Generalized Batched Decomposition Key-Switching for CKKS

Antonio J. Peña, Zaira Pindado, and Hugo Sanz-González

Barcelona Supercomputing Center (BSC)
{antonio.pena, zaira.pindado, hugo.sanzgonzalez}@bsc.es

Abstract. In lattice-based homomorphic encryption schemes, ciphertexts carry an inherent noise that grows as computation advances. The key-switching procedure of these schemes allows the homomorphic re-encryption of a ciphertext under a new secret key, but it incurs substantial noise growth and computational cost. In this work, we generalize the RNS-based decomposition technique of the CKKS scheme to an arbitrary number of input polynomials and secret keys. Performing multiple relinearizations and rotations in batch enables one to hoist the modulus lowering and decomposition steps of key-switching, respectively, which restricts noise growth and avoids redundant computation. We prove the security and correctness of all the proposed algorithms, and derive explicit noise bounds for them.

As an application, we introduce a ternary multiplication algorithm that relinearizes the non-affine components of a product of three ciphertexts in batch, and performs rescaling in two phases so that relinearization runs at a lower modulus level. Compared with the usual sequence of two binary multiplications, this algorithm introduces a single key-switching error term, improving precision by up to half a bit and reducing latency by up to 1.31×. We generalize to n-ary ciphertext multiplication via a polynomial-time algorithm that combines binary and ternary multiplications into a tree of optimum depth minimizing a given latency cost function. Such products arise, for instance, in the SHIP bootstrapping algorithm; experiments with 128-bit-secure parameters on top of the Lattigo library show an average speedup of 1.20× over the standard approach for $n = 192$ with 32 threads. Further examples illustrate the impact of these optimizations on private deep neural network inference.

In addition, we provide—to the best of our knowledge—the first noise analysis of the double-hoisted BSGS matrix-vector multiplication algorithm of Bossuat *et al.* (2021), a building block of state-of-the-art CKKS bootstrapping circuits. We discuss a novel optimization concerning the automorphism applications during its giant steps.

Keywords: Fully Homomorphic Encryption · Key-Switching · Gadget Decomposition · Privacy-preserving Machine Learning

1 Introduction

Fully Homomorphic Encryption (FHE) [19] is a cryptographic primitive that enables direct computation on encrypted data, without intermediate decryptions. Thanks to the appearance of many open-source implementations [4, 17, 39], FHE has become a building block of various privacy-preserving solutions, and in particular of private deep neural network (DNN) inference [34].

Key-switching. FHE schemes that are secure under the Learning with Errors (LWE) [29, 35] assumption or its ring variant (RLWE) [37] contain an inherent error that grows with homomorphic computations; error growth needs to be kept under control to guarantee a correct decryption. A large amount of research has focused both on reducing the noise growth and on improving the computational efficiency of homomorphic operations. Key-switching (KS) converts a ciphertext under a secret key into an equivalent ciphertext under another secret key. Many homomorphic operations require key-switching in order to cancel the effect of encrypted arithmetic on the decryption circuit. Key-switching is the main bottleneck and noise contributor of homomorphic multiplication and slot rotation.

Several approaches have been proposed to improve the performance and to reduce the noise of key-switching in lattice-based FHE schemes. Gentry *et al.* [20] perform KS over an extended modulus QP, and then lower the result back to Q, thus dividing the KS error by the auxiliary P. Residue number systems (RNS) improve computational efficiency by representing large integers modulo Q as tuples of residues modulo smaller coprime integers $q_0, \ldots, q_L$ whose product is Q. Cheon *et al.* [11] introduced a full-RNS variant of the CKKS scheme [13]. Gadget decompositions reduce the noise introduced during key-switching by decomposing ring elements into shorter d-dimensional vectors with smaller entries. Bajard *et al.* [7] proposed an RNS-friendly digit decomposition for the FV scheme; and Han-Ki [24] adapted it to the CKKS scheme, and combined it with the temporary modulus technique for further noise control. Kim *et al.* [27] expressed the resulting *hybrid* key-switching method as an external product of the evaluation key (a $2 \times d$ matrix) and a gadget decomposition of the input ciphertext (a d-dimensional vector). Their formulation reduces key-switching to four steps: a gadget decomposition, its extension to the temporary modulus $Q \cdot P$, an inner product with the evaluation key, and a modulus lowering to Q.

Batching. In this work, we generalize the external-product formulation of Kim *et al.* [27] to an arbitrary number of input polynomials and secret keys. We present a *batched key-switching* (BKS) algorithm, which finds an affine ciphertext (with two components) under a target secret key that approximately opens to the linear combination of the n input polynomials with the n source secret keys. The term "batched" refers to the fact that key-switching for the n polynomials is performed at once via a single external product by an evaluation key of shape $2 \times nd$. The main advantage of batching the n key-switches is that the modulus reduction is performed only on the two components of the output ciphertext. BKS captures several key-switching algorithms presented in the literature; we prove its correctness, and derive explicit noise growth bounds.

Multiplication. We first use batched key-switching to relinearize the product of two or three ciphertexts. In the binary case, we recover the relinearization of Han and Ki [24] within our framework, and provide a corrected noise bound. In the ternary case, we first compute a ciphertext that decrypts under $(1, \mathsf{sk}, \mathsf{sk}^2, \mathsf{sk}^3)$, and relinearize the sk^2 and sk^3 components via a single batched key-switch. This *lazy* relinearization strategy is supported by libraries such as SEAL [36], OpenFHE [4], and Lattigo [17], and it has been exploited by FHE DNN compilers like EVA [15]

and Orion [16]. However, it is mostly treated as an implementation technique, and its noise behavior has rarely been analyzed; the works of Akherati *et al.* [2, 3] are notable exceptions, which we discuss in Section 4. Our ternary multiplication method places one of the two required rescalings before key-switching, so that the external product is evaluated over a smaller modulus. Compared with two consecutive binary multiplications, ternary multiplication consumes the same number of levels, introduces a single key-switching error instead of two, and does not amplify the error of the first multiplication by the third operand. A Lattigo implementation with 128-bit-secure parameters improves average precision by up to half a bit, and attains speedups between 1.07× and 1.32×.

The product of n ciphertexts, $n > 3$, can be evaluated via many different arrangements of binary and ternary multiplications, which differ in their depth consumption, precision, and latency. We present an algorithm that, given the levels of the input ciphertexts and a cost function $w(l, k)$ estimating the latency of a k-ary multiplication at level l, with $k \in \{2, 3\}$, returns a multiplication tree that delivers the product at the highest possible level and, among all such trees, minimizes the total cost. Its running time is $O(n^5)$ in general, and $O(n^4)$ when all inputs share the same level. In the latter case, if $n = 2^a 3^b$, the optimal tree consists of a binary tree of height a on top of ternary trees of height b; otherwise, the algorithm uses ternary multiplications whenever they reduce latency without consuming additional levels.

Products of three or more ciphertexts are commonplace in private inference. For instance, the SHIP bootstrapping algorithm [12], designed to minimize level consumption and exploit computational parallelism, requires the product of h ciphertexts, where h is the Hamming weight of the (usually sparse) secret key, which need not be a power of two. SHIP and its follow-ups [32] directly benefit from our optimal trees. Products of three ciphertexts also arise when nonlinear functions are approximated homomorphically. For example, the private transformer frameworks NEXUS [40] and THOR [31] approximate the reciprocal in the softmax function with Goldschmidt's algorithm [21], and NEXUS uses Newton's method [38] to approximate the inverse square root in layer normalization. When expanded, the iterations of both methods contain products of three ciphertexts.

Rotations. Batched key-switching can also be used to accumulate automorphisms. This kind of operations are a key part of the double-hoisted baby-step giant-step algorithm [9] for homomorphically evaluating a linear transform, which we recast in terms of our batched algorithms in order to study its noise behavior. The implication of our results extends to two important parts of a CKKS bootstrapping procedure: its coefficients-to-slots and slots-to-coefficients steps.

The main drawback of our approach is that it requires handling a larger number of polynomials simultaneously, which leads to increased memory usage. In the case of batched multiplication, we postulate that our proposal is practical when the number of ciphertext factors is ≤ 4, even in applications such as DNN inference where the ring dimension $N \geq 2^{15}$.

Our contributions. In summary, our work includes the following contributions:

– We present *batched key-switching* (Algorithm 2), which extends gadget-decomposition key-switching [27] to multiple polynomial inputs. This algorithm subsumes the most commonly used RNS-based key-switching methods in the literature. We prove its security and correctness, and bound the noise it introduces.
– We correct the noise bound for the relinearization method of Han–Ki [24], and propose a ternary multiplication method (Algorithm 4) that relinearizes the sk^2 and sk^3 components via a single batched key-switch and a two-phase rescaling procedure. We show that it introduces less noise than two consecutive binary multiplications, at the same level consumption.
– We describe a polynomial-time algorithm (Algorithm 5) that, for any number of ciphertexts at arbitrary levels, finds a tree of binary and ternary multiplications that minimizes level consumption and, subject to this, a user-specified cost function. We bound its running time and give a closed-form expression for its maximum attainable output level.
– We implement these algorithms on top of Lattigo [17], fit the latency cost function to measured running times, and compare against the standard binary-tree approach using 128-bit-secure parameters, observing gains of up to 0.5 bits of precision and speedups of up to 32%.
– We give a concise yet complete description of Bossuat *et al.*'s double-hoisted BSGS algorithm [9] in Algorithm 7, discover one optimization concerning the automorphism applications during its giant steps, and provide to the best of our knowledge its first explicit worst-case error analysis in the literature.

The rest of this paper is organized as follows: Section 2 reviews the relevant background concepts related to our work. Section 3 introduces the batched key-switching algorithm. Section 4 applies BKS to binary and ternary multiplication, discusses how to construct optimal multiplication trees, and evaluates an implementation on realistic multiplication problems. Section 5 shows the use of our key-switching techniques within the double-hoisted BSGS matrix-vector multiplication algorithm. Section 6 concludes by discussing potential future work.

2 Background

In this section, we review the CKKS encryption scheme [13] and its full-RNS variant [11]. We also study the gadget decomposition technique.

Notation. Bold lowercase and capital letters refer to vectors and matrices, respectively. If N is a power of two and $Q \geq 2$, let $\mathcal{R}$ and $\mathcal{R}_Q$ denote respectively the cyclotomic rings $\mathbb{Z}[X]/(X^N+1)$ and $\mathbb{Z}_Q[X]/(X^N+1)$. We identify an integer in $\mathbb{Z}_Q$ by its representative in $(-Q/2, Q/2]$. For $1 \leq p \leq \infty$ and a polynomial $a \in \mathcal{R}$, we use $\|a\|_p$ to denote the ℓ_p norm of the coefficient vector of a. The notation $v \leftarrow \chi$ means to sample a value v from the probability distribution χ. The operation $\lfloor \cdot \rceil$ rounds its input to the nearest integer; it applies coefficient-wise to a real polynomial, and in case of ties, it rounds up. Let PPT denote the class of decision problems solved by a probabilistic polynomial-time machine with error probability $< 1/2$, and let $\lambda \in \mathbb{N}$ denote the security parameter.

The RLWE Problem. We recall the decision variant of the ring learning with errors problem [35] over the ring $\mathcal{R}_Q$, and its hardness assumption with respect to the distributions $\chi_{\text{key}}, \chi_{\text{err}}$ over $\mathcal{R}$: Sample $\mathsf{sk} \leftarrow \chi_{\text{key}}$. It is computationally hard to distinguish between two sets of $\lambda^{O(1)}$ samples (a_i, b_i) and $(a_i, (-a_i \cdot \mathsf{sk} + e_i) \bmod Q)$, where a_i, b_i are uniformly random in $\mathcal{R}_Q$, and $e_i \leftarrow \chi_{\text{err}}$.

The CKKS Scheme. The Cheon-Kim-Kim-Song scheme [13] is an approximate FHE scheme built on the hardness of the RLWE problem. It operates on vectors of complex numbers by encoding these into a polynomial form suitable for encrypted computation. Let ω be a primitive $(2N)$th root of unity, and define $\tau(f) = (f(\omega), f(\omega^3), \ldots, f(\omega^{N-1}))$ to be the canonical embedding from $\mathbb{R}[X]/(X^N + 1)$ into $\mathbb{C}^{N/2}$. To obtain the plaintext representation of a message $\boldsymbol{m} \in \mathbb{C}^{N/2}$, apply the inverse of τ to $\boldsymbol{m}$, multiply the resulting polynomial by a given scaling factor Δ, and round its coefficients to the nearest integer. The result is a polynomial in $\mathcal{R}$ that—following a procedure that we shall explain shortly—may be encrypted into a ciphertext, namely a polynomial in the ring $\mathcal{R}_Q[S] = \mathbb{Z}_Q[X, S]/(X^N + 1)$.[1] The CKKS scheme provides the following elementary operations on keys and ciphertexts:

$\mathsf{Setup}(N, Q, P, \lambda, \eta)$. Take a power of two N, coprime integers Q and P, a security level λ, and a bit precision parameter η as input. Define $\tilde{Q} := QP$. Select a secret key distribution χ_{key}, an encryption key distribution χ_{enc}, and an error distribution χ_{err} over $\mathcal{R}$ that guarantee λ-bit security.

KeyGen. Sample the secret key $\mathsf{sk} \leftarrow \chi_{\text{key}}$, and construct the public key $\mathsf{pk} = a(S - \mathsf{sk}) + e \in \mathcal{R}_Q[S]$, where a is uniformly random in $\mathcal{R}_Q$ and $e \leftarrow \chi_{\text{err}}$.

$\mathsf{Encrypt}(\mathsf{pt}, \mathsf{pk})$. Given a plaintext $\mathsf{pt} \in \mathcal{R}$, sample $v \leftarrow \chi_{\text{enc}}$ and $e_0, e_1 \leftarrow \chi_{\text{err}}$; then output the ciphertext $\mathsf{ct}(S) = \mathsf{pk} + \mathsf{pt} + e_1 S + e_0 \in \mathcal{R}_Q[S]$.

$\mathsf{Decrypt}(\mathsf{ct}, \mathsf{sk})$. Evaluate a given ciphertext $\mathsf{ct}(S) \in \mathcal{R}_Q[S]$ at $S = \mathsf{sk}$, mod Q.

The full-RNS variant of the CKKS scheme [11] uses a residue number system (RNS) to represent an element of $\mathbb{Z}_Q$ as the list of its residues modulo some coprime integers $q_0, q_1, \ldots, q_L$ whose product is Q. The isomorphism $\mathcal{R}_Q \cong \mathcal{R}_{q_0} \times \cdots \times \mathcal{R}_{q_L}$ extends this idea in a coefficient-wise manner, allowing us to split a polynomial in $\mathcal{R}_Q$ into $L + 1$ polynomials that feature smaller coefficients (and thus are easier to compute with). For reasons that we discuss later, some CKKS operations work over the *extended* ring $\mathcal{R}_{\tilde{Q}}$ instead of over $\mathcal{R}_Q$, and so it becomes necessary to convert the RNS representation of an integer between two coprime bases. Let $(a_0, \ldots, a_{l-1})$ be the RNS representation of an integer $a \in \mathbb{Z}_Q$ with respect to the basis $(q_0, \ldots, q_{l-1})$, for some $0 \leq l \leq L + 1$. The following formula approximately transforms this RNS representation into one over another basis $(p_1, \ldots, p_k)$ of

[1] Han–Ki [24] and many other authors regard a ciphertext as a tuple of polynomials in $\mathcal{R}_Q$, but the multivariate polynomial representation is more convenient for our purposes.

coprime moduli:

$$(\tilde{a}_1, \ldots, \tilde{a}_k) = \left(\sum_{j=0}^{l-1} (a_j \cdot \hat{q}_j^{-1} \bmod q_j) \cdot \hat{q}_j \pmod{p_i} \right)_{i=1}^{k}, \quad \text{where } \hat{q}_j = \frac{Q}{q_j}. \quad (1)$$

More precisely, $(\tilde{a}_1, \ldots, \tilde{a}_k)$ is the RNS representation of an integer $\tilde{a} = a + Q \cdot e$ such that $\tilde{a} \equiv a \pmod{Q}$ and $|e| \leq l/2$. In this way, the RNS extension of a from $\mathbb{Z}_Q$ to $\mathbb{Z}_{\tilde{Q}}$ is the concatenation of the RNS representations $(a_0, \ldots, a_{l-1})$ and $(\tilde{a}_1, \ldots, \tilde{a}_k)$. Proceeding in the reverse direction, to approximately lower the RNS representation of $\tilde{a} \in \mathbb{Z}_{\tilde{Q}}$ back to the basis $(q_0, \ldots, q_{l-1})$, use the basis conversion formula (1) to convert $(\tilde{a}_1, \ldots, \tilde{a}_k)$ from the basis $(p_1, \ldots, p_k)$ into $(c_0, \ldots, c_{l-1})$ in the basis $(q_0, \ldots, q_{l-1})$; then compute $(b_0, \ldots, b_{l-1}) = (\lfloor P^{-1} \cdot (a_j - c_j) \rceil)_{0 \leq j < l}$, which represents an integer $b \in \mathbb{Z}_Q$ such that $|b - P^{-1} \cdot \tilde{a}| \leq k/2$.

The full-RNS variant of CKKS manages the magnitude of ciphertexts through a *rescaling* procedure. A ciphertext $\mathsf{ct}(S)$ starts as a polynomial in the ring $\mathcal{R}_Q[S]$ and has an associated scaling factor Δ. Homomorphic operations progressively grow the effective scale of $\mathsf{ct}(S)$ and consume the coefficient modulus $Q = q_0 \cdots q_L$. Rescaling reverts these effects by lowering the RNS representation of $\mathsf{ct}(S)$ from the basis $(q_0, \ldots, q_l)$ down to $(q_0, \ldots, q_{l-1})$ for $0 < \ell < L$, i.e., by dropping the last modulus q_l. Defining $Q_l = \prod_{k=0}^{l} q_k$, this operation is formalized as follows:

$\mathsf{Rescale}(\mathsf{ct}, \Delta)$. Given a ciphertext $\mathsf{ct}(S) \in \mathcal{R}_{Q_l}$ at level $0 < l \leq L$ and scale Δ, output the ciphertext $\mathsf{ct}'(S) = \lfloor q_l^{-1} \cdot \mathsf{ct}(S) \rceil \in \mathcal{R}_{Q_{l-1}}$ at level $l-1$ and scale Δ/q_l.

The discussion above shows that $\mathsf{ct}'(\mathsf{sk}) = \lfloor q_l^{-1} \cdot \mathsf{ct}(\mathsf{sk}) \rceil + e(\mathsf{sk})$ for some $e(S) \in \mathbb{Q}[X, S]/(X^N + 1)$ satisfying $\|e(\mathsf{sk})\|_\infty \leq B_{\mathrm{rs}} := (1 + \|\mathsf{sk}\|_1)/2$.

Gadget Decompositions. The gadget decomposition technique is a useful tool for cutting down noise growth during homomorphic computations. Genise *et al.* [18] define a $\mathbb{Z}_Q$*-gadget of quality* β as a vector $\boldsymbol{g} \in \mathbb{Z}_Q^d$ for which there exists a function $h \colon \mathbb{Z}_Q \to \mathbb{Z}_Q^d$ such that $\boldsymbol{g}^T h(a) = a$ and $\|h(a)\|_\infty \leq \beta$ for every integer $a \in \mathbb{Z}_Q$. Kim *et al.* [27] described a practical method for constructing gadget vectors from an RNS basis $\mathcal{B} = \{q_0, \ldots, q_L\}$: Partition $\mathcal{B}$ into nonempty subsets $\mathcal{B}_1, \ldots, \mathcal{B}_d$, and define the *digits* $D_j = \prod_{q \in \mathcal{B}_j} q$ for $1 \leq j \leq d$. By Equation (1), the vector $\boldsymbol{g} \in \mathbb{Z}_Q^d$ with entries $g_j = ((Q/D_j)^{-1} \bmod D_j) \cdot (Q/D_j)$ is a $\mathbb{Z}_Q$-gadget of quality $\beta = \frac{1}{2} \max_{1 \leq j \leq d} D_j$. The relevant decomposition function h maps each element of $\mathbb{Z}_Q$ into its RNS representation in the basis $(D_1, \ldots, D_d)$.

In this setting, the *external product* $\boldsymbol{u} \boxdot v$ of a vector $\boldsymbol{u} \in \mathbb{Z}_{\tilde{Q}}$ by an integer $v \in \mathbb{Z}_Q$ is defined as $\lfloor P^{-1} \cdot (\boldsymbol{u}^T h(v) \bmod \tilde{Q}) \rceil \pmod{Q}$. This operation arises naturally in the key-switching procedure of CKKS. It generalizes to matrices $\boldsymbol{U} \in \mathbb{Z}_{\tilde{Q}}^{k \times d}$ as $\boldsymbol{U} \boxdot v = \left[\boldsymbol{u}_1 \boxdot v, \ldots, \boldsymbol{u}_k \boxdot v \right]^T$, and to polynomials $v \in \mathcal{R}_Q$ by decomposing each coefficient independently.

The NTT. Suppose that $q_i \equiv 1 \pmod{2N}$ for every $0 \leq i \leq L$, and let ω_i be a primitive $(2N)$th root of unity modulo q_i. A Number Theoretic Transform maps a polynomial $a \in \mathcal{R}_{q_i}$ into the evaluation vector $\hat{\boldsymbol{a}} = (a(\omega_i), a(\omega_i^3), \ldots, a(\omega_i^{2N-1})) \in \mathbb{Z}_{q_i}^N$. Since the NTT is a Fourier transform, it reduces polynomial multiplication

to entry-wise multiplication of evaluation vectors. A polynomial is said to be in *coefficient form* when represented by its coefficients, and in *evaluation form* when represented by its NTT vector. Applying an NTT to a polynomial in $\mathcal{R}_{Q_l}$ amounts to applying the NTT to each component of its RNS representation in the basis $(q_0, \ldots, q_l)$. It will be convenient to define the norm $|||b|||_\infty = \|\hat{\boldsymbol{b}}\|_\infty$ for $b \in \mathcal{R}_Q$, and to remember the inequalities

$$|||a \cdot b|||_\infty \le |||a|||_\infty |||b|||_\infty, \qquad |||a|||_\infty \le \|a\|_1, \qquad \|a\|_\infty \le |||a|||_\infty,$$

for $a, b \in \mathcal{R}_Q$; the last one holds true only because N is a power of 2 [14].

3 Batched Key-Switching

In this section, we extend the standard decomposition key-switching technique to handle an arbitrary number of input polynomials at once.

3.1 Hoisting

Many CKKS circuits proceed in two steps: First, they transform one or more ciphertexts into an affine-linear form $f(S, S'_1, \ldots, S'_n) = b + aS + \sum_{i=1}^n r_i S'_i$ whose coefficients are all in $\mathcal{R}_Q$. Second, they apply n key-switching operations to the r_i's in order to obtain a ciphertext $b' + a'S$ over the secret-key indeterminate S alone. For example, to multiply two affine ciphertexts $\mathsf{ct}(S)$ and $\mathsf{ct}'(S)$ that encrypt messages under a common secret key sk, we compute the form $f(S, S^2) = \mathsf{ct}(S) \cdot \mathsf{ct}'(S)$ and subsequently replace its term rS^2 by $\boldsymbol{U} \boxdot r$, where $\boldsymbol{U}$ is an evaluation key from sk^2 to sk. The key-switching step has the general form

$$\begin{array}{l} \text{Initialize } \mathsf{ct}(S) \text{ to some affine ciphertext in } \mathcal{R}_Q[S]. \\ \text{For } 1 \le i \le n, \text{ add } \boldsymbol{U}_i \boxdot r_i \text{ to } \mathsf{ct}(S). \end{array} \tag{2}$$

Here $\boldsymbol{U}_1, \ldots, \boldsymbol{U}_n$ are evaluation keys constructed from the same gadget vector $\boldsymbol{g} \in \mathbb{Z}_Q^d$. The loop in Program (2) requires a significant amount of computation, because each external product evaluation involves a gadget decomposition, several NTT operations, and $d + 2$ RNS reconstructions. Furthermore, since the last two of these reconstructions need an integer rounding operation to reduce the modulus space, they introduce a nonnegligible amount of noise into $\mathsf{ct}(S)$. These deficiencies have led many authors dealing with particular instances of Program (2) to suggest taking the modulus lowering step out of the loop. In this way, the circuit becomes

$$\begin{array}{l} \text{Initialize } \tilde{\mathsf{ct}}(S) \text{ to some affine ciphertext in } \mathcal{R}_{\tilde{Q}}[S]. \\ \text{For } 1 \le i \le n, \\ \quad \text{Construct the gadget decomposition } \boldsymbol{d}_i \text{ of } r_i \text{ with respect to } \boldsymbol{g}. \\ \quad \text{Lift } \boldsymbol{d}_i \text{ to its RNS representation } \tilde{\boldsymbol{d}}_i \text{ over } \mathcal{R}_{\tilde{Q}}^d. \\ \quad \text{Add the affine polynomial } \begin{bmatrix} 1 & S \end{bmatrix} \boldsymbol{U}_i \tilde{\boldsymbol{d}}_i \text{ to } \tilde{\mathsf{ct}}(S). \\ \text{Reduce } \tilde{\mathsf{ct}}(S) \text{ back to } \mathcal{R}_Q[S] \text{ to obtain } \mathsf{ct}(S). \end{array} \tag{3}$$

Notice that the addition of the key-switched r_i's now proceeds over the extended ring $\mathcal{R}_{\tilde{Q}}[S]$, but this is more than ample compensation for the savings of $2n - 2$

RNS reconstructions and a reduction of noise due to rounding from $O(n)$ to $O(1)$ in the ℓ_∞-norm sense.

There are circumstances where we face a slightly different problem: given a ciphertext $\mathsf{ct}(S) = b + aS$ in $\mathcal{R}_Q[S]$, the objective now is to obtain multiple affine ciphertexts $\mathsf{ct}_i(S)$ that decrypt under different secret keys:

$$\begin{array}{l}\text{Initialize } \mathsf{ct}'(S) \text{ to some affine ciphertext} \\ \quad \text{in } \mathcal{R}_Q[S], \text{ possibly depending on } \mathsf{ct}(S). \\ \text{For } 1 \le i \le n, \text{ set } \mathsf{ct}_i(S) \text{ to } \mathsf{ct}'(S) + \boldsymbol{U}_i \boxdot a.\end{array} \tag{4}$$

Notice that all external products take the polynomial a as input, and thus perform the same d RNS reconstructions in order to construct its gadget decomposition. We may avoid the redundant computation by extracting this step out of the loop:

$$\begin{array}{l}\text{Initialize } \mathsf{ct}'(S) \text{ to some affine ciphertext in } \mathcal{R}_Q[S]. \\ \text{Construct the gadget decomposition } \boldsymbol{d} \text{ of } a \text{ in the basis } \boldsymbol{g}. \\ \text{Lift } \boldsymbol{d} \text{ to its RNS representation } \tilde{\boldsymbol{d}} \text{ over } \mathcal{R}_{\tilde{Q}}. \\ \text{For } 1 \le i \le n, \\ \quad \text{Set } \tilde{\mathsf{ct}}_i(S) \text{ to the affine polynomial } \begin{bmatrix}1 & S\end{bmatrix} \boldsymbol{U}_i \tilde{\boldsymbol{d}}. \\ \quad \text{Reduce } \tilde{\mathsf{ct}}_i(S) \text{ back to } \mathcal{R}_Q[S] \text{ to obtain } \mathsf{ct}_i(S). \\ \quad \text{Add } \mathsf{ct}'(S) \text{ to } \mathsf{ct}_i(S).\end{array} \tag{5}$$

The change from Program (4) to (5) poses no effect on noise growth, but it sometimes gives us the opportunity to reduce the amount of noise introduced by subsequent operations, namely by applying these to the intermediate $\tilde{\mathsf{ct}}_i(S)$ over the extended ring $\mathcal{R}_{\tilde{Q}}[S]$, instead of to $\mathsf{ct}_i(S)$. (Of course, this usually leads to a slight increase in execution time and/or space.)

The two *hoisting* transformations[2] we have just seen turn out to apply in a wide variety of situations, which makes them good candidates for integration as optimization passes into homomorphic encryption compilers like HEaaN.MLIR [33] and Google HEIR [5]. To illustrate this point, Sections 4 and 5 discuss two particular, yet popular, applications from the literature. Their running times have already been analyzed in detail (and the complexity analyses extend readily to the general case) but, to the best of our knowledge, other important aspects such as their noise growth behavior remain unknown. Our present goal is to settle some of these matters by studying Programs (3) and (5) in full generality; later, we will discuss their implications on the applications.

3.2 The Algorithm

We begin by describing a procedure that generates the necessary evaluation keys in batch, Algorithm 1. Bundling these keys into a single matrix does not compromise security, as Theorem 1 shows.

Theorem 1. *Under the RLWE hardness assumption, an evaluation key $\boldsymbol{U}$ generated by an honest party via Algorithm 1 is computationally indistinguishable*

[2] The term "hoisting" is borrowed from the field of compiler optimization, where it refers to moving a loop-invariant computation out of a loop.

Algorithm 1: Evaluation Key Generation

Given secret keys $\mathsf{sk}'_1, \mathsf{sk}'_2, \ldots, \mathsf{sk}'_n, \mathsf{sk} \in \mathcal{R}$, and a gadget vector $\boldsymbol{g} \in \mathbb{Z}_Q^d$, this algorithm generates an evaluation key $\boldsymbol{U} \in \mathcal{R}_{\tilde{Q}}^{2\times nd}$ necessary for the key-switching procedure in Algorithm 2.

Express the matrix $\boldsymbol{U}$ in the block form $\left[\begin{smallmatrix} \boldsymbol{u}_{0,1} & \cdots & \boldsymbol{u}_{0,n} \\ \boldsymbol{u}_{1,1} & \cdots & \boldsymbol{u}_{1,n} \end{smallmatrix}\right]$, where each $\boldsymbol{u}_{i,j}$ is a row vector in $\mathcal{R}_{\tilde{Q}}^d$. Select the n coefficients of each entry of $\boldsymbol{u}_{1,i}$ uniformly at random, for $1 \le i \le n$. Then generate a row vector $\boldsymbol{e}_i \in \mathcal{R}^d$ whose entries have coefficients sampled from χ_{err}, and set $\boldsymbol{u}_{0,i} := -\mathsf{sk} \cdot \boldsymbol{u}_{1,i} + P \cdot \mathsf{sk}'_i \cdot \boldsymbol{g}^T + \boldsymbol{e}_i \bmod \tilde{Q}$.

from a $2 \times nd$ matrix of independently and uniformly distributed polynomials in $\mathcal{R}_{\tilde{Q}}$, provided that n and d are polynomial in λ.

Proof. The goal is to show that the nd columns of $\boldsymbol{U}$ have a joint distribution $\mathcal{D}_0$ that is computationally indistinguishable from that of nd independent and uniformly random pairs of polynomials in $\mathcal{R}_{\tilde{Q}}$. For $1 \le k \le nd$, define $\mathcal{D}_k$ to be the same distribution as $\mathcal{D}_{k-1}$, but with the kth pair being uniformly random and independent of the others; in terms of this notation, we need to prove that $\mathcal{D}_0$ and $\mathcal{D}_{nd}$ are indistinguishable.

Let $\mathcal{A}$ be a PPT adversary that can distinguish between two consecutive distributions $\mathcal{D}_k$ and $\mathcal{D}_{k-1}$ with advantage ε, for any $1 \le k \le nd$. We shall now construct a PPT reduction $\mathcal{B}$ that uses $\mathcal{A}$ to break the RLWE assumption over $\mathcal{R}_{\tilde{Q}}$ with the same advantage ε. The reduction $\mathcal{B}$ receives a challenge pair (a, b) that is either a RLWE sample (which satisfies $b = -a \cdot \mathsf{sk} + e$ for some $e \leftarrow \chi_{\mathrm{err}}$), or a pair of polynomials sampled uniformly from $\mathcal{R}_{\tilde{Q}}$. Next, it samples secret keys $\mathsf{sk}'_{\lfloor (k-1)/d \rfloor + 1}, \ldots, \mathsf{sk}'_n, \mathsf{sk} \leftarrow \chi_{\mathrm{key}}$, and partly simulates the evaluation key generation process in Algorithm 1. More precisely, it constructs a matrix $\boldsymbol{V} \in \mathcal{R}_{\tilde{Q}}^{2\times nd}$ such that

- The first $k-1$ columns consist of uniformly random polynomials in $\mathcal{R}_{\tilde{Q}}$.
- Columns $j = k+1, \ldots, nd$ have the form $\begin{bmatrix} v_{0,j} & v_{1,j} \end{bmatrix}^T$, where $v_{1,j}$ is uniformly random in $\mathcal{R}_{\tilde{Q}}$, and $v_{0,j} := -\mathsf{sk} \cdot v_{1,j} + P \cdot \mathsf{sk}'_{\lfloor (j-1)/d \rfloor + 1} \cdot \boldsymbol{g}_{(j-1) \bmod d+1} + e_j$ with $e_j \leftarrow \chi_{\mathrm{err}}$. These are valid evaluation keys.
- The kth column $\begin{bmatrix} v_{0,k} & v_{1,k} \end{bmatrix}^T$ satisfies $v_{1,k} := a$ and $v_{0,k} := b + P \cdot \mathsf{sk}'_{\lfloor (k-1)/d \rfloor + 1} \cdot \boldsymbol{g}_{(k-1) \bmod d+1}$.

Note that $v_{0,k} + \mathsf{sk} \cdot v_{1,k} = b + \mathsf{sk} \cdot a + P \cdot \mathsf{sk}'_{\lfloor (k-1)/d \rfloor + 1} \cdot \boldsymbol{g}_{(k-1) \bmod d+1}$ is a valid evaluation key if and only if (a, b) is an RLWE sample. (When b is uniformly random, it acts as a blinding mask for the other term in $v_{0,k}$, which makes the kth column of $\boldsymbol{V}$ uniformly random.) The reduction $\mathcal{B}$ finishes by outputting the same bit as $\mathcal{A}$ on input $\boldsymbol{V}$. The previous construction guarantees that $\boldsymbol{V}$ has the distribution $\mathcal{D}_k$ if (a, b) is a RLWE sample, and $\mathcal{D}_{k-1}$ otherwise. Consequently, $\mathcal{B}$ breaks the RLWE problem with the same advantage that $\mathcal{A}$ has for distinguishing

between two consecutive distributions. So ε is negligible by the RLWE hardness assumption.

Finally, an adversary that can distinguish between $\mathcal{D}_0$ and $\mathcal{D}_{nd}$ has advantage

$$\left| \Pr_{h \leftarrow \mathcal{D}_{nd}} (\mathcal{A}(h) = 1) - \Pr_{h \leftarrow \mathcal{D}_0} (\mathcal{A}(h) = 1) \right| \\ \leq \sum_{k=1}^{nd} \left| \Pr_{h \leftarrow \mathcal{D}_k} (\mathcal{A}(h) = 1) - \Pr_{h \leftarrow \mathcal{D}_{k-1}} (\mathcal{A}(h) = 1) \right| \leq nd\varepsilon,$$

which is also negligible, since nd is polynomial in λ. □

Using the evaluation key generated by Algorithm 1, the method described in Algorithm 2 applies the key-switch operation to a batch of polynomials (mimicking Program (3)). Its full complexity analysis can be carried out in an analogous way as that of Kim *et al.* [27]; we only mention that in some cases $\tilde{b}$ is given in evaluation form, and then it is advantageous to swap the order of steps 5 and 6 to save on one INTT operation.

Algorithm 2: Batched Key-Switching

Let $\mathsf{sk}'_1, \mathsf{sk}'_2 \ldots, \mathsf{sk}'_n, \mathsf{sk}$ be secret keys in $\mathcal{R}$. Given polynomials $r_1, \ldots, r_n \in \mathcal{R}_Q$ and $\tilde{b} \in \mathcal{R}_{\tilde{Q}}$ in coefficient form, and an evaluation key $\boldsymbol{U} \in \mathcal{R}_{\tilde{Q}}^{2 \times nd}$ from $(\mathsf{sk}'_1, \ldots, \mathsf{sk}'_n)$ to sk produced by Algorithm 1 for a gadget vector $\boldsymbol{g} \in \mathbb{Z}_Q^d$, this algorithm finds an affine ciphertext $\mathsf{ct}(S) \in \mathcal{R}_Q[S]$ in coefficient form such that $\mathsf{ct}(\mathsf{sk}) \approx r_1 \mathsf{sk}'_1 + \cdots + r_n \mathsf{sk}'_n + P^{-1} \cdot \tilde{b} \pmod{Q}$.

1. [Decompose.] For $1 \leq i \leq n$, construct the gadget decomposition $\boldsymbol{d}_i \in \mathcal{R}_Q^d$ of r_i with respect to the basis vector $\boldsymbol{g}$.
2. [Extend.] For $1 \leq i \leq n$, find the RNS representation $\tilde{\boldsymbol{d}}_i$ of $\boldsymbol{d}_i$ over $\mathcal{R}_{\tilde{Q}}^d$.
3. [NTT.] Convert each $\tilde{\boldsymbol{d}}_i$ into its evaluation form, for $1 \leq i \leq n$.
4. [Batch.] Compute the bilinear form

$$\tilde{\mathsf{ct}}(S) := \begin{bmatrix} 1 & S \end{bmatrix} \boldsymbol{U} \begin{bmatrix} \tilde{\boldsymbol{d}}_1^T & \cdots & \tilde{\boldsymbol{d}}_n^T \end{bmatrix}^T = \sum_{i=1}^{n} \boldsymbol{u}_{0,i} \tilde{\boldsymbol{d}}_i + S \sum_{i=1}^{n} \boldsymbol{u}_{1,i} \tilde{\boldsymbol{d}}_i \mod \tilde{Q}.$$

5. [INTT.] Convert $\tilde{\mathsf{ct}}(S)$ into its coefficient form.
6. [Add bias.] Add the polynomial $\tilde{b}$ to $\tilde{\mathsf{ct}}(S)$.
7. [Lower.] Output the ciphertext $\mathsf{ct}(S) = \lfloor P^{-1} \cdot \tilde{\mathsf{ct}}(S) \rceil$.

Theorem 2 establishes the correctness of Algorithm 2, and provides a bound on the amount of noise that it introduces.

Theorem 2. *Let $Q = q_0 \cdots q_l$ and $P = p_1 \cdots p_k$ be coprime products of coprime integers, let $\boldsymbol{g} \in \mathbb{Z}_Q^d$ be a gadget vector, and let $Q = D_1 \cdots D_d$ be the digit decomposition used to construct $\boldsymbol{g}$. Define $C := Nn\varepsilon_{\mathrm{f}}(l+1) \max_{1 \leq j \leq d} D_j$, where*

ε_{f} is an upper bound on the ℓ_∞ norm of the error term in a fresh encryption. If $P > C$, then Algorithm 2 produces a ciphertext $\mathsf{ct}(S) \in \mathcal{R}_Q[S]$ that opens to $\mathsf{ct}(\mathsf{sk}) \equiv \sum_{i=1}^{n} r_i \cdot \mathsf{sk}'_i + P^{-1} \cdot \tilde{b} + \varepsilon(\mathsf{sk}) \bmod Q$, where $\|\varepsilon(\mathsf{sk})\|_\infty \leq \frac{1}{2}(1 + k + \|\mathsf{sk}\|_1 k)$.

Proof. After step 4, the ciphertext $\tilde{\mathsf{ct}}$ opens to

$$\tilde{\mathsf{ct}}(\mathsf{sk}) \equiv \sum_{i=1}^{n} (\boldsymbol{u}_{0,i} + \mathsf{sk} \cdot \boldsymbol{u}_{1,i})\, \tilde{\boldsymbol{d}}_i \pmod{Q},$$

where $\boldsymbol{u}_{0,i} + \boldsymbol{u}_{1,i} S$ is an encryption of sk'_i under sk with respect to the gadget basis $\boldsymbol{g}$. Hence

$$\begin{aligned}\tilde{\mathsf{ct}}(\mathsf{sk}) &= \sum_{i=1}^{n} \left(-\mathsf{sk} \cdot \boldsymbol{u}_{1,i} + P \cdot \mathsf{sk}'_i \cdot \boldsymbol{g}^T + \boldsymbol{e}_i + \mathsf{sk} \cdot \boldsymbol{u}_{1,i}\right) \tilde{\boldsymbol{d}}_i \\ &= \sum_{i=1}^{n} \left(P \cdot \mathsf{sk}'_i \cdot \boldsymbol{g}^T + \boldsymbol{e}_i\right) \tilde{\boldsymbol{d}}_i \mod \tilde{Q}.\end{aligned}$$

All the polynomial products in step 4 are performed in the evaluation domain; by Minkowski's inequality and the properties of the evaluation norm $|||\cdot|||$, the error term $\tilde{e} := \sum_{i=1}^{n} \boldsymbol{e}_i \tilde{\boldsymbol{d}}_i$ of $\tilde{\mathsf{ct}}(S)$ satisfies the bound

$$|||\tilde{e}|||_\infty \leq \sum_{i=1}^{n} |||\boldsymbol{e}_i \tilde{\boldsymbol{d}}_i|||_\infty \leq \sum_{i=1}^{n} \sum_{j=1}^{d} |||(\boldsymbol{e}_i)_j|||_\infty |||(\tilde{\boldsymbol{d}}_i)_j|||_\infty. \tag{6}$$

Recall that vector $\tilde{\boldsymbol{d}}_i$ is the extension to $\mathcal{R}_{\tilde{Q}}^d$ of the gadget decomposition $\boldsymbol{d}_i$ in the basis $\boldsymbol{g} \in \mathbb{Z}_Q^d$. Denote by L_j the length of digit D_j, namely the number of moduli $q_0, \ldots, q_l$ that comprise it. We know in particular that $L_1 + \cdots + L_d = l + 1$, because the digits $D_1, \ldots, D_d$ correspond to a partition of the q_i. After step 2—which uses the fast basis conversion technique [7, 11] to extend the RNS representation of each polynomial $(\boldsymbol{d}_i)_j \in \mathcal{R}_{D_j}$ to $(\tilde{\boldsymbol{d}}_i)_j \in \mathcal{R}_{\tilde{Q}}$—the inequality $\|(\tilde{\boldsymbol{d}}_i)_j\|_1 \leq N L_j D_j / 2$ holds. Applying this bound together with $\|\boldsymbol{e}_i\|_\infty \leq \varepsilon_{\mathrm{f}}$ and $|||(\tilde{\boldsymbol{d}}_i)_j|||_\infty \leq \|(\tilde{\boldsymbol{d}}_i)_j\|_1$ to Eq. (6) yields

$$|||\tilde{e}|||_\infty \leq \frac{1}{2} N n \varepsilon_{\mathrm{f}} \sum_{j=1}^{d} L_j D_j \leq \frac{1}{2} N n \varepsilon_{\mathrm{f}} (l+1) \max_{1 \leq j \leq d} D_j = \frac{C}{2}.$$

Step 5 brings $\tilde{\mathsf{ct}}(S)$ into coefficient form. The previous bound still holds for $\|\tilde{e}\|_\infty$, because N is a power of two. Finally, step 7 outputs a ciphertext $\mathsf{ct}(S) \in \mathcal{R}_Q[S]$ such that

$$\mathsf{ct}(\mathsf{sk}) \equiv \sum_{i=1}^{n} \mathsf{sk}'_i \cdot \boldsymbol{g}^T \tilde{\boldsymbol{d}}_i + P^{-1} \cdot \tilde{b} + P^{-1} \cdot \tilde{e} + \varepsilon_{\mathrm{r}}(\mathsf{sk}) \pmod{Q}, \tag{7}$$

where the polynomial $\varepsilon_{\mathrm{r}}(S) := \varepsilon_0 + \varepsilon_1 S \in \mathbb{Q}[S, X]/(X^N + 1)$ contains the rounding residuals. The first two terms on the right-hand side of Eq. (7) are precisely the

result of key-switching without error, because $\boldsymbol{d}_i$ is the gadget decomposition of r_i in the basis $\boldsymbol{g}$. It follows from $\|\tilde{e}\|_\infty \leq C/2$ and the assumption $P > C$ that $\|P^{-1} \cdot \tilde{e}\|_\infty < 1/2$. And since $\|\varepsilon_0\|_\infty$ and $\|\varepsilon_1\|_\infty$ are both at most $k/2$,[3] Minkowski's and Young's convolution inequalities yield

$$\|\varepsilon_{\mathrm{r}}(\mathsf{sk})\|_\infty \leq \|\varepsilon_0\|_\infty + \|\mathsf{sk}\|_1 \|\varepsilon_1\|_\infty \leq \tfrac{1}{2}(1 + \|\mathsf{sk}\|_1)k.$$

The triangle inequality then implies that the total approximation error is

$$\|\varepsilon(\mathsf{sk})\|_\infty \leq \|P^{-1} \cdot \tilde{e}\|_\infty + \|\varepsilon_{\mathrm{r}}(\mathsf{sk})\|_\infty \leq \tfrac{1}{2}(1 + k + \|\mathsf{sk}\|_1 k). \qquad \square$$

Exchanging sk with sk'_i in the description of Algorithm 1 leads to a procedure that generates an evaluation key with which Program (5) may key-switch a single polynomial from one secret key into several distinct keys. It is easy to see that Theorem 1 still holds after this modification, i.e., that the evaluation key remains computationally indistinguishable from random. A noise analysis of this batched $\mathsf{sk} \to (\mathsf{sk}'_1, \ldots, \mathsf{sk}'_n)$ key-switching operation may be performed in exactly the same manner as in the proof of Theorem 2, and, in fact, that result applies to each output ciphertext $\mathsf{ct}_i(S)$ when $n = 1$.

4 General Ciphertext Multiplication

4.1 Related Work

Akherati *et al.* [2] proposed a three-input ciphertext multiplication that first multiplies the three ciphertexts, and then relinearizes the sk^2 and sk^3 components, using two separate key-switches and performing both rescalings at the end. In follow-up work, Akherati–Zhang [3] observed that both relinearizations may be merged into a single equation, so that a modulus lowering is applied only once per output component, instead of twice. Since the output must be a two-component ciphertext, this is the minimum possible, and it corresponds to what we call *batched relinearization*. They also generalize lazy relinearization to an arbitrary number of inputs, with intermediate rescalings placed so as not to increase the multiplicative depth, and propose several partitions of the inputs for different numbers of ciphertexts.

Our work differs from theirs in three respects. First, we formalize batched relinearization through BKS for any number of inputs and source keys, providing correctness and noise growth proofs. Second, our ternary multiplication performs one of the two rescalings before KS, and hence the most expensive part of KS operates over a smaller modulus. This introduces an additional rounding error before KS compared with [2], but the resulting noise remains below that of two consecutive binary multiplications, because only one KS error is introduced. Third, the solutions in [3] aim to minimize the number of rescalings, are given only for $n \leq 12$, and come without a criterion for choosing among the proposed alternatives. We obtain multiplication trees by minimizing an arbitrary cost

[3] In the original proof of Han–Ki [24, Theorem 2], this bound is missing the factor k.

function that accounts for the level at which each KS is performed. Our approach is also more general: the input ciphertexts may be at different levels, and the target level can be chosen freely; we evaluate it for up to $n = 192$ inputs.

We restrict our multiplication trees to binary and ternary nodes. Multiplying four or more ciphertexts at once increases the number of components to relinearize, the memory footprint, and the scale growth before rescaling, which we show to be impractical.

4.2 Low-Order Multiplication

In this section, we present the two building blocks of our multiplication trees in Section 4.3: the binary and the ternary multiplication algorithms. We prove their correctness, and analyze their noise growth behavior. Both algorithms assume that their inputs arrive at the same level; otherwise rescalings should be performed to equalize their level beforehand.

Binary Multiplication. Algorithm 3 adapts Han–Ki's [24] relinearization process to the case $n = 2$ of our BKS technique (Algorithm 2). Theorem 3 establishes its correctness and bounds its error. We omit the proof since it closely follows that of Han–Ki, except that we account for the factor k arising from the KS error analyzed in Theorem 2, which is missing from their bound.

Algorithm 3: Binary Multiplication

Given an evaluation key $\boldsymbol{U} \in \mathcal{R}_{\tilde{Q}}^{2\times 2d}$ from sk^2 to sk, generated by Algorithm 1, and two affine ciphertexts $\mathsf{ct}_1(S), \mathsf{ct}_2(S)$ in the secret-key variable S encrypting messages $\mathsf{m}_1, \mathsf{m}_2$, all in evaluation form, at level l and scale Δ, this algorithm outputs an affine ciphertext at level $l' := l - 1$ in coefficient form that approximately decrypts to $\mathsf{m}_1 \cdot \mathsf{m}_2$ under sk.

1. [Multiply.] Compute the product $a(S) := \mathsf{ct}_1(S) \cdot \mathsf{ct}_2(S) = \sum_{s=0}^{2} a_s S^s$.
2. [INTT.] Convert $a(S)$ to coefficient form.
3. [Key-switch.] Apply Algorithm 2 with $r_1 = a_2$, $\tilde{b} = 0$; the result is an affine ciphertext $\mathsf{ct}(S) \in \mathcal{R}_Q[S]$.
4. [Add.] Compute $\mathsf{ct}'(S) := a_0 + a_1 S + \mathsf{ct}(S)$.
5. [Rescale.] Output the ciphertext $\mathsf{ct}_{\texttt{Mult}}(S) = \lfloor q_\ell^{-1} \cdot \mathsf{ct}'(S) \rceil$.

Theorem 3. *Let $\mathsf{ct}_1(S), \mathsf{ct}_2(S)$ be encryptions of $\mathsf{m}_1 + e_1$, $\mathsf{m}_2 + e_2$ under sk. If the conditions in the statement of Theorem 2 hold, then Algorithm 3 returns a ciphertext $\mathsf{ct}_{\texttt{Mult}}(S)$ such that*

$$\mathsf{ct}_{\texttt{Mult}}(\mathsf{sk}) \equiv q_\ell^{-1} \mathsf{m}_1 \cdot \mathsf{m}_2 + \varepsilon_{\mathrm{bm}} \pmod{Q},$$

where $\varepsilon_{\text{bm}} = q_\ell^{-1}(\mathsf{m}_1 e_2 + \mathsf{m}_2 e_1 + e_1 e_2 + \varepsilon) + e_R$, ε *is defined in Theorem 2, and* e_R *is a rounding error. We also have* $\|\varepsilon_{\text{bm}}\|_\infty < q_\ell^{-1}(2\nu B + B^2 + (1 + k + \|\mathsf{sk}\|_1 k)/1) + 1/2$ *if* $\|\mathsf{m}_i\|_\infty < \nu$ *and* $\|e_i\|_\infty < B$.

Ternary Multiplication. We now present an algorithm to multiply three ciphertexts at the same level and scale, with only a single invocation of the BKS algorithm. This has the benefit that the error of the KS process is generated only once, and hence we exploit the batching inside it to save computations in the modulus lowering step. Since two multiplications take place, we have to apply rescaling twice in order to restore the original scale of the inputs. Instead of rescaling twice at the end of the process, we place one of the rescalings in an intermediate stage, concretely just before the call to the BKS algorithm. This introduces slightly more rounding noise than if the two rescales were done after the lowering phase, but it speeds up the computations significantly in the KS, because it operates in a smaller modulus.

Algorithm 4 describes the steps of the ternary multiplication with 2-phase rescaling and BKS. Theorem 4 proves its correctness, and analyzes the noise growth.

Algorithm 4: Ternary Multiplication with 2-Phase Rescaling

Given an evaluation key $\boldsymbol{U} \in \mathcal{R}_{\tilde{Q}}^{2\times 3d}$ from sk^3 to sk, generated by Algorithm 1, and three affine ciphertexts $\mathsf{ct}_1(S), \mathsf{ct}_2(S), \mathsf{ct}_3(S)$ in the secret-key variable S, encrypting messages $\mathsf{m}_1, \mathsf{m}_2, \mathsf{m}_3$, all in evaluation form, at level ℓ and scale Δ, this algorithm outputs an affine ciphertext at level $l' := \ell - 2$ in coefficient form that approximately decrypts to $\mathsf{m}_1 \cdot \mathsf{m}_2 \cdot \mathsf{m}_3$ under sk.

1. [Multiply.] Compute $a(S) := \mathsf{ct}_1(S) \cdot \mathsf{ct}_2(S) \cdot \mathsf{ct}_3(S) = \sum_{s=0}^{3} a_s S^s$.
2. [INTT.] Convert $a(S)$ to coefficient form.
3. [Rescale.] Compute $a'(S) := \lfloor q_\ell^{-1} \cdot a(S) \rceil$.
4. [Key-switch.] Apply Algorithm 2 with $r_1 = a'_2$ and $r_2 = a'_3$; the result is an affine ciphertext $\mathsf{ct}(S) \in \mathcal{R}_Q[S]$.
5. [Add.] Compute $\mathsf{ct}'(S) := a'_0 + a'_1 S + \mathsf{ct}(S)$.
6. [Rescale.] Output the ciphertext $\mathsf{ct}_{\texttt{Mult}}(S) = \lfloor q_{\ell-1}^{-1} \cdot \mathsf{ct}'(S) \rceil$.

Theorem 4. *Let* $\mathsf{ct}_i(S)$ *be an encryption of* $\mathsf{m}_i + e_i$ *under* sk*, for* $1 \le i \le 3$*. If the conditions in the statement of Theorem 2 hold, then Algorithm 4 returns a ciphertext* $\mathsf{ct}_{\texttt{Mult}}(S)$ *such that*

$$\mathsf{ct}_{\texttt{Mult}}(\mathsf{sk}) \equiv q_{\ell-1}^{-1} q_\ell^{-1} \prod_{i=1}^{3} \mathsf{m}_i + \varepsilon_{\text{tm}} \pmod{Q},$$

where $\varepsilon_{\mathrm{tm}} = q_{\ell-1}^{-1}\left(q_\ell^{-1}((\mathsf{m}_1+e_1)(\mathsf{m}_2+e_2)(\mathsf{m}_3+e_3)-\mathsf{m}_1\mathsf{m}_2\mathsf{m}_3)+e_R^{(1)}+\varepsilon\right)+e_R^{(2)}$*, and the error term* ε *is defined in Theorem 2. Moreover, if* $\|\mathsf{m}_i\|_\infty < \nu$ *and* $\|e_i\|_\infty < B$*, then the bound* $\|\varepsilon_{\mathrm{tm}}\|_\infty < q_{\ell-1}^{-1}q_\ell^{-1}((\nu+B)^3-\nu^3+\frac{1}{2}(k+\|\mathsf{sk}\|_1 k))+(2q_{\ell-1})^{-1}+1$ *holds.*

Proof. Step 1 produces a polynomial $a(S) \in \mathcal{R}_Q[S]$ that decrypts to

$$a(\mathsf{sk}) = \prod_{i=1}^{3} \mathsf{ct}_i(\mathsf{sk}) = \prod_{i=1}^{3}(\mathsf{m}_i + e_i). \tag{8}$$

Step 3 produces another polynomial $a'(S) \in \mathcal{R}_Q[S]$ that decrypts to

$$a'(\mathsf{sk}) = q_\ell^{-1}\prod_{i=1}^{3}(\mathsf{m}_i+e_i) + e_R^{(1)} \mod Q, \tag{9}$$

where $e_R^{(1)}$ is the rounding error introduced in that same step. Step 4 returns a ciphertext $\mathsf{ct}(S)$ that, by Theorem 2, opens to $\sum_{i=2}^{3} a_i' \cdot \mathsf{sk}^i + \varepsilon$ under the secret key sk, where ε is the error introduced in BKS. Step 5 produces $\mathsf{ct}'(S)$, which opens to

$$\mathsf{ct}'(\mathsf{sk}) := a_0' + a_1'\mathsf{sk} + \mathsf{ct}(\mathsf{sk}) = \sum_{i=0}^{3} a_i' \cdot \mathsf{sk}^i + \varepsilon = a'(\mathsf{sk}) + \varepsilon$$

as desired. Step 6 finally outputs the ciphertext $\mathsf{ct}_{\mathtt{Mult}}(S)$, which satisfies $\mathsf{ct}_{\mathtt{Mult}}(\mathsf{sk}) = q_{\ell-1}^{-1}\cdot(a'(\mathsf{sk})+\varepsilon)+e_R^{(2)}$, where $e_R^{(2)}$ is the error due to rescaling. Putting everything together, the noise in the final output of Algorithm 4 is therefore

$$\begin{aligned}\varepsilon_{\mathrm{tm}} &:= \mathsf{ct}_{\mathtt{Mult}}(\mathsf{sk}) - q_{\ell-1}^{-1}q_\ell^{-1}\prod_{i=1}^{c}\mathsf{m}_i = q_{\ell-1}^{-1}\cdot\left(a'(\mathsf{sk})+\varepsilon - q_\ell^{-1}\prod_{i=1}^{c}\mathsf{m}_i\right)+e_R^{(2)} \\ &= q_{\ell-1}^{-1}\left(q_\ell^{-1}\left(\prod_{i=1}^{3}(\mathsf{m}_i+e_i)-\mathsf{m}_1\mathsf{m}_2\mathsf{m}_3\right)+e_R^{(1)}+\varepsilon\right)+e_R^{(2)}.\end{aligned}$$

Finally, we use the bounds $\|\mathsf{m}_i\|_\infty < \nu$ and $\|e_i\|_\infty < B$ to finish the proof:

$$\|\varepsilon_{\mathrm{tm}}\|_\infty < q_{\ell-1}^{-1}q_\ell^{-1}((\nu+B)^3-\nu^3) + q_{\ell-1}^{-1}(e_R^{(1)}+\varepsilon)+e_R^{(2)}. \qquad \square$$

Noise Comparison. We now compare the noise bound of Theorem 4 for our ternary multiplication scheme (Algorithm 4) with that of the standard approach for multiplying three ciphertexts in sequence. Given three affine ciphertexts, the standard approach applies Algorithm 3 twice in sequence; its noise accumulates as follows:

1. Multiplication of $\mathsf{ct}_1, \mathsf{ct}_2$: Algorithm 3 produces a ciphertext $\mathsf{ct}_{\mathtt{Mult}}(S)$ at level $\ell-1$ that opens to $q_\ell^{-1}\mathsf{m}_1\mathsf{m}_2$ with an error $\varepsilon_{\mathrm{bm}}$, defined in Theorem 3.

2. Multiplication of $\mathsf{ct}_{\mathtt{Mult}}(S)$ and ct_3: Algorithm 3 produces a ciphertext $\mathsf{ct}'_{\mathtt{Mult}}(S)$ that opens to $q_{\ell-1}^{-1} q_\ell^{-1} \prod_{i=1}^{3} \mathsf{m}_i$. Apply Theorem 3 with $\mathsf{m}'_1 = q_\ell^{-1}(((\mathsf{m}_1 + e_1)(\mathsf{m}_2 + e_2) - \mathsf{m}_1\mathsf{m}_2) + \varepsilon) + e_R^{(1)}$, $\mathsf{m}'_2 = m_3 + e_3$, and $\ell' = \ell - 1$ to obtain

$$\varepsilon_{\mathrm{dbm}} = q_{\ell-1}^{-1}\left[(q_\ell^{-1}\left[((\mathsf{m}_1 + e_1)(\mathsf{m}_2 + e_2) - \mathsf{m}_1\mathsf{m}_2) + \varepsilon\right] + e_R^{(1)})(\mathsf{m}_3 + e_3) + \tilde{\varepsilon}\right] + e_R^{(2)}.$$

Comparing $\varepsilon_{\mathrm{dbm}}$ with the ternary multiplication error $\varepsilon_{\mathrm{tm}}$, which is

$$\varepsilon_{\mathrm{tm}} = q_{\ell-1}^{-1}\left(q_\ell^{-1}\left((\mathsf{m}_1 + e_1)(\mathsf{m}_2 + e_2)(\mathsf{m}_3 + e_3) - \mathsf{m}_1\mathsf{m}_2\mathsf{m}_3\right) + e_R^{(1)} + \varepsilon\right) + e_R^{(2)},$$

we observe that there is only one KS error ε instead of two $(\varepsilon, \tilde{\varepsilon})$, as was expected, which in turn is only multiplied by the scaling factor. In the expression for $\varepsilon_{\mathrm{dbm}}$, the first KS error is multiplied by the message m_3 and other crossed terms that are avoided in our process. The message m_3 in $\varepsilon_{\mathrm{dbm}}$ is also multiplied by the first rounding error, while in $\varepsilon_{\mathrm{tm}}$ no error is multiplied by any message.

On the other hand, the main disadvantage of ternary multiplication is that it requires additional key material, since the evaluation key produced in Algorithm 1 is of size $2 \times 3d$ instead of $2 \times 2d$. In most cases, key generation forms part of precomputation, so this does not affect latency. But it requires a larger amount of memory, because the polynomial $a(S)$ contains an extra component.

4.3 Optimal Multiplication Trees

In this section, we combine the low-order multiplication algorithms from Section 4.2 into an optimal procedure for multiplying an arbitrary number n of ciphertexts. Suppose we are given a heuristic function $w(l, k)$ that estimates the cost of multiplying k affine ciphertexts at level l, where $1 \le k \le 3$, $l \ge k - 1$, $w(l, 1) = 0$. The cases $k = 2$ and $k = 3$ correspond respectively to algorithms 3 and 4. Here we use the term "cost" as a synonym for latency, although other metrics such as memory usage may be considered as well. The values of $w(l, k)$ should depend on the particular implementation of the CKKS relinearization and rescaling primitives; the following section explains how to obtain an appropriate w empirically, and provides examples under typical 128-bit-secure parameter sets for the Lattigo library.

An *oriented b-ary tree* is defined as a set of nodes that is either empty or consists of a root and b disjoint oriented b-ary trees. Let $l_1 \ge l_2 \ge \cdots \ge l_n$ be the levels of all ciphertexts whose product is to be computed. The entire product calculation may be represented by an oriented tree with n external nodes, where each internal node represents the product of its children, relinearized and rescaled back to approximately their original scale. If the internal nodes correspond to multiplications of arities $k_1, \ldots, k_m \in \{2, 3\}$ at respective levels $l'_1, \ldots, l'_m$, our problem is to find an oriented b-ary tree

1. whose root delivers the final result at the highest possible level $t^\star$, and
2. that minimizes the sum $\sum_j w(l'_j, k_j)$ in case of ties.

Notice that the first condition is *not* equivalent to finding a tree of minimal height, because a node with $k > 0$ inputs consumes $k - 1$ levels.

Before we address this problem, let us consider the simpler task of finding a minimum cost tree with an output level that satisfies some prescribed lower bound t. This is a natural candidate for the dynamic programming technique [8], due to the crucial fact that all subtrees of a minimum cost tree feature minimum cost. For convenience, we write $L_{i:j} = (l_i, l_{i+1}, \ldots, l_j)$ for $i \le j$, and $L = L_{1:n}$. The idea is to partition the sequence L of input levels into as many as b consecutive subsequences in all possible manners, to repeat this process recursively on the subsequences of each partition, and to keep a partition of minimum total cost; any infeasible subproblem whose target level is too high is discarded. Algorithm 5 details the search process for the case of ternary trees ($b = 3$). It records the solution to each subproblem upon first encounter, so that later occurrences of the same subproblem may be handled in constant time. In principle, the procedure may be generalized to arbitrary values of b—for example, to n-ary trees by replacing steps 2 and 3 by a single step that visits all nonempty proper subsets of $\{1, 2, \ldots, n-1\}$. However, our discussion in Section 4.2 shows that four-input ciphertext multiplications are already impractical, so restricting consideration to the case $b = 3$ is justified.

Algorithm 5: Minimum Cost Ternary Multiplication Trees

Given nonnegative integers $l_1 \ge l_2 \ge \cdots \ge l_n$, t, and a cost function $w(l, k)$, this algorithm constructs a tree T of minimum total cost c for obtaining the product of affine ciphertexts at levels $l_1, \ldots, l_n$ as an affine ciphertext at a level $\ge t$; when no such tree exists, the algorithm outputs (Λ, ∞). It uses a global hash table A to remember the values recorded in step 4 across all its invocations.

1. [Done?] Terminate with $(T, c) := (S, 0)$ if $n = 1$ and $l_1 \ge t$, where S is the 1-node tree, or with $(T, c) := A[L, t]$ if defined. Set $(T, c) := (\Lambda, \infty)$.
2. [Two-splits.] For $i = 1, 2, \ldots, n-1$; do: Apply Algorithm 5 with $(L_{1:i}, t+1)$ and $(L_{(i+1):n}, t+1)$ in place of (L, t) to obtain (T_1, c_1) and (T_2, c_2), respectively. Set $c' := c_1 + c_2 + w(2, t+1)$, and if $c' < c$, set $c := c'$ and T to the tree whose root has children $\{T_1, T_2\}$.
3. [Three-splits.] For $i = 1, 2, \ldots, n-2$ and $j = i+1, i+2, \ldots, n-1$; do: Apply Algorithm 5 with $(L_{1:i}, t+2)$, $(L_{(i+1):j}, t+2)$, and $(L_{(j+1):n}, t+2)$ in place of (L, t) to obtain (T_1, c_1), (T_2, c_2), and (T_3, c_3), respectively. Set $c' := c_1 + c_2 + c_3 + w(3, t+2)$, and if $c' < c$, set $c := c'$ and T to the tree whose root has children $\{T_1, T_2, T_3\}$.
4. [Memoize.] Terminate with $A[L, t] := (T, c)$.

The following result provides an upper bound on the performance of Algorithm 5:

Theorem 5. *The worst-case running time of Algorithm 5 is at most proportional to*

$$\sum_{m=2}^{n} \underbrace{(n-m+1)}_{\text{level slices}} \underbrace{(n-m+1)}_{\text{target levels}} \underbrace{\binom{m}{2}}_{\text{calls}} = \frac{n^5}{60} + \frac{n^4}{24} + O(n^2). \tag{10}$$

Proof. We first show that every call corresponds to a pair $(L_{i:j}, t')$ with $t' + n' \leq t + n$, where $n' = j - i + 1$. The claim is obviously true for the initial call $(L_{1:n}, t)$. Every nonterminal call partitions $L_{i:j}$ into k consecutive subsequences of lengths $n_1, \ldots, n_k > 0$, all of which are then associated with the target level $t' + k - 1$. Since the lengths are positive and sum to n', we have $n_l \leq n' - k + 1$ for $1 \leq l \leq k$. It follows by induction that $(t' + k - 1) + n_l \leq (t' + k - 1) + (n' - k + 1) = t' + n' \leq t + n$ as desired.

The first two factors of each term on the left-hand side of (10) correspond to the two indices of table A, namely the number of subsequences $L_{i:j}$ of length m, and the maximum number of target levels per subsequence (which follows from $t' - t \leq n - m$); the third factor is the number of recursive calls across steps 2 and 3 required to populate each entry of A. □

Algorithm 5 attains the worst-case bound (10) whenever no input level appears more than twice. When all the input levels are equal, the running time drops to $O(n^4)$ thanks to better reutilization of the entries in Table A. In any case, it will usually be possible to generate minimum cost trees for all n-ary multiplications appearing in a CKKS circuit during precomputation, because $\log n$ is typically bounded by circuit depth. For larger n, future work could extend Algorithm 5 to perform bootstrap placement [41].

We are now ready to fully solve the general problem stated above:

Theorem 6. *The maximum target level $t^\star$ for a sequence of input levels $l_1, l_2, \ldots, l_n$ is*

$$t^\star = \max\left\{t \geq 0 \;\middle|\; 2^t \textstyle\sum_i 2^{-l_i} \leq 1\right\} = -\left\lceil \log\left(\textstyle\sum_i 2^{-l_i}\right)\right\rceil. \tag{11}$$

Proof. We may assume without loss of generality that the multiplication tree is binary: replacing a node with three children by two nodes with two children preserves the depth consumption of the tree (but may change its total cost). Let $h_1, \ldots, h_n$ be the distances from the root of the tree to each input node. Note that the final output arrives at level $t = \min_{1 \leq i \leq n}(l_i - h_i)$, because every internal node consumes exactly one level. The Kraft–McMillan inequality [28, 30] states that such a tree exists if, and only if $\sum_i 2^{-h_i} \leq 1$. The stated inequality follows because $h_i \leq l_i - t$. □

Intuitively, the maximum level (11) is obtained by repeatedly multiplying the inputs at the highest level; inputs at lower levels are placed closer to the root. In the special case in which all inputs are at some common level l, we have $t^\star = l - \lceil \log n \rceil$, which is achieved by the complete binary tree with n leaves. Of course, if there exists a ternary tree with the same depth consumption and a lower cost than this binary tree, Algorithm 5 will prefer it.

4.4 Evaluation

In this section, we shall conduct experiments with implementations of the binary, ternary, and general ciphertext multiplication algorithms built on top of Lattigo v6.2.0 [17]. We first study the precision and runtime of the first two operations in isolation, and fit a cost function $w(l, k)$ that models their latency as a function of the input level l and arity k. We then use this cost function to generate optimal multiplication trees for n input ciphertexts, and compare their precision and latency against the standard binary tree approach.

Unless otherwise stated, we take ring dimension $N = 2^{16}$, modulus $Q = q_0 \cdots q_L$ with $L = 10$, $\log q_0 \approx 55$, $\log q_i \approx 40$ for $i \geq 1$, auxiliary primes such that $\log p_i \approx 61$, and ternary secret keys with Hamming weight 192. The number of auxiliary primes depends on the number d of digits in the gadget decomposition used for KS. Every value of d that we consider maintains the total modulus size $\log QP$ below 1 550, guaranteeing 128-bit security [9].[4] The experiments run on an AMD EPYC 9645 processor with base and boost frequencies of 2.3 and 3.7 GHz, respectively, and SMT disabled. Each thread is pinned to a single core to avoid NUMA effects. Main memory operates at 6,400 MT/s, and no disk accesses occur during measurement. All programs are compiled with Go 1.27.

Binary and Ternary Multiplications. We start by comparing the precision and runtime of the two strategies for multiplying three affine ciphertexts $\mathsf{ct}_1, \mathsf{ct}_2, \mathsf{ct}_3$ at the same level l, using Algorithms 3 and 4. Let the ciphertexts be fresh encryptions of vectors of random complex numbers uniformly and independently distributed on the unit circle. We measure noise growth by computing the bit precision $-\log|\tilde{v}_i - v_i|$ of each slot $\tilde{v}_i$ of the decrypted-and-decoded output vector $\tilde{v}$, where v_i is the corresponding entry of the exact answer.

Table 1 shows the minimum, average, and maximum precision over all slots of $\tilde{v}$. Each result is the average of ten independent runs. The fact that the precision of both algorithms is the same for all d indicates that the error introduced by the gadget products (steps 1–5 of Algorithm 2, excluding the final modulus lowering) is insignificant. At first glance, the different positioning of the rescalings during the multiplication also seems to pose no effect on precision. To test this hypothesis, we isolate the noise due to the plaintext encoding and the modulus lowerings by making the input ciphertexts carry no fresh error. Our batched approach then retains 0.48 more bits of minimum, average, and maximum precision than the sequential approach, for all d. However, if the rescaling steps are removed from both algorithms, they produce almost-identical outputs. A similar situation occurs when the input entries have uniformly distributed real and imaginary parts in an interval $[-r, r]$: with rescalings, Algorithm 4 offers 0.10, 0.34, 0.73, 0.89, and 0.58 further bits of precision for $r = 1/2, 1, 2, 4$, and 8, respectively; without rescalings, it ties the standard approach. This indicates that the noise contribution of rescaling dominates up to $r \approx 4$, after which the encoding error

[4] This parameter set is typically used in private DNN inference workloads [16, 6].

		$(\mathsf{ct}_1 \cdot \mathsf{ct}_2) \cdot \mathsf{ct}_3$					$\mathsf{ct}_1 \cdot \mathsf{ct}_2 \cdot \mathsf{ct}_3$					
		Precision (bits)			Time (ms)		Precision (bits)			Time (ms)		
d	l	min	avg	max	avg	dev	min	avg	max	avg	dev	Speedup
	2	26.3	29.6	38.0	44.8	0.4	26.3	29.8	38.3	36.8	1.5	1.22
	3	26.2	29.6	38.4	54.0	1.3	26.4	29.8	37.9	45.2	0.3	1.19
3	4	26.2	29.6	37.4	71.5	1.5	26.4	29.8	38.1	54.4	1.2	1.31
	7	26.2	29.6	38.0	112.6	2.1	26.4	29.8	38.2	100.8	0.9	1.12
	10	26.2	29.6	37.7	171.3	1.7	26.3	29.8	37.6	158.3	1.6	1.08
	2	26.2	29.6	37.7	39.7	0.5	26.3	29.8	38.1	32.9	0.8	1.20
	3	26.2	29.6	37.8	53.8	0.2	26.3	29.8	38.0	41.4	0.3	1.30
4	4	26.1	29.6	37.7	70.2	1.1	26.4	29.8	38.4	62.0	0.3	1.13
	7	26.1	29.6	37.4	120.6	0.9	26.4	29.8	37.8	110.9	0.6	1.09
	10	26.1	29.6	37.5	182.0	1.5	26.4	29.8	37.6	171.3	1.5	1.06
	2	26.3	29.6	37.5	40.3	0.3	26.4	29.8	37.5	33.3	0.4	1.21
	3	26.2	29.6	38.1	59.8	0.6	26.4	29.8	38.0	50.9	0.7	1.17
11	4	26.3	29.6	38.2	82.2	0.4	26.5	29.8	37.8	71.2	0.2	1.15
	7	26.2	29.6	38.1	167.6	0.4	26.4	29.8	38.0	152.4	0.3	1.10
	10	26.2	29.6	38.1	283.0	1.0	26.5	29.8	38.3	262.3	0.9	1.08

Table 1: The two strategies for multiplying three affine ciphertexts $\mathsf{ct}_1, \mathsf{ct}_2, \mathsf{ct}_3$ at various levels l, using d-digit gadget decompositions for KS. The ciphertexts encrypt vectors of uniformly random complex numbers on the unit circle.

begins to pose a noticeable effect on the output. Repeating the experiment with zero-mean normal distributions yields the same results.

The reason our rescaling policy does not improve precision in Table 1 is that fresh errors have a much bigger magnitude than encoding errors, and they grow at the same rate during the multiplication process. Accordingly, the error terms in Theorem 4 that involve fresh errors are usually greater than the rounding errors of rescaling and KS. Note, however, that there are special cases where just the opposite is true, and thus Algorithm 4 comes out ahead. For example, if $r = 1$ but the (complex) entries of the third input ct_3 lie in a wider square $[-10, 10]^2$ than the rest, we observe an average precision gain of half a bit over the standard approach.

Now let us shift our attention to running time. The sequential and batched methods for multiplying three ciphertexts involve almost the same basic operations, but importantly they differ at the levels at which they are carried out. The sequential method relinearizes the product $\mathsf{ct}_1 \cdot \mathsf{ct}_2$ at the original input level l, and then it rescales the two components of the resulting affine ciphertext. By contrast, the batched method "moves" that rescaling before the computation of the two gadget products, which is therefore performed one level lower. Recall that a digit encompasses $\alpha = \lceil (L+1)/d \rceil$ limbs, hence a polynomial at level l is decomposed into $\delta(l) = \lceil (l+1)/\alpha \rceil$ digits. It follows that the sequential and batched approaches operate on $\delta(l) + \delta(l-1)$ and $2\delta(l-1)$ digits in total during the gadget products, respectively. These quantities differ by one digit precisely when α divides l, which occurs in just the two cases with the largest speedups in Table 1, namely $(d, \alpha, l) = (3, 4, 4)$ with 1.31, and $(4, 3, 3)$ with 1.30.

The sequential approach also needs to lower two affine polynomials from $\tilde{Q}$ to Q during the two relinearizations, whereas the batched algorithm sums the two gadget products modulo $\tilde{Q}$ before bringing the result down to $\mathcal{R}_Q$. In compensation, the batched approach multiplies three affine polynomials at once instead of two pairs in succession, and it applies the first rescaling to four polynomials in $\mathcal{R}_Q$ instead of two. The first drawback is negligible, because with Karatsuba's method [26], the number of $\mathcal{R}_Q$ multiplications over the NTT domain only increases from eight to nine. The additional rescalings, however, have a relative cost from roughly 5.4% for $(d, l) = (11, 10)$ up to 12.3% at $l = 2$.

Altogether, our batched method generally achieves its largest gains when l is small, because the limb that it removes represents a larger fraction of the work done by the sequential method. Table 1 reports a speedup of 1.21 at $l = 2$ for all d, and lower speedups between 1.06 and 1.08 at $l = 10$. And when $d = 11$ every digit consists of a single limb, so the speedup decreases monotonically with l. In any case, the batched approach is faster across all tested configurations.

Cost Functions. The previous discussion suggests that a cost function $w(l, k)$ that characterizes the relative latency of binary and ternary multiplication ought to depend not only on the level l of the inputs, but also on the number of digits $\delta(l)$. In Algorithm 3, the cost of the polynomial product, the lowering from $\tilde{Q}$ to Q, and the rescaling, is roughly linear in the number $l + 1$ of limbs; the work

involved in the gadget product is approximately proportional to the number $\delta(l)$ of digits times the number $l+1+\alpha$ of limbs to which each of them is then extended, namely the original limbs together with the $\alpha = \lceil (L+1)/d \rceil$ auxiliary counterparts. This gives us the first ansatz

$$w(l,2) = \beta_2(l+1) + \gamma_2\delta(l)(l+1+\alpha), \qquad l \geq 1. \tag{12}$$

The main difference in the case $k=3$ is that Algorithm 4 computes the two gadget products at level $l-1$. Therefore, our second ansatz shall be

$$w(l,3) = \beta_3(l+1) + \gamma_3\delta(l-1)(l+\alpha), \qquad l \geq 2. \tag{13}$$

We fit (12) and (13) by least squares to the running times of our implementations of the binary and ternary multiplication algorithms, averaged over ten runs, for $d=3$ and all $k-1 \leq l \leq 10 = L$. Normalizing so that the most expensive cost $w(L,2)$ equals 1, we obtain the coefficients $(\beta_2, \gamma_2) = (0.0705, 0.00523)$ and $(\beta_3, \gamma_3) = (0.0944, 0.0174)$, with normalized root mean square errors of 3.9% and 0.8%, respectively. We remark that this particular heuristic depends on the latency of KS and rescaling in Lattigo, and hence it applies only to the cryptographic and hardware setup described above. Other parameter sets or implementations of these CKKS primitives (such as OpenFHE's [4]) require a new fit.

Multiplication Trees. We conclude this section by constructing minimum cost trees for multiplying n ciphertexts at various input levels l. When n is a power of 2, we have seen that the perfect binary tree is the only solution that minimizes depth consumption. For other values of n, however, the shape of a minimum cost tree remains unclear.

For the sake of exposition, we restrict consideration to instances where the target level $t = t^\star$ is the highest attainable; that is, to cases where our approach can be compared with the standard binary tree reduction. We merely mention that there are applications of Algorithm 5 where the target level may be smaller than $t^\star$. For example, if bootstrapping is known to be required immediately after the entire product computation, then it is wise to set $t = 0$; in this case, Algorithm 5 may trade additional level consumption for more of the cheaper ternary multiplications.

Using the cost functions fitted above for $d=3$, and assuming that all inputs arrive at the same level l, Algorithm 5 outputs the minimum cost trees illustrated in Table 2. As expected, the perfect binary tree is obtained when n is a power of two. Just as important is the fact that every other tree contains at least one ternary node.

The most interesting case is $n=9$, where the search yields different trees for $l = \lceil \log n \rceil$ and $l = 10$. Notice that both trees consume the least possible number of levels, but the one for $l = \lceil \log n \rceil$ it performs two binary multiplications (orange) near the root, hence the ternary multiplication (blue) in the left subtree of the root is performed one level lower. At $l = \lceil \log n \rceil$, the pair of binary nodes

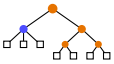
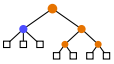
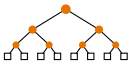
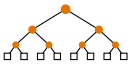
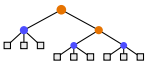
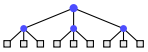
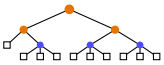
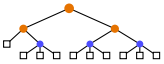
		$l = \lceil \log n \rceil$				$l = 10$			
		# Multiplications				# Multiplications			
n	Depth	Binary	Ternary	Total	Shape	Binary	Ternary	Total	Shape
2	1	1	0	1		1	0	1	
3	2	0	1	1		0	1	1	
4	2	3	0	3		3	0	3	
5	3	2	1	3		2	1	3	
6	3	1	2	3		1	2	3	
7	3	4	1	5		4	1	5	
8	3	7	0	7		7	0	7	
9	4	2	3	5		0	4	4	
10	4	3	3	6		3	3	6	
192	8	63	64	127	(1a)	63	64	127	(1a)

Table 2: The optimum trees produced by Algorithm 5 when given all-equal input levels l, the target level $t = l - \lceil \log n \rceil$, and the cost function $w(l, k)$ found above for $d = 3$.

costs only $w(1, 2) + w(2, 2) - w(2, 3) = 0.029$ more than the ternary node that they replace, whereas the level reduction saves $w(4, 3) - w(3, 3) = 0.099$ units of cost. By contrast, at $l = 10$ the pair costs 0.249 more, but the single-level reduction saves only 0.133. In practice, the tree proposed for $l = 10$ attains a modest single-thread speedup of 1.5% over the tree for $l = \lceil \log n \rceil$ when they are both evaluated at $l = 10$. Nevertheless, this example shows how the input level parameters may affect the final tree. This dependence becomes more pronounced for larger n, where the search space is naturally larger.

To validate the precision and runtime of our approach, we generate minimum cost trees for all $3 \leq n \leq 10$ and $\lceil \log n \rceil \leq l \leq 10$, and we compare them with their corresponding standard binary tree reductions. The gains in minimum, average, and maximum precision range from 0.06 to 0.16 bits, with the largest gains stemming from the trees with a larger fraction of ternary nodes. As noted earlier, total noise is dominated by the fresh error of the operands, hence the different rescaling placements pose only a minor influence on precision. The

improvement would be larger if the input cleartexts were to follow a more uneven distribution.

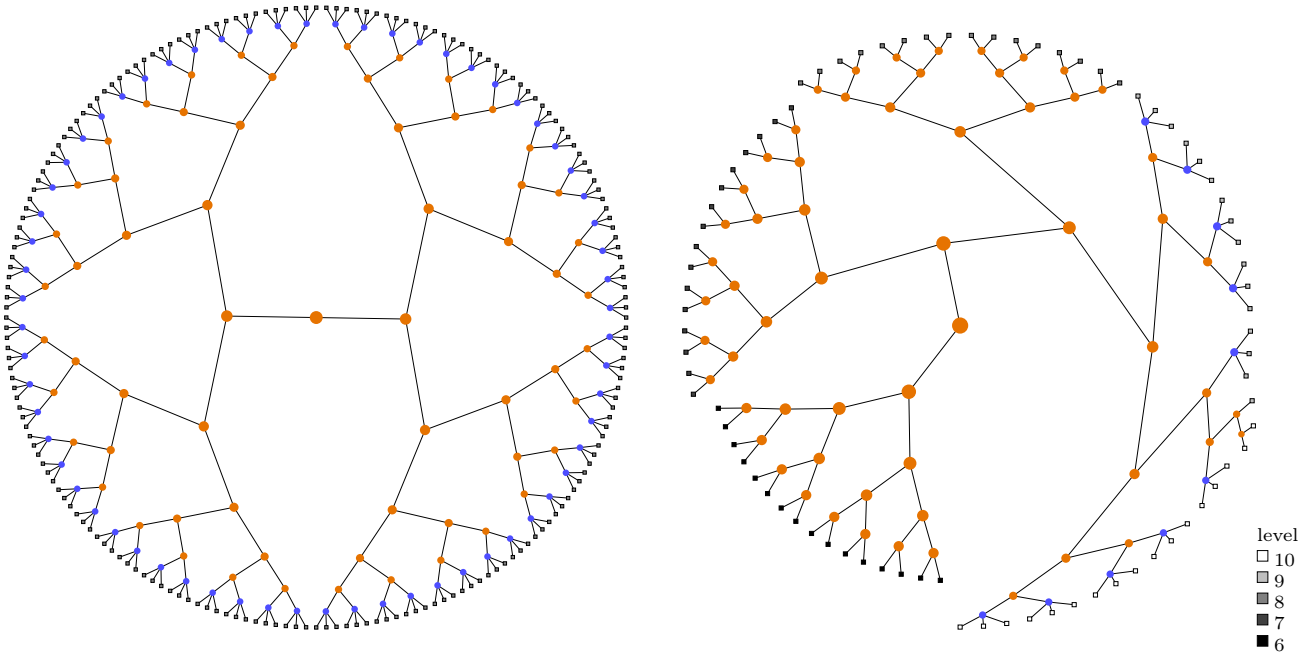
Table 3 shows that judicious placement of ternary nodes may speed up n-ary ciphertext multiplication by up to 30% at low input levels. The single-thread speedup generally decreases as l grows, but some benefit remains at higher levels. Like precision, speedups are best in trees with a large share of ternary nodes.

n \ l	1	2	3	4	5	6	7	8	9	10
3	1.184	1.200	1.307	1.133	1.124	1.112	1.205	1.093	1.095	
4	–	–	–	–	–	–	–	–	–	
5		1.125	1.156	1.073	1.063	1.058	1.100	1.058	1.041	
6		1.142	1.251	1.110	1.100	1.085	1.181	1.073	1.070	
7		1.059	1.084	1.042	1.038	1.026	1.064	1.033	1.033	
8			–	–	–	–	–	–	–	–
9				1.151	1.120	1.108	1.083	1.100	1.077	1.076
10				1.164	1.104	1.072	1.078	1.100	1.076	1.048

Table 3: The mean speedup of the optimum multiplication trees generated by Algorithm 5 with $d = 3$, relative to the binary tree approach.

In general, our scheme works best when the number n of inputs has a "favorable" factorization. For example, consider the multiplication circuit appearing at the end of the SHIP bootstrapping circuit [12]. In this case, n is equal to the Hamming weight of the ternary secret key, which should be at least 192 to attain 128-bit security with our modulus size [9]. Since $192 = 2^6 \cdot 3$, any tree consisting of six binary layers and a single ternary layer attains the minimum depth consumption $\lceil \log 192 \rceil = 8$. Algorithm 5 finds that the unique tree of minimum cost places 64 ternary nodes in its bottom layer, and combines their outputs via binary multiplication. The rationale of this placement is that even if the individual savings of a ternary node near the inputs are smaller—given $w(l, 3)/w(l, 2) \approx 1.72$ for $l = 8$, the cost of two binary multiplications is close to that of a single ternary one—their sheer number is larger. The benchmarks in Figure 1a show that this tree outperforms the complete binary tree reduction by up to 19.9%, even when evaluated with multiple threads.

Nevertheless, Algorithm 5 remains useful when the prime factorization of n does not suggest an obvious placement, or when the input ciphertexts are at different levels. A case in point is the tree drawn in Figure 1b for $n = 81$ inputs at levels selected uniformly at random between 6 and 10. Its shape would be hard to predict by hand, and it attains speedups of up to 10.4% on a multi-threaded system.



Threads	1	2	4	8	16	32	64
Speedup	1.141	1.144	1.157	1.158	1.163	1.199	1.185

(a) A 192-ary multiplication, as appearing in a typical 128-bit secure instantiation of the SHIP bootstrapping algorithm [12].

Threads	1	2	4	8	16	32	64
Speedup	1.102	1.104	1.093	1.072	1.081	1.066	1.070

(b) A multiplication of 81 ciphertexts at uniformly distributed levels between 6 and 10.

Fig. 1: Solutions produced by Algorithm 5 for two multiplication problems with large input counts n, and their speedups over the standard binary approach.

5 Double-Hoisted Matrix-Vector Multiplication

As another example of the application of the batched key-switching technique to a practical problem, we consider the homomorphic evaluation of a matrix–vector product $\boldsymbol{M}\boldsymbol{v}$, where $\boldsymbol{M} \in \mathbb{C}^{n\times n}$ is a cleartext matrix and $\boldsymbol{v} \in \mathbb{C}^n$ is the slot vector associated with a CKKS ciphertext. This operation arises frequently in practice, since, for example, the coefficient-to-slots and slot-to-coefficients steps in a CKKS bootstrapping circuit (which raises the level of a ciphertext to allow further computation) are linear transformations [10]. Also, FHE inference engines tend to express the linear layers of a DNN as matrix–vector products [16, 34].

Halevi-Shoup [22] observed that the product $\boldsymbol{M}\boldsymbol{v}$ may be expressed in terms of the *generalized diagonals* of $\boldsymbol{M}$, namely all vectors $\left[(m_{i,i+k \pmod n})_{0\le i<n}\right]$ for $0 \le k < n$. Given plaintexts $m_0, \ldots, m_{n-1} \in \mathcal{R}$ encoding these diagonals, the formula $\sum_{k=0}^{n-1} m_k \cdot \mathsf{Rotate}(\mathsf{ct}(S), k)$ converts an encryption $\mathsf{ct}(S)$ of $\boldsymbol{v}$ into one

of $\boldsymbol{Mv}$. The operation $\mathsf{Rotate}(\mathsf{ct}(S), k)$ applies the automorphism $\phi_k \colon X \to X^{5^k} \pmod{X^N + 1}$ to a ciphertext $\mathsf{ct}(S)$, which corresponds to a circular shift of its underlying cleartext vector by k positions to the left. Afterwards it uses key-switching in order to obtain a valid encryption under the original secret key sk rather than under $\phi_k(\mathsf{sk})$. This last step makes the computation of the rotations the most computationally costly part in the homomorphic evaluation of $\boldsymbol{Mv}$. Fortunately, the number of rotations can be brought down from $\Theta(n)$ to $\Theta(\sqrt{n})$ by considering a *baby-step giant-step* (BSGS) decomposition [23]: write $n = n_\mathrm{b} n_\mathrm{g}$ for some integers n_b and n_g, construct the ciphertexts $\mathsf{ct}_i(S) := \mathsf{Rotate}(\mathsf{ct}(S), i)$ for $0 \leq i < n_\mathrm{b}$, and compute

$$\sum_{0 \leq j < n_\mathrm{g}} \phi_{n_\mathrm{b} j} \left(\sum_{0 \leq i < n_\mathrm{b}} \phi_{-n_\mathrm{b} j}(m_{n_\mathrm{b} j + i}) \cdot \mathsf{ct}_i(S) \right). \tag{14}$$

This procedure is a good candidate for the hoisting optimization discussed in Section 3: the n_b rotations of $\mathsf{ct}(S)$ may be constructed in batch from the same gadget decomposition of $\mathsf{ct}(S)$, thereby reducing the number of NTT operations and RNS reconstructions involved. Bossuat *et al.* [9] realized that hoisting may be applied a second time, by delaying the lowering of the rotations to $\mathcal{R}_Q$ until these are grouped into each term of the outer sum in Equation (14). With this method, the total number of modulus lowerings goes down from $n + n_\mathrm{g}$ to $n_\mathrm{g} + 1$, which saves time in spite of carrying the plaintext-ciphertext multiplications over the extended ring $\mathcal{R}_{\tilde{Q}}$.

Bossuat *et al.*'s approach is the current state of the art for evaluating dense matrix–vector products homomorphically, and it is part of the bootstrapping circuits provided by popular CKKS libraries such as Lattigo [17] and OpenFHE [4]. Although its execution time and space requirements have been analyzed in detail (both theoretically and empirically), its precise impact on noise growth remains unknown. In order to investigate this question, we shall now look into the "baby" and "giant" stages of the double-hoisted algorithm more deeply. Among other things, this will let us to confirm the rule of thumb that the noise introduced by the coefficient-to-slots and slot-to-coefficients steps in the aforementioned bootstrap circuits is negligible.

5.1 Batched Rotations

Under the generalized-diagonals formulation, the first step in applying a linear transformation to an encryption $\mathsf{ct}(S)$ of a vector is to compute the n_b distinct rotations of $\mathsf{ct}(S)$ that appear in the sum (14). This task is amenable to hoisting, and in fact it almost follows the structure of Program (4). (For reasons to be explained shortly, the "main" section of the double-hoisted algorithm introduces less noise when it works with the rotations directly over the extended ring $\mathcal{R}_{\tilde{Q}}$. So here we insist that the output of the "baby" part is not lowered to $\mathcal{R}_Q$ prematurely, i.e., we skip the final modulus lowering step.) Extracting the initial gadget step shared by the key-switches—just as we did to obtain Program (5)—yields Algorithm 6.

Algorithm 6: Hoisted Automorphism Applications [9]

Let $\rho_1, \ldots, \rho_n$ be automorphisms of $\mathcal{R}$, and let ρ_0 be the identity map. Given a ciphertext $\mathsf{ct}(S) = b + aS \in \mathcal{R}_Q[S]$ in coefficient form, and an evaluation key $\boldsymbol{U} \in \mathcal{R}_{\tilde{Q}}^{2\times nd}$ from $(\rho_1(\mathsf{sk}), \ldots, \rho_n(\mathsf{sk}))$ to sk produced by Algorithm 1 for a gadget vector $\boldsymbol{g} \in \mathbb{Z}_{\tilde{Q}}^d$, this algorithm outputs an affine ciphertext $\tilde{\mathsf{ct}}_i(S) \in \mathcal{R}_{\tilde{Q}}[S]$ in evaluation form that approximately decrypts to $P \cdot \rho_i(\mathsf{ct}(\mathsf{sk}))$ under sk, for each $0 \le i \le n$. It uses a precomputed matrix $\boldsymbol{U}'$ with entries $u'_{j,k} = \rho^{-1}_{\lfloor (k-1)/d \rfloor + 1}(u_{j,k})$, for $j \in \{0, 1\}$ and $1 \le k \le nd$.

1. [Decompose.] Compute the gadget decomposition $\boldsymbol{d} \in \mathcal{R}_{\tilde{Q}}^d$ of a with respect to the basis $\boldsymbol{g}$.
2. [Extend.] Find the RNS representation $\tilde{\boldsymbol{d}}$ of $\boldsymbol{d}$ over $\mathcal{R}_{\tilde{Q}}^d$. Perform the same for $P \cdot \mathsf{ct}(S) \in \mathcal{R}_Q[S]$ in order to obtain $\tilde{\mathsf{ct}}_0(S) = \tilde{b} + \tilde{a}S \in \mathcal{R}_{\tilde{Q}}[S]$.
3. [NTT.] Convert $\tilde{\boldsymbol{d}}$ and $\tilde{\mathsf{ct}}_0(S)$ to evaluation form.
4. [Key-switch.] Compute the row vector
$$\left[\tilde{\mathsf{ct}}_1(S) \cdots \tilde{\mathsf{ct}}_n(S)\right] := \left[1 \; S\right] \boldsymbol{U}'(\boldsymbol{I}_n \otimes \tilde{\boldsymbol{d}}) = \left[\left((\boldsymbol{u}'_{0,i} + \boldsymbol{u}'_{1,i} S)\tilde{\boldsymbol{d}}\right)_i\right].$$
Here $\boldsymbol{I}_n$ is the identity matrix of order n, and $\otimes$ denotes the Kronecker product.
5. [Rotate.] For $1 \le i \le n$, set $\tilde{\mathsf{ct}}_i(S)$ to $\rho_i\left(\tilde{b} + \tilde{\mathsf{ct}}_i(S)\right)$.

Step 4 of Algorithm 6, which is in charge of key-switching, constructs ciphertexts that decrypt under the secret keys $\rho_i^{-1}(\mathsf{sk})$ rather than the original sk. Bossuat *et al.* [9] observed that this change from the traditional method of key-switching $\rho_i(a)$ allows us to apply ρ_i only at the final step of the algorithm, thereby reducing the total number of automorphism applications by a factor of 3/2. We prove that the algorithm (and this change) is correct, and give an explicit bound for the noise growth it incurs in Theorem 7.

Theorem 7. *Let $Q = q_0 \cdots q_l$ be a product of coprime integers, let $Q = D_1 \cdots D_d$ be the digit decomposition used to construct a gadget vector $\boldsymbol{g} \in \mathbb{Z}_Q^d$, and let ε_{f} be an upper bound on the ℓ_∞ norm of the error term in a fresh encryption. Each ciphertext $\tilde{\mathsf{ct}}_i(S)$ produced by Algorithm 6 opens to $\tilde{\mathsf{ct}}_i(\mathsf{sk}) \equiv P \cdot \rho_i(\mathsf{ct}(\mathsf{sk})) + \varepsilon \pmod{Q}$, where $\|\varepsilon\|_\infty \le \frac{1}{2} N \varepsilon_{\mathrm{f}} (l+1) \max_{1 \le j \le d} D_j$.*

Proof. The ith block column $\boldsymbol{U}'_i = \begin{bmatrix} \boldsymbol{u}'^T_{0,i} & \boldsymbol{u}'^T_{1,i} \end{bmatrix}^T \in \mathcal{R}_{\tilde{Q}}^{2\times d}$ of $\boldsymbol{U}'$, $1 \le i \le n$, is the result of transforming an evaluation key from $\rho_i(\mathsf{sk})$ to sk into an evaluation key from sk to $\rho_i^{-1}(\mathsf{sk})$. To see why, notice that $\boldsymbol{u}_{1,i}$ and $\boldsymbol{u}_{0,i}$ are replaced respectively by $\boldsymbol{u}'_{1,i} = \rho_i^{-1}(\boldsymbol{u}_{1,i})$ and

$$\begin{aligned} \rho_i^{-1}(\boldsymbol{u}_{0,i}) &= -\rho_i^{-1}(\mathsf{sk}) \cdot \rho_i^{-1}(\boldsymbol{u}_{1,i}) + P \cdot \rho_i^{-1}\left(\rho_i(\mathsf{sk})\right) \cdot \boldsymbol{g}^T + \rho_i^{-1}(\boldsymbol{e}_i) \mod \tilde{Q} \\ &= -\rho_i^{-1}(\mathsf{sk}) \cdot \boldsymbol{u}'_{1,i} + P \cdot \mathsf{sk} \cdot \boldsymbol{g}^T + \boldsymbol{e}'_i \mod \tilde{Q} \end{aligned}$$

in the notation of Algorithm 1, because an automorphism distributes over addition and multiplication. Here and in the sequel we take a ring automorphism to apply

entrywise to matrices and vectors of ring elements. Importantly, the corresponding entries of the terms $\boldsymbol{e}_i$ and $\boldsymbol{e}'_i = \rho_i^{-1}(\boldsymbol{e}_i)$ have the same ℓ_∞ norm, because an automorphism of $\mathcal{R}$ permutes the coefficients of its input without changing their absolute value—doing arithmetic modulo the negacyclic polynomial $X^N + 1$ may at most induce a sign change in the coefficients.

A similar argument shows that the algorithm terminates with

$$\tilde{\mathsf{ct}}_i(S) \equiv \rho_i(\tilde{b} + \left[1\ S\right] \boldsymbol{U}_i \tilde{\boldsymbol{d}}) = \rho_i(\tilde{b}) + \left[1\ S\right] \rho_i(\boldsymbol{U}_i)\rho_i(\tilde{\boldsymbol{d}}) \pmod{Q}.$$

It is not difficult to see that $\rho_i(\tilde{\boldsymbol{d}})$ is the gadget decomposition of $\rho_i(a)$ in the basis $\boldsymbol{g}$. This observation, together with the fact that $\tilde{b}$ is the extension of $P \cdot b$ to $\mathcal{R}_{\tilde{Q}}$, implies that $\tilde{\mathsf{ct}}_i(\mathsf{sk}) = \rho_i(\tilde{b}) + \left(P \cdot \rho_i(\mathsf{sk}) \cdot \boldsymbol{g}^T + \boldsymbol{e}_i\right) \rho_i(\tilde{\boldsymbol{d}}) \bmod Q$ is approximately equal to $P \cdot \rho_i(\mathsf{ct}(\mathsf{sk})) = P \cdot \rho_i(b + a \cdot \mathsf{sk})$ as desired. Continue as in the proof of Theorem 2 to bound the error $|||\varepsilon(\mathsf{sk})|||_\infty = |||\boldsymbol{e}_i \rho_i(\tilde{\boldsymbol{d}})|||_\infty$. □

5.2 The Baby and Giant Steps

We are now ready to analyze the double-hoisted BSGS algorithm in its entirety. All the rotations of vector $\boldsymbol{v}$ needed to evaluate the linear transformation (14) may be obtained by invoking Algorithm 6 with $n = n_\mathrm{b} - 1$ and the automorphisms $\phi_1, \ldots, \phi_{n_\mathrm{b}-1}$, corresponding to rotations by $1, 2, \ldots, n_\mathrm{b} - 1$ slots, respectively. The remaining job in forming an encryption of $\boldsymbol{M}\boldsymbol{v}$ is:

(i) to multiply these rotations together with the original unrotated input by each group of n_b consecutive diagonals of the plaintext matrix $\boldsymbol{M}$; and
(ii) to add the resulting n_g ciphertexts after being rotated by multiples of n_b slots.

Before we formalize the entire procedure, let us give some comments about these two individual steps.

First, it is worth noting that the multiplications in step (i) are carried out over the extended ring $\mathcal{R}_{\tilde{Q}}$. The alternative—as we discussed in the context of Algorithm 6—would be to lower the n_b rotations involved from $\mathcal{R}_{\tilde{Q}}$ to $\mathcal{R}_Q$, and then go on to step (i). Bossuat *et al.* [9] argued that, since the additional cost of the extended multiplications is usually smaller than that of the extra modulus lowerings, it is best to wait until step (i) is complete before going down to $\mathcal{R}_Q$. (Of course, this is no advantage at all when n_g is considerably larger than n_b, because in that case we would be doing *more* reductions. It is customary to set the parameters n_b and n_g such that $8 \leq n_\mathrm{b}/n_\mathrm{g} \leq 16$, for this and other reasons explained in the cited paper.) Working over $\mathcal{R}_{\tilde{Q}}$ is also helpful from the standpoint of noise growth, since this ring provides extra precision and allows for a later modulus lowering operation to effectively reduce the accumulated error by a factor of P.

Step (ii) is another example of a circuit that combines the results of several key-switches into a single ciphertext, and thus a viable candidate for acceleration via the batched Algorithm 2. Note that in this case the source keys are the results

Algorithm 7: Double-Hoisted BSGS Matrix–Vector Multiplication [9]

Let $\tilde{m}_0, \ldots, \tilde{m}_{n-1} \in \mathcal{R}_{\tilde{Q}}$ be plaintexts encoding the generalized diagonals of a matrix $\boldsymbol{M} \in \mathbb{C}^{n \times n}$. Given evaluation keys $\boldsymbol{U}_\mathrm{b}$ from $(\phi_1(\mathsf{sk}), \ldots, \phi_{n_\mathrm{b}-1}(\mathsf{sk}))$ to sk, and $\boldsymbol{U}_\mathrm{g}$ from $(\phi_{n_\mathrm{b}}(\mathsf{sk}), \phi_{2n_\mathrm{b}}(\mathsf{sk}), \ldots, \phi_{(n_\mathrm{g}-1)n_\mathrm{b}}(\mathsf{sk}))$ to sk constructed by Algorithm 1 for a gadget basis $\boldsymbol{g} \in \mathbb{Z}_Q^d$, where n_b and n_g are integers such that $n = n_\mathrm{b} n_\mathrm{g}$, this algorithm turns an affine encryption $\mathsf{ct}(S) \in \mathcal{R}_Q[S]$ of a vector $\boldsymbol{v} \in \mathbb{C}^n$ under sk into an encryption $\mathsf{ct}'(S) \in \mathcal{R}_Q[S]$ of $\boldsymbol{M}\boldsymbol{v}$ under sk. Both $\mathsf{ct}(S)$ and $\mathsf{ct}'(S)$ are in coefficient form; the $\tilde{m}_k$'s are in evaluation form.

1. [Rotate input.] Apply Algorithm 6 with $\boldsymbol{U} = \boldsymbol{U}_\mathrm{b}$ to obtain the rotation $\tilde{\mathsf{ct}}_i(S) \in \mathcal{R}_{\tilde{Q}}[S]$ of $\mathsf{ct}(S)$ by i slots, for all $0 \le i < n_\mathrm{b}$.
2. [Baby steps.] For $0 \le j < n_\mathrm{g}$, set $\tilde{\mathsf{ct}}'_j(S)$ to $\sum_{i=0}^{n_\mathrm{b}-1} \phi_{-n_\mathrm{b} j}(\tilde{m}_{n_\mathrm{b} j + i}) \cdot \tilde{\mathsf{ct}}_i(S)$.
3. [INTT.] Convert $\tilde{\mathsf{ct}}'_j(S)$ to coefficient form, for $0 \le j < n_\mathrm{g}$.
4. [Lower.] Set a_j to $\lfloor P^{-1} \cdot \tilde{a}_j \rceil$ for each $0 \le j < n_\mathrm{g}$, where $\tilde{\mathsf{ct}}'_j(S) = \tilde{b}_j + \tilde{a}_j S$.
5. [Giant steps.] Apply Algorithm 2 with $r_j = \phi_{n_\mathrm{b} j}(a_j)$ for $1 \le j < n_\mathrm{g}$, $\tilde{b} = \sum_{j=1}^{n_\mathrm{g}-1} \phi_{n_\mathrm{b} j}(\tilde{b}_j)$, and $\boldsymbol{U} = \boldsymbol{U}_\mathrm{g}$; add $\tilde{\mathsf{ct}}'_0(S)$ to the result $\mathsf{ct}'(S) \in \mathcal{R}_Q[S]$.
6. [Rescale.] Output the ciphertext $\mathsf{Rescale}(\mathsf{ct}'(S))$.

of applying the automorphisms $\phi_{n_\mathrm{b}}, \phi_{2n_\mathrm{b}}, \ldots, \phi_{(n_\mathrm{g}-1)n_\mathrm{b}}$ to the destination secret key sk.

Algorithm 7 describes the procedure formally. Step 5 is slightly different from its original description by Bossuat *et al.* [9, lines 15 and 16 of Algorithm 6]: As in a regular CKKS rotation, we apply each "giant" automorphism $\phi_{n_\mathrm{b} j}$ to its corresponding polynomial a_j before constructing its gadget decomposition; the original method delays the application of the automorphism until the key-switch of a_j from sk to $\phi_{n_\mathrm{b} j}^{-1}(\mathsf{sk})$ is complete. The reason for preferring the latter strategy when computing the baby rotations (step 1) is that it allows us to generate all the rotations from a single gadget decomposition. But in this case the ciphertext $\tilde{\mathsf{ct}}'_j(S)$ only needs to be rotated once, so it is faster to apply $\phi_{n_\mathrm{b} j}$ to a_j over $\mathcal{R}_Q$ than to the linear part of the key-switched ciphertext over the extended $\mathcal{R}_{\tilde{Q}}$. The proof of Theorem 7 shows that this change poses no effect on noise, hence Theorem 8 remains valid for the original version of the algorithm.

Theorem 8. *Let $Q = q_0 \cdots q_{l-1}$ and $P = p_1 \cdots p_k$ be products of coprime integers, and let $Q = D_1 \cdots D_d$ be the digit decomposition used to construct a gadget vector $\boldsymbol{g} \in \mathbb{Z}_Q^d$. Define $C := \frac{1}{2} N \varepsilon_\mathrm{f} (l+1) \max_{1 \le j \le d} D_j$, where ε_f is an upper bound on the ℓ_∞ norm of a fresh encryption's error. Also let $\mathsf{ct}(S) \in \mathcal{R}_Q[S]$ be a ciphertext that opens to $\mathsf{ct}(\mathsf{sk}) \equiv \mathsf{m} + \varepsilon \pmod{Q}$. If $P > C$, Algorithm 7 outputs a ciphertext that decrypts to $\sum_{k=0}^{n-1} \tilde{m}_k \cdot \phi_k(\mathsf{m}) + \varepsilon'(\mathsf{sk})$, where the error term satisfies*

$$\|\varepsilon'(\mathsf{sk})\|_\infty \le \frac{1}{q_l} \left(n\nu B + \frac{k\left((n_\mathrm{g}+1)\|\mathsf{sk}\|_1 + 1\right) + 1}{2} \right) + B_\mathrm{rs}, \tag{15}$$

for some $\nu, B \ge 0$ such that $\|\!|\tilde{m}_i|\!\|_\infty \le \nu$ and $\|\!|\varepsilon|\!\|_\infty \le B$ for all $0 \le i < n$.

Proof. By Minkowski's inequality and Theorem 7, after step 1 we have $\tilde{\mathsf{ct}}_i(\mathsf{sk}) \equiv P \cdot \phi_i(\mathsf{m}) + \varepsilon_1 \pmod{Q}$ for some error $\varepsilon_1 \in \mathcal{R}$ such that $|||\varepsilon_1|||_\infty \leq P \cdot |||\varepsilon|||_\infty + C \leq P \cdot B + C$. Next, step 2 computes a linear combination of n_b of these ciphertexts with plaintext diagonals of matrix $\boldsymbol{M}$. Each of the resulting ciphertexts decrypts to

$$\tilde{\mathsf{ct}}'_j(\mathsf{sk}) = P \cdot \sum_{i=0}^{n_\mathrm{b}} \phi_{-n_\mathrm{b} j}(\tilde{m}_{n_\mathrm{b} j + i}) \cdot \phi_i(\mathsf{m}) + \varepsilon_2, \tag{16}$$

where the error term satisfies

$$\|\varepsilon_2\|_\infty \leq |||\varepsilon_2|||_\infty \leq \sum_{i=0}^{n_\mathrm{b}} |||\phi_{-n_\mathrm{b} j}(\tilde{m}_{n_\mathrm{b} j + i})|||_\infty |||\varepsilon_1|||_\infty \leq n_\mathrm{b} \nu |||\varepsilon_1|||_\infty$$

due to the fact that N is a power of 2 and all the necessary products are carried out in the evaluation domain. The approximate modulus lowering of $\tilde{a}_j$ from $\mathcal{R}_{\tilde{Q}}$ to $\mathcal{R}_Q$ in step 4 yields a polynomial $a_j = P^{-1} \cdot \tilde{a}_j + \varepsilon_4$ with $\|\varepsilon_4\|_\infty \leq k/2$. (See Appendix A of [11].) Finally, by Theorem 2, step 5 yields a ciphertext $\mathsf{ct}'(S)$ that opens to

$$\mathsf{ct}'(\mathsf{sk}) = \sum_{j=0}^{n_\mathrm{g}} \phi_{n_\mathrm{b} j}(a_j) \cdot \phi_{n_\mathrm{b} j}(\mathsf{sk}) + P^{-1} \sum_{j=0}^{n_\mathrm{g}} \phi_{n_\mathrm{b} j}(\tilde{b}_j) + \varepsilon_5,$$

with $\|\varepsilon_5\|_\infty \leq (1 + k + \|\mathsf{sk}\|_1 k)/2$. We can rewrite this expression as

$$\mathsf{ct}'(\mathsf{sk}) = P^{-1} \cdot \sum_{j=0}^{n_\mathrm{g}} \phi_{n_\mathrm{b} j}(\tilde{b}_j + \tilde{a}_j \cdot \mathsf{sk}) + \varepsilon'_5 = P^{-1} \cdot \sum_{j=0}^{n_\mathrm{g}} \phi_{n_\mathrm{b} j}(\tilde{\mathsf{ct}}'_j(\mathsf{sk})) + \varepsilon'_5,$$

in which $\varepsilon'_5 := \sum_{j=0}^{n_\mathrm{g}-1} \phi_{n_\mathrm{b} j}(\varepsilon_4 \cdot \mathsf{sk}) + \varepsilon_5$ satisfies $\|\varepsilon'_5\|_\infty \leq n_\mathrm{g} \|\varepsilon_4\|_\infty \|\mathsf{sk}\|_1 + \|\varepsilon_5\|_\infty$. Now apply (16) to obtain

$$\mathsf{ct}'(\mathsf{sk}) = P^{-1} \sum_{j=0}^{n_\mathrm{g}} \phi_{n_\mathrm{b} j} \left(P \sum_{i=0}^{n_\mathrm{g}} \phi_{-n_\mathrm{b} j}(\tilde{m}_{n_\mathrm{b} j + i}) \cdot \phi_i(\mathsf{m}) + \varepsilon_2 \right) + \varepsilon'_5.$$

Since the automorphisms distribute over addition, this decryption is equal to the sum of the desired noiseless result $\sum_{k=0}^{n-1} \tilde{m}_k \cdot \phi_k(\mathsf{m})$ plus an error term ε' such that

$$\|\varepsilon'\|_\infty \leq P^{-1} n_\mathrm{g} \|\varepsilon_2\|_\infty + \|\varepsilon'_5\|_\infty. \tag{17}$$

Substitute all the error bounds into (17), eliminate the term $P^{-1} \cdot C$ by using the assumption $P > C$, and add the factor q_l^{-1} and the noise term B_rs introduced by the final rescaling step to establish the stated inequality (15). □

It is reassuring to know that the dominant term in the upper bound (15) indeed comes from the n plaintext–ciphertext multiplications. The fact that the other part of the bound depends on n_g but not on n_b also deserves mention, because it suggests that taking n_b to be larger than n_g is advantageous not only from the standpoint of speed but also from that of noise growth.

6 Conclusion

We have constructed a general and flexible procedure that performs the key-switch of multiple polynomials in batch, while incorporating the gadget decomposition technique. We have applied this algorithm to two operations that require key-switching: multiplications and rotations.

For multiplication, we have proposed a ternary method with two-phase rescaling, which relinearizes the product of three ciphertexts through a single key-switch whose gadget products are computed one level lower than in the sequential approach. Our implementation outperforms two consecutive binary multiplications in every configuration tested, with speedups between 1.06× and 1.31× that are largest at low levels. We have then given a dynamic-programming algorithm that, for any number of ciphertexts at arbitrary levels, combines binary and ternary multiplications into a tree of minimum depth consumption and, subject to this, minimum latency. With an adequate cost function, the resulting trees speed up products of up to ten ciphertexts by up to 1.31×. For the 192-ary product at the end of SHIP bootstrapping [12], the optimal tree places 64 ternary nodes at its bottom layer and is between 1.14× and 1.20× faster than the binary tree, depending on the number of threads.

For rotations, we have used batched key-switching to accumulate automorphisms of a ciphertext. Based on its noise bounds, we have provided the first explicit error analysis of the double-hoisted BSGS matrix-vector multiplication algorithm of Bossuat *et al.* [9]. Our results provide noise-based guidance for selecting the baby- and giant-step parameters, complementing existing efficiency-based criteria in the literature.

Future work. A natural extension is to combine batched key-switching with the double-decomposition method of Kim *et al.* [27]. At the circuit level, bootstrap placement could be optimized jointly with multiplication trees, in the spirit of Orbit [41]. Already within our framework, selecting a target level below the maximum when the product is followed by bootstrapping allows Algorithm 5 to trade levels for cheaper ternary nodes. Batched key-switching and the tree search are also natural candidates for optimization passes in FHE compilers [33, 5]. Finally, the main cost of batching is memory: more polynomials must be kept over the extended modulus at once, and ternary multiplication requires a larger evaluation key. These trade-offs should be revisited on GPUs, dedicated accelerators [1, 25], and emerging memory architectures, where evaluation-key size and data movement weigh more heavily on latency. Our tree search already accommodates such platforms, provided that its cost function accounts for memory as well as computation.

References

1. Agrawal, R., Chandrakasan, A., Joshi, A.: HEAP: A fully homomorphic encryption accelerator with parallelized bootstrapping. In: ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA). pp. 756–769 (2024)

2. Akherati, S., Tang, Y.J., Zhang, X.: Three-input ciphertext multiplication for homomorphic encryption. In: IEEE International Symposium on Circuits and Systems (ISCAS). pp. 1–5 (2025)
3. Akherati, S., Zhang, X.: Multi-input ciphertext multiplication for homomorphic encryption. arXiv preprint arXiv:2601.15401 (2026)
4. Al Badawi, A., Bates, J., Bergamaschi, F., Cousins, D.B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., et al.: OpenFHE: Open-source fully homomorphic encryption library. In: 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (2022)
5. Ali, A., Choi, J., Gipson, B., Gorantala, S., Kun, J., Legiest, W., Lim, L., Viand, A., Demissie, M.Z., Zheng, H.: HEIR: A universal compiler for homomorphic encryption (2025), https://arxiv.org/abs/2508.11095
6. Badawi, A.A., Alexandru, A., Polyakov, Y., Vaikuntanathan, V.: SoK: Private LLM inference using approximate homomorphic encryption. Cryptology ePrint Archive, Paper 2026/935 (2026), https://eprint.iacr.org/2026/935
7. Bajard, J.C., Eynard, J., Hasan, M.A., Zucca, V.: A full RNS variant of FV like somewhat homomorphic encryption schemes. pp. 423–442 (2016). https://doi.org/10.1007/978-3-319-69453-5_23
8. Bellman, R.: Dynamic programming. Science **153**(3731), 34–37 (1966). https://doi.org/10.1126/science.153.3731.34, https://www.science.org/doi/abs/10.1126/science.153.3731.34
9. Bossuat, J.P., Mouchet, C., Troncoso-Pastoriza, J.R., Hubaux, J.P.: Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. pp. 587–617 (2021). https://doi.org/10.1007/978-3-030-77870-5_21
10. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: Bootstrapping for approximate homomorphic encryption. pp. 360–384 (2018). https://doi.org/10.1007/978-3-319-78381-9_14
11. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A full RNS variant of approximate homomorphic encryption. pp. 347–368 (2019). https://doi.org/10.1007/978-3-030-10970-7_16
12. Cheon, J.H., Hanrot, G., Kim, J., Stehlé, D.: SHIP: A shallow and highly parallelizable CKKS bootstrapping algorithm. In: Fehr, S., Fouque, P.A. (eds.) Advances in Cryptology – EUROCRYPT 2025, Part III. Lecture Notes in Computer Science, vol. 15603, pp. 398–428. Springer, Cham, Switzerland, Madrid, Spain (May 4–8, 2025). https://doi.org/10.1007/978-3-031-91131-6_14
13. Cheon, J.H., Kim, A., Kim, M., Song, Y.S.: Homomorphic encryption for arithmetic of approximate numbers. pp. 409–437 (2017). https://doi.org/10.1007/978-3-319-70694-8_15
14. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. pp. 643–662 (2012). https://doi.org/10.1007/978-3-642-32009-5_38
15. Dathathri, R., Kostova, B., Saarikivi, O., Dai, W., Laine, K., Musuvathi, M.: EVA: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation (2020)
16. Ebel, A., Garimella, K., Reagen, B.: Orion: A fully homomorphic encryption framework for deep learning. In: Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2. pp. 734–749 (2025). https://doi.org/10.1145/3676641.3716008, https://doi.org/10.1145/3676641.3716008

17. EPFL-LDS, Tune Insight SA: Lattigo v6 (Aug 2024), https://github.com/tuneinsight/lattigo
18. Genise, N., Micciancio, D., Polyakov, Y.: Building an efficient lattice gadget toolkit: Subgaussian sampling and more. pp. 655–684 (2019). https://doi.org/10.1007/978-3-030-17656-3_23
19. Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University (2009), crypto.stanford.edu/craig
20. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. pp. 850–867 (2012). https://doi.org/10.1007/978-3-642-32009-5_49
21. Goldschmidt, R.E.: Applications of division by convergence. In: Thesis (M.S.)–Massachusetts Institute of Technology, Dept. of Electrical Eng. (1964)
22. Halevi, S., Shoup, V.: Algorithms in HElib. In: Advances in Cryptology (CRYPTO). pp. 554–571 (2014)
23. Halevi, S., Shoup, V.: Faster homomorphic linear transformations in HElib. In: Advances in Cryptology (CRYPTO). pp. 93–120 (2018)
24. Han, K., Ki, D.: Better bootstrapping for approximate homomorphic encryption. pp. 364–390 (2020). https://doi.org/10.1007/978-3-030-40186-3_16
25. Jayashankar, S., Chen, E., Tang, T., Zheng, W., Skarlatos, D.: Cinnamon: A framework for scale-out encrypted AI. In: 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1. p. 133–150. ASPLOS (2025)
26. Karatsuba, A., Ofman, Y.: Multiplication of multidigit numbers on automata. Soviet Physics Doklady **7**, 595–596 (1963)
27. Kim, M., Lee, D., Seo, J., Song, Y.: Accelerating HE operations from key decomposition technique. In: Advances in Cryptology (CRYPTO). pp. 70–92 (2023)
28. Kraft, L.G.: A Device for Quantizing, Grouping, and Coding Amplitude-Modulated Pulses. Master's thesis, Massachusetts Institute of Technology (1949)
29. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. pp. 1–23 (2010). https://doi.org/10.1007/978-3-642-13190-5_1
30. McMillan, B.: Two inequalities implied by unique decipherability. IRE Transactions on Information Theory **2**(4), 115–116 (1956)
31. Moon, J., Yoo, D., Jiang, X., Kim, M.: THOR: Secure transformer inference with homomorphic encryption. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. pp. 3765–3779 (2025)
32. Ouyang, R., Liu, J., Gao, P., Xiao, P., Zhang, H., Au, M.H., Zhang, C., Ren, K., Chen, C.: CKKS bootstrapping in less than 10ms. Cryptology ePrint Archive, Paper 2026/2119 (2026)
33. Park, S., Song, W., Nam, S., Kim, H., Shin, J., Lee, J.: HEaaN.MLIR: An optimizing compiler for fast ring-based homomorphic encryption. Proc. ACM Program. Lang. **7**(PLDI) (2023). https://doi.org/10.1145/3591228, https://doi.org/10.1145/3591228
34. Pindado, Z., Spendlhofer, T., Allam, M., Mehta, P., Martens, L., Peña, A.J.: SoK of private deep neural network inference with approximate fully homomorphic encryption. Cryptology ePrint Archive, Paper 2026/047 (2026), https://eprint.iacr.org/2026/047
35. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. Journal of the ACM (JACM) **56**(6), 1–40 (2009)
36. Microsoft SEAL (release 4.3). https://github.com/Microsoft/SEAL (Apr 2026), microsoft Research, Redmond, WA.
37. Stehlé, D., Steinfeld, R., Tanaka, K., Xagawa, K.: Efficient public key encryption based on ideal lattices. In: Advances in Cryptology (ASIACRYPT). pp. 617–635 (2009)

38. Ypma, T.J.: Historical development of the Newton–Raphson method. SIAM Review **37**(4), 531–551 (1995)
39. Zama: Concrete: TFHE compiler that converts Python programs into FHE equivalent (2022), https://github.com/zama-ai/concrete
40. Zhang, J., Yang, X., He, L., Chen, K., Lu, W.j., Wang, Y., Hou, X., Liu, J., Ren, K., Yang, X.: Secure transformer inference made non-interactive. In: The Network and Distributed System Security (NDSS) Symposium (2025)
41. Zhou, Z., Seo, W., Chen, E., Ozdemir, A., Brown, F., Zheng, W.: Orbit: Optimizing rescale and bootstrap placement with integer linear programming techniques for secure inference. In: 35th USENIX Security Symposium (USENIX Security 26). pp. 61–80. USENIX Association, Baltimore, MD (Aug 2026), https://www.usenix.org/conference/usenixsecurity26/presentation/zhou