6%	+10.5%
256	33.270	21.644	28.123	19.296	−29.9%	+42.0%	+10.8%
512	66.646	43.297	56.106	38.638	−29.6%	+42.0%	+10.8%
1024	133.579	86.911	112.961	77.448	−30.0%	+42.0%	+10.9%

Table 4. Time vs. precision at $s = 256$ under $r = 1$. CKKS, $N = 65536$, depth 20.

Strategy	Time (s)	Precision (bits) Status
$H(0)$	33.27	31.83 admissible
$H(1)$	21.64	31.83 admissible
$H(2)$	19.30	31.83 admissible
$H(3)$	19.07	$26.94 \approx 5$ bits loss
$H(4)$	19.08	$18.01 \approx 14$ bits loss

confirming the prediction of Theorem 18: the noise ceiling binds well before the tree-height limit ($H \geq 3$). The cost function $C^{r=1}(k)$ drops steeply from $k = 0$ to $k = 2$ and then plateaus, so the precision constraint does not sacrifice any measurable speedup.

5.2 End-to-End Evaluation

Platform and parameters. All experiments run on Linux (AMD Ryzen 9 7950X, 16 cores, 128 GB DDR5, Ubuntu) using OpenFHE v1.2.3 [4]. The three schemes share ring dimension $N = 32768$ and multiplicative depth 17. BFV and BGV use plaintext modulus $t = 65537$. CKKS uses scaling modulus size $\log \Delta = 50$ and first modulus size 60. All experiments use hybrid key switching with $d_{\text{num}} = 3$ digits and $P = 3$ special primes, placing every evaluated configuration in case (b) of Proposition 8; the cost-asymmetry ratio therefore satisfies $\alpha > 10/3$ throughout. We evaluate polynomial degrees 2^{14}, 2^{15}, and 2^{16}, comparing three configurations:

- **PS**: the standard Paterson–Stockmeyer algorithm.
- **Stream-BSGS**: our algorithm with Contribution 1's optimal scheduling ($k^* = k_{\text{noise}}$, determined by trial evaluation: $k^* = 6$ for BFV/BGV, $k^* = 2$ for CKKS at these parameters).

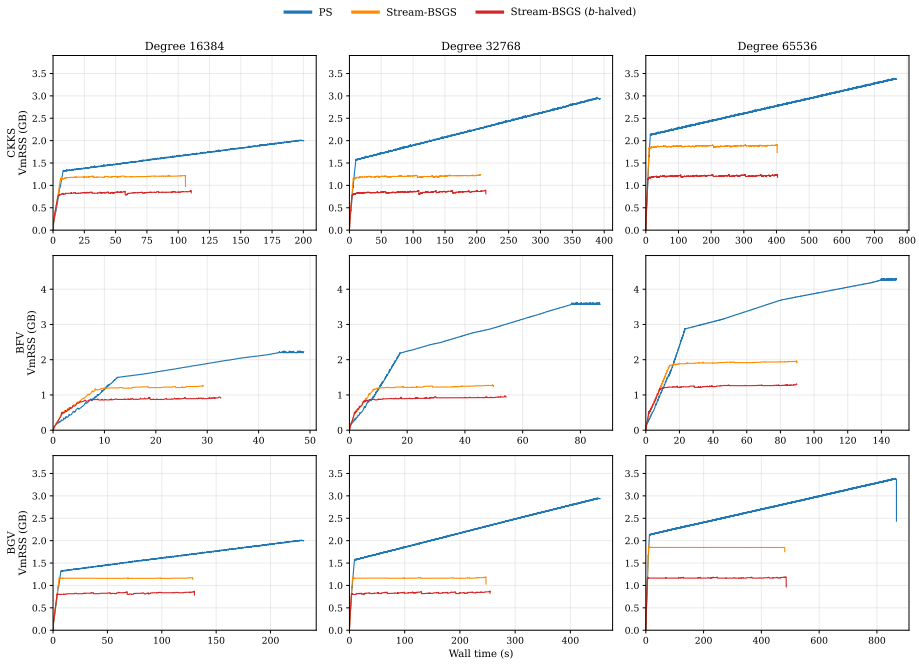

Fig. 1. Resident memory (VmRSS) over wall time for BFV, BGV, and CKKS at degrees 2^{14}, 2^{15}, 2^{16} ($N = 32768$, depth 17). Each panel compares PS, Stream-BSGS, and Stream-BSGS (b-halved).

– **Stream-BSGS (b-halved)**: $\log b$ reduced by one, demonstrating the further time-memory tradeoff.

Correctness of every run is verified against a plaintext-domain computation. Our implementation, benchmark harness, and parameter files are publicly available at `https://github.com/WayneOnEarth/Stream-BSGS` (tag `v1.0.0`, commit `9f0e93f`). All reported timings and peak resident set size (RSS) values are averaged over 5 independent runs[3].

Wall-clock and memory profile. Figure 1 makes both the wall-clock and memory behaviors directly visible (Table 5 records the precise values). PS exhibits a monotonically growing heap during Phase 3 that plateaus once all leaves are materialized, consistent with the $b+2s-1$ live-ciphertext profile of Proposition 2; Stream-BSGS instead maintains a flat profile throughout its fused streaming pass, consistent with the $b + 2\log_2 s + 2$ profile of Theorem 21; the b-halved

[3] The peak working set counts live ciphertext objects, whereas peak resident set size (RSS) measures the process-level memory footprint observed by the operating system. `VmRSS` is read from `/proc/self/status` by a background sampler at 100 ms intervals; peak RSS is the maximum sample over the run.

Table 5. End-to-end polynomial evaluation. $N = 32768$, depth 17. Peak RSS measured via VmRSS sampling. Speedup and RSS reduction are relative to PS.

Scheme	Degree	Method	Time (s)	Speedup	Peak RSS	RSS Red.
CKKS	2^{14}	PS	200.3	—	2.01 GB	—
		Stream-BSGS	105.9	1.89×	1.22 GB	1.65×
		b-halved	110.5	1.81×	0.88 GB	2.28×
	2^{15}	PS	393.5	—	2.97 GB	—
		Stream-BSGS	206.0	1.91×	1.24 GB	2.40×
		b-halved	214.1	1.84×	0.90 GB	3.32×
	2^{16}	PS	767.3	—	3.39 GB	—
		Stream-BSGS	402.2	1.91×	1.91 GB	1.78×
		b-halved	403.2	1.90×	1.25 GB	2.72×
BFV	2^{14}	PS	48.8	—	2.25 GB	—
		Stream-BSGS	29.2	1.67×	1.27 GB	1.77×
		b-halved	32.6	1.49×	0.94 GB	2.39×
	2^{15}	PS	86.8	—	3.63 GB	—
		Stream-BSGS	49.9	1.74×	1.29 GB	2.81×
		b-halved	54.3	1.60×	0.98 GB	3.70×
	2^{16}	PS	149.0	—	4.31 GB	—
		Stream-BSGS	89.8	1.66×	1.97 GB	2.19×
		b-halved	89.7	1.66×	1.32 GB	3.28×
BGV	2^{14}	PS	230.5	—	2.01 GB	—
		Stream-BSGS	128.5	1.79×	1.18 GB	1.71×
		b-halved	130.1	1.77×	0.87 GB	2.31×
	2^{15}	PS	453.8	—	2.95 GB	—
		Stream-BSGS	247.3	1.84×	1.19 GB	2.48×
		b-halved	254.7	1.78×	0.88 GB	3.37×
	2^{16}	PS	867.6	—	3.39 GB	—
		Stream-BSGS	481.0	1.80×	1.87 GB	1.81×
		b-halved	486.0	1.79×	1.19 GB	2.84×

variant sits at an even lower level. Stream-BSGS achieves 1.66× to 1.91× speedup over PS (Table 5), and the b-halved variant trades near-zero (degree 2^{16}) to approximately 10% (BFV, degree 2^{14}) wall-clock cost for additional memory reduction while retaining 1.49× to 1.90× speedup over PS.

5.3 Ablation Study

We isolate the individual effects of the two contributions through an ablation study. We compare four configurations under the same parameters as §5.2:

– **Tree-BSGS**: standard tree-BSGS with full relinearization at every tree level ($H(0)$).

– **Tree-BSGS (partial-relin)**: standard tree-BSGS with Contribution 1's optimal scheduling ($H(k^*)$, $r = 1$, where $k^* = k_{\text{noise}} = 6$ for BFV/BGV and $k^* = k_{\text{noise}} = 2$ for CKKS at these parameters, per Theorem 18), retaining the phase-separated execution of standard tree-BSGS.
– **Stream-BSGS**: our streaming algorithm with the same scheduling as the previous variant.
– **Stream-BSGS (b-halved)**: $\log b$ reduced by one for further memory reduction.

The first two configurations differ only in the relinearization scheduling (Contribution 1); the second and third differ only in the execution order (Contribution 2, streaming vs. phase-separated). In particular, Tree-BSGS (partial-relin) and Stream-BSGS execute exactly the same ciphertext-level arithmetic operations (identical ct–ct multiplication count, relinearization count, and multiplicative depth); the sole difference is whether leaf generation and tree reduction are fused into a streaming pass or performed in separate phases.

Figure 2 visualizes the VmRSS trace over wall time; the corresponding wall-clock time and peak RSS are reported in Table 6.

The time-memory tradeoff. Applying Contribution 1's optimal scheduling to standard tree-BSGS (the "partial-relin" configuration) demonstrates the time-memory tradeoff of §1.2 in quantitative terms. The VmRSS traces of the two phase-separated Tree-BSGS variants in Figure 2 make the BSGS phase structure directly visible: an initial ramp corresponding to baby-step and giant-step precomputation, followed by the dominant leaf-generation phase, and finally a descending tree-reduction phase. These traces reveal three distinct signatures of the tradeoff. First, during the leaf generation phase, the partial-relin trace runs parallel to but above the Tree-BSGS trace, reflecting the larger key-switching keys required for higher-order ciphertext relinearization. Second, during tree reduction, the partial-relinearization trace exhibits a steeper slope on the wall-time axis, confirming that the optimized scheduling accelerates the reduction phase. Third, the vertical gap between the two traces remains visible throughout tree reduction in BFV and BGV, reflecting the larger per-ciphertext memory footprint of higher-order intermediates that lazy relinearization keeps alive; in CKKS, this gap is opposite in sign because the standard Tree-BSGS exhibits a transient peak at the start of tree reduction (the burst of $s/2$ back-to-back full-relinearization calls each performs OpenFHE's hybrid key-switching with scratch buffers that are not reclaimed between calls), an effect avoided by partial relinearization which uses key-switching-free multiplications at the bottom tree levels.

In the non-streaming variants, end-to-end runtime is dominated by the leaf-generation phase. Leaf generation (ciphertext–plaintext scalar multiplications) accounts for approximately 70% to 98% of the total wall-clock time across all non-streaming configurations, far exceeding the tree-reduction phase that the scheduling optimization directly targets. Because the enlarged working set increases memory-subsystem pressure during this dominant phase, the tree-reduction speedup is substantially diluted, and in extreme cases offset, by the

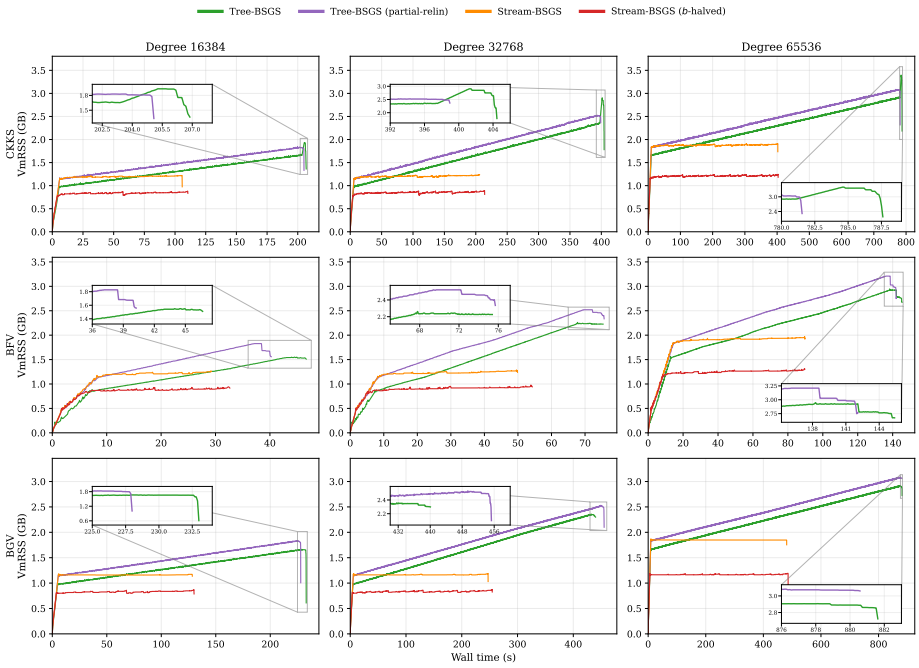

Fig. 2. Ablation study: resident memory (VmRSS) over wall time ($N = 32768$, depth 17). Each panel compares Tree-BSGS, Tree-BSGS (partial-relin), Stream-BSGS, and Stream-BSGS (b-halved). During leaf generation, the partial-relin trace is shifted upward by the larger key-switching keys; during tree reduction, it descends more steeply (faster reduction) but at a higher absolute RSS (larger intermediates). Stream-BSGS maintains a flat, low RSS profile throughout.

increased memory pressure during leaf generation. The CKKS and BGV cases further illustrate this dilution: the noise ceiling binds at $k^* = 2$ for CKKS (vs. $k^* = 6$ for BFV/BGV), leaving only the bottom two tree levels un-relinearized. Although the VmRSS traces (Figure 2) clearly show a steeper descent during tree reduction, confirming the scheduling acceleration, the resulting tree-reduction speedup is too small relative to the dominant leaf-generation phase to register in end-to-end wall-clock time. This is consistent with the observation of de Castro et al. [9].

Stream-BSGS resolves the tradeoff. Stream-BSGS executes exactly the same ciphertext-level arithmetic as Tree-BSGS (with partial-relin) but fuses the two phases into a streaming pass. In every configuration, Stream-BSGS simultaneously achieves lower wall-clock time *and* lower peak RSS than both Tree-BSGS variants (Table 6). Across all nine configurations, Stream-BSGS achieves 1.38× to 1.94× speedup and 1.44× to 2.13× peak-RSS reduction over Tree-BSGS (partial-relin), despite executing exactly the same ciphertext-level arith-

Table 6. Ablation study at degree 2^{15} ($N = 32768$, depth 17). Speedup and RSS reduction are relative to Tree-BSGS. RSS reduction $< 1\times$ indicates *increased* memory consumption. Full results for all degrees are in the artifact repository.

Scheme	Method	Time (s)	Speedup	Peak RSS	RSS Red.
CKKS	Tree-BSGS	404.5	—	2.90 GB	—
	(partial-relin)	399.0	1.01×	2.53 GB	1.15×
	Stream-BSGS	206.0	1.96×	1.24 GB	2.34×
	b-halved	214.1	1.89×	0.90 GB	3.22×
BFV	Tree-BSGS	75.4	—	2.26 GB	—
	(partial-relin)	75.7	1.00×	2.52 GB	0.90×
	Stream-BSGS	49.9	1.51×	1.29 GB	1.75×
	b-halved	54.3	1.39×	0.98 GB	2.31×
BGV	Tree-BSGS	440.0	—	2.36 GB	—
	(partial-relin)	455.3	0.97×	2.53 GB	0.93×
	Stream-BSGS	247.3	1.78×	1.19 GB	1.98×
	b-halved	254.7	1.73×	0.88 GB	2.68×

metic. For instance at CKKS degree 2^{15}, Stream-BSGS achieves 1.94× speedup (399.0 → 206.0 s) while reducing peak RSS by 2.04× (2.53 → 1.24 GB). Since the arithmetic operation counts are identical, the wall-clock difference is consistent with the reduction of memory pressure. We do not claim a more specific runtime mechanism in the absence of micro-architectural measurements. The streaming execution order keeps the peak working set small enough to improve memory-hierarchy efficiency not only during tree reduction, where the algorithmic benefit is direct, but also during the dominant leaf generation phase, where the reduced ambient memory pressure yields an additional secondary benefit.

Ablation summary. The ablation study decomposes the end-to-end performance of Stream-BSGS along the contribution chain. Contribution 1 (scheduling) improves the tree-reduction subroutine but, in the phase-separated setting of standard tree-BSGS, induces a memory penalty that can reduce the wall-clock gain. Contribution 2 (streaming) removes the memory penalty, allowing the scheduling speedup of Contribution 1 to translate into end-to-end wall-clock improvement. Together, they produce the simultaneous reduction in both wall-clock time and peak memory reported in Table 5 across all schemes and degrees. The results also validate the importance of peak working set as an evaluation dimension for FHE algorithms: the memory cost of an algorithm affects both the peak RSS through memory-subsystem effects and the wall-clock time of computationally dominant phases.

6 Conclusion

We optimized tree-BSGS along both computational and memory dimensions. Within level-wise threshold schedules $H(k)$, single-step partial relinearization

($r = 1$) at the deepest noise-feasible switch level $k^* = k_{\text{noise}}$ is optimal under the cost ratio $\alpha = \Delta_{\text{relin}}/\Delta_{\text{mul}} > 2$, which holds uniformly across BFV, BGV, and CKKS at standard parameters. Stream-BSGS fuses leaf generation and tree reduction into a streaming pass, reducing the peak working set from $b+s+\log_2 s$ to $b + 2\log_2 s + 2$ at unchanged arithmetic counts; the ablation confirms that this memory reduction is what lets the scheduling speedup translate into end-to-end improvement, by removing the memory pressure during leaf generation that otherwise prevents the speedup from registering.

Future work. The noise and precision impact of deferred relinearization remains theoretically uncharacterized. For BFV and BGV, the noise growth through higher-order key switching depends on library-specific modulus-chain management, making it difficult to establish a unified analytical model for k_{noise}. For CKKS, the precision degradation mechanism through tensor products involves data-dependent error amplification and requires a fundamentally different analysis. Developing scheme-specific analytical models for these phenomena would deepen the theoretical understanding of lazy relinearization and potentially enable a priori prediction of k_{noise} without trial evaluation.

Two further directions remain. Our optimality result is confined to the level-wise family $H(k)$; whether per-node schedules outside this family can improve on $H(k_{\text{noise}})$ is open. Integrating Stream-BSGS into complete bootstrapping pipelines (§5.1 already covers the tree-combination stage at CKKS bootstrapping parameters) would quantify the end-to-end gains in that setting.

Acknowledgments. We thank the anonymous SAC 2026 reviewers for their constructive comments. This work was supported in part by JSPS KAKENHI Grant Number 25K03116 and JST SPRING Grant Number JPMJSP2124.

References

1. Al Badawi, A., Polyakov, Y.: Demystifying bootstrapping in fully homomorphic encryption. Cryptology ePrint Archive, Paper 2023/149 (2023)
2. Albrecht, M., Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., Halevi, S., Hoffstein, J., Laine, K., Lauter, K., Lokam, S., Micciancio, D., Moody, D., Morrison, T., Sahai, A., Vaikuntanathan, V.: Homomorphic encryption security standard (2018), homomorphicEncryption.org, Toronto, Canada
3. Angel, S., Chen, H., Laine, K., Setty, S.T.V.: PIR with compressed queries and amortized query processing. In: IEEE Symposium on Security and Privacy (S&P). pp. 962–979. IEEE (2018)
4. Badawi, A.A., Bates, J., Bergamaschi, F., Cousins, D.B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., Kim, A., Lee, Y., Liu, Z., Micciancio, D., Quah, I., Polyakov, Y., R.V., S., Rohloff, K., Saylor, J., Suponitsky, D., Triplett, M., Vaikuntanathan, V., Zucca, V.: OpenFHE: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915 (2022)
5. Bajard, J.C., Eynard, J., Hasan, M.A., Zucca, V.: A full RNS variant of FV like somewhat homomorphic encryption schemes. In: SAC. LNCS, vol. 10532, pp. 423–442. Springer (2016)

6. Blelloch, G.E.: Prefix sums and their applications. In: Reif, J.H. (ed.) Synthesis of Parallel Algorithms, pp. 35–60. Morgan Kaufmann (1990), also available as Carnegie Mellon University technical report CMU-CS-90-190
7. Bossuat, J.P., Mouchet, C., Troncoso-Pastoriza, J., Hubaux, J.P.: Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In: EUROCRYPT. LNCS, vol. 12696, pp. 587–617. Springer (2021)
8. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory **6**(3), 1–36 (2014)
9. de Castro, L., Agrawal, R., Yazicigil, R., Chandrakasan, A., Vaikuntanathan, V., Juvekar, C., Joshi, A.: Does fully homomorphic encryption need compute acceleration? Cryptology ePrint Archive, Paper 2021/1636 (2021)
10. Chen, H., Laine, K., Rindal, P.: Fast private set intersection from homomorphic encryption. In: CCS. pp. 1243–1255. ACM (2017)
11. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: Bootstrapping for approximate homomorphic encryption. In: EUROCRYPT. pp. 360–384 (2018)
12. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A full RNS variant of approximate homomorphic encryption. In: SAC. LNCS, vol. 11349, pp. 347–368. Springer (2019)
13. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: ASIACRYPT. LNCS, vol. 10624, pp. 409–437. Springer (2017)
14. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144 (2012)
15. Geelen, R., Vercauteren, F.: Bootstrapping for BGV and BFV revisited. Journal of Cryptology **36**(2), 12 (2023)
16. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: STOC. pp. 169–178 (2009)
17. Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., Wernsing, J.: CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In: ICML. pp. 201–210 (2016)
18. Halevi, S., Polyakov, Y., Shoup, V.: An improved RNS variant of the BFV homomorphic encryption scheme. In: CT-RSA. LNCS, vol. 11405, pp. 83–105. Springer (2019)
19. Halevi, S., Shoup, V.: Algorithms in HElib. In: CRYPTO. pp. 554–571 (2014)
20. Halevi, S., Shoup, V.: Bootstrapping for HElib. In: EUROCRYPT. LNCS, vol. 9056, pp. 641–670. Springer (2015)
21. Halevi, S., Shoup, V.: Bootstrapping for HElib. Journal of Cryptology **34**(1), 7 (2021)
22. Han, K., Ki, D.: Better bootstrapping for approximate homomorphic encryption. In: CT-RSA. pp. 364–390. Springer (2020)
23. Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.: GAZELLE: A low latency framework for secure neural network inference. In: USENIX Security Symposium. pp. 1651–1669 (2018)
24. Kim, D., Guyot, C.: Optimized privacy-preserving CNN inference with fully homomorphic encryption. IEEE Transactions on Information Forensics and Security **18**, 2175–2187 (2023)
25. Kim, J., Seo, J., Song, Y.: Simpler and faster BFV bootstrapping for arbitrary plaintext modulus from CKKS. In: CCS. pp. 2535–2546. ACM (2024)

26. Lee, J.W., Lee, E., Lee, Y., Kim, Y.S., No, J.S.: High-precision bootstrapping of RNS-CKKS homomorphic encryption using optimal minimax polynomial approximation and inverse sine function. In: EUROCRYPT. LNCS, vol. 12696, pp. 618–647. Springer (2021)
27. Lee, Y., Lee, J.W., Kim, Y.S., Kim, Y., No, J.S., Kang, H.: High-precision bootstrapping for approximate homomorphic encryption by error variance minimization. In: EUROCRYPT. LNCS, vol. 13275, pp. 551–580. Springer (2022)
28. Lou, Q., Jiang, L.: HEMET: A homomorphic-encryption-friendly privacy-preserving mobile neural network architecture. In: ICML. vol. 139, pp. 7102–7110 (2021)
29. Lu, W., Huang, Z., Hong, C., Ma, Y., Qu, H.: PEGASUS: Bridging polynomial and non-polynomial evaluations in homomorphic encryption. In: IEEE Symposium on Security and Privacy (S&P). pp. 1057–1073. IEEE (2021)
30. Min, S., Lee, J.W., Song, Y.: Enhanced CKKS bootstrapping with generalized polynomial composites approximation. In: AsiaCCS. ACM (2025)
31. Moon, J., Yoo, D., Jiang, X., Kim, M.: THOR: Secure transformer inference with homomorphic encryption. In: CCS. ACM (2025)
32. Okada, H., Player, R., Pohmann, S.: Homomorphic polynomial evaluation using galois structure and applications to BFV bootstrapping. In: ASIACRYPT. LNCS, vol. 14443, pp. 69–100. Springer (2023)
33. Paterson, M.S., Stockmeyer, L.J.: On the number of nonscalar multiplications necessary to evaluate polynomials. SIAM Journal on Computing **2**(1), 60–66 (1973)
34. Rovida, L., Leporati, A.: Encrypted image classification with low memory footprint using fully homomorphic encryption. International Journal of Neural Systems **34**(05), 2450025 (2024)
35. Ünay, E., Franke, B., Woodruff, J.: KeyMemRT compiler and runtime: Unlocking memory-scalable FHE. arXiv:2601.18445 (2026)
36. Wang, Q., Wang, L.P.: A novel asymmetric BSGS polynomial evaluation algorithm under homomorphic encryption. In: AsiaCCS. ACM (2025)
37. Wang, Z., Fujita, H., Narisada, S., Nishide, T.: Improving homomorphic evaluation of bivariate functions in BFV via tree-based BSGS with lazy relinearization. In: International Symposium on Computing and Networking (CANDAR). pp. 181–187. IEEE (2025)