Faster Amortized FHEW bootstrapping using Ring Automorphisms*

Gabrielle De Micheli[1], Duhyeong Kim[2], Daniele Micciancio[1], and Adam Suhl[1]

[1] University of California, San Diego
[2] Intel

January 28, 2023

Abstract. Amortized bootstrapping offers a way to simultaneously refresh many ciphertexts of a fully homomorphic encryption scheme, at a total cost comparable to that of refreshing a single ciphertext. An amortization method for FHEW-style cryptosystems was first proposed by (Micciancio and Sorrell, ICALP 2018), who showed that the amortized cost of bootstrapping n FHEW-style ciphertexts can be reduced from $\tilde{O}(n)$ basic cryptographic operations to just $\tilde{O}(n^{\epsilon})$, for any constant $\epsilon > 0$. However, despite the promising asymptotic saving, the algorithm was rather inpractical due to a large constant (exponential in $1/\epsilon$) hidden in the asymptotic notation. In this work, we propose an alternative amortized boostrapping method with much smaller overhead, still achieving $O(n^{\epsilon})$ asymptotic amortized cost, but with a hidden constant that is only linear in $1/\epsilon$, and with reduced noise growth. This is achieved following the general strategy of (Micciancio and Sorrell), but replacing their use of the Nussbaumer transform, with a much more practical Number Theoretic Transform, with multiplication by twiddle factors implemented using ring automorphisms. A key technical ingredient to do this is a new "scheme switching" technique proposed in this paper which may be of independent interest.

1 Introduction

Fully Homomorphic Encryption (FHE) schemes support the evaluation of arbitrary programs on encrypted data. Since a first solution to the problem was proposed in [6], FHE has become both a central tool in the theory of cryptography, and an attractive cryptographic primitive to be used to secure privacy sensitive applications. Still, improving the efficiency of these schemes is a major obstacle to the use of FHE in practice, and a very active area of research.

All reasonably efficient currently known constructions of FHE are based on the "Ring Learning With Errors" (RingLWE) problem [12,15]. There are two main approaches to design FHE schemes based on Ring LWE: the one pioneered by the BGV cryptosystem and its variants (e.g., see [3,7,8]) and the one put forward by the FHEW cryptosystem and follow up work (e.g., see [2,4,5].) In BGV, ring operations are directly used to implement (componentwise) addition and multiplication of ciphertexts encrypting *vectors* of values. The ability to simultaneously work on all the components of a vector (at the cost of a single cryptographic operation) makes these schemes very powerful. The downside is that they also require fairly large parameters, leading to stronger security assumptions (namely, the hardness of approximating lattice problems within superpolynomial factors), a very slow bootstrapping procedure, and complex programming model. By constrast, in FHEW, ciphertexts are simple LWE encryptions (which offer native support only for homomorphic addition,) while Ring LWE is used only internally, to implement a special "functional bootstrapping" procedure that, given an encryption of m, produces a (bootstrapped) encryption of $f(m)$, for a given

* Research supported in part by the Swiss National Science Foundation Early Postdoc Mobility Fellowship, Intel Crypto Frontiers award, and NSF Award 1936703.

function f. The combination of linearly homomorphic LWE addition and functional bootstrapping still allows to perform arbitrary computations: for example, as originally done in [5], one can represent bits $x, y \in \{0, 1\}$ as integers modulo 4, and then implement a (universal) NAND boolean gate as an addition followed by a (functional bootstrapping) rounding operation $\lfloor (x + y + 2 \bmod 4)/2 \rfloor$. The FHEW approach has several attractive features: (1) since bootstrapping is performed after every operation, gates can be arbitrarily composed, leading to a very simple and easy to use programming model; (2) since we only need to bootstrap basic LWE ciphertexts supporting a single homomorphic addition, the scheme can be instantiated with much smaller parameters; (3) in turn, this leads to weaker security assumptions (hardness of approximating lattice problems within polynomial factors), and substantially simpler and faster bootstrapping, orders of magnitude faster than BGV.

However, the lower speeds of BGV bootstrapping are largely compensated by its ability to encrypt and operate on many values (encrypted as a vector) at the same time, allowing, for example, to simultaneously bootstrap thousands of ciphertexts. This drastically reduces the *amortized* cost of BGV bootstrapping, and making it still preferable to FHEW in terms of overall performance in many settings.

In an effort to bridge the gap between the two approaches, a method to amortize FHEW bootstrapping was proposed in [14]. The suggested method consists in combining several (say n) FHEW/LWE input ciphertexts into a single RingLWE ciphertext, and then perform FHEW-style bootstrapping on a single RingLWE ciphertext. This results in a major asymptotic performance improvement, reducing the amortized cost of FHEW bootstrapping from $O(n)$ homomorphic multiplications to just $O(n^\epsilon)$, for any fixed constant $\epsilon > 0$. Unfortunately, the method of [14] is rather far from being practical, due in large part to a large constant $2^{O(1/\epsilon)}$ hidden in the asymptotic notation.

Challenges, Results and Techniques In this paper we propose a variant of the bootstrapping procedure of [14] with similar asymptotics, but substantially smaller multiplicative overhead. In particular, we reduce the amortized cost of FHEW bootstrapping from $2^{O(1/\epsilon)} \cdot n^\epsilon$ to just $(1/\epsilon) \cdot n^\epsilon$. In other words, we achieve a similar asymptotic cost $O(n^\epsilon)$ (for any constant $\epsilon > 0$), but with an exponentially smaller constant hidden in the asymptotic notation.

The main challenge faced by [14] was the use of RingGSW registers to implement the homomorphic fourier transform required to bootstrap a RingLWE ciphertext. These registers, introduced in [5], encrypt messages in the exponent as X^m. This allows to implement homomorphic addition using some form of ciphertext multiplication $X^{m_0} \cdot X^{m_1} = X^{m_0+m_1}$, but other homomorphic operations required by FFT/NTT algorithms (like subtraction and constant multiplication by so-called "twiddle factors") are much harder, seemingly requiring homomorphic division and exponentiation on ciphertexts. This is addressed in [14] by using the Nussbaumer transform, a variant of the FFT/NTT algorithm that does not require multiplication by twiddle factors. Unfortunately, the use of the Nussbaumer transform in [14] also introduces a $2^{O(1/\epsilon)}$ factor in the running time, making the algorithm impractical.

Methods to perform homomorphic multiplication in the exponent (i.e., exponentiation by a constant) are known, using automorphisms, and have been used in connection to the bootstrapping of FHEW-like cryptosystems [2, 11], but they only work for RingLWE ciphertexts, making them inapplicable to the RingGSW ciphertexts required by [14]. In this paper we introduce three technical innovations that allow to overcome these issues:

– We introduce a new RingLWE-to-RingGSW "scheme switching" procedure, which allows us to transform RingLWE ciphertexts into equivalent RingGSW ones. The method seems of independent interest and may find applications elsewhere
– We design a new variant of the amortized FHEW bootstrapping of [14] that operates on RingLWE registers, rather than RingGSW. This allow us to implement multiplication by arbitrary twiddle factors using the automorphism techniques of [2, 11], and instantiate the amortized FHEW bootstrapping framework with a standard (homomorphic) FFT/NTT computation, which carries a much smaller overhead. Then, when RingGSW registers are required, we resort to our scheme switching procedure to convert RingLWE to RingGSW on the fly.
– We replace the power-of-two cyclotomic rings [5, 11, 14] and circulant rings [2] used by previous FHEW bootstrapping algorithms, with prime cyclotomics. This requires a new error analysis for prime cyclotomics, which we describe in this paper. (Error analysis for power-of-two cyclotomic and circulant rings is comparatively much easier.) This speeds up and simplifies various steps of our bootstrapping procedure, e.g., by supporting a standard radix 2 FFT (as opposed to the radix 3 Nussbaumer transform of [14]), and completely bypassing the problem that automorphisms only exists for invertible exponents [11].

One important problem that still remains open is that of reducing the noise growth in amortized FHEW bootstrapping. Just as in previous work [14], the ciphertext noise of our bootstrapping procedure increases multiplicatively at every level of the FFT/NTT computation. In order to keep the RingLWE noise (and underlying lattice inapproximability factors) polynomial, this requires to limit the recursive depth of the FFT/NTT algorithms to a constant. This is the reason why both [14] and our work only achieve $O(n^\epsilon)$ amortized complexity, rather than the $O(\log n)$ one would expect from a full ($O(\log n)$-depth) FFT algorithm. In practice, this limits the recursive depth to a small constant, typically just two levels or so. Further improving amortized FHEW boostrapping, allowing the execution of a homomorphic FFT with $O(\log n)$ levels is left as an open problem.

Related and Concurrent work: Ring automorphisms have been used in many other works aimed at improving the efficiency of lattice cryptography based on the RingLWE problem, most notably the evaluation of linear functions in HElib [10] and algebraic trace computations [1]. Our use of automorphisms is most closely related to [11], which recently used them to improve the performance of FHEW (sequential, non-amortized) bootstrapping. In a concurrent and independent work [9], an algorithm very similar to ours is presented. The algorithm achieves essentially the same results, improving the cost of amortized FHEW bootstrapping from $2^{O(1/\epsilon)} \cdot n^{1/\epsilon}$ to $(1/\epsilon) \cdot n^{1/\epsilon}$. The overall structure of the algorithm is very similar, using automorphisms to replace the Nussbaumer transform in [14] with a standard FFT. However, the algorithms differ in some technical details. For example, while [9] uses the circular rings [2], we use prime cyclotomics, which results in marginally smaller ciphertexts. Another difference is that while [9] extends the automorphism multiplication technique to work directly on RingGSW ciphertexts, we center our FFT algorithm aroung RingLWE registers (which are smaller than RingGSW by a factor 2) and convert them to RingGSW only when necessary using our new scheme switching technique.

2 Preliminaries

We start by recalling some fundamental notions and definitions that will be used in this work.

2.1 Cyclotomic rings and embeddings

Given a positive integer N, the N^{th} cyclotomic polynomial is defined as $\Phi_N(X) = \prod_{i \in \mathbb{Z}_N^*}(X - \omega_N^i)$ for $\omega_N = e^{2\pi i/N} \in \mathbb{C}$ the complex N^{th} principal root of unity. The N^{th} cyclotomic ring is defined as $\mathcal{R}_N = \mathbb{Z}[X]/\Phi_N(X)$. In this work, we will consider the dth cyclotomic ring modulo q, for d a power-of-2, defined as $\mathcal{R}_d = \mathbb{Z}_q[X]/\Phi_d(X) \simeq \mathbb{Z}_q^{\phi(d)}$. Each element of this ring corresponds to a polynomial $\mathbf{a} \in \mathcal{R}_d$ of degree less than $\phi(d)$ and with coefficients taken modulo q. There exist various ways of representing a ring element. One can first map the polynomial $\mathbf{a}(X) = \sum_{i \leq \phi(d)} a_i \cdot X^i$ to its vector of coefficients $(a_1, a_2 \cdots, a_{\phi(d)}) \in \mathbb{Z}_q^{\phi(d)}$. This is known as the coefficient embedding. The norm of any ring element then refers to the ℓ_2 norm of the corresponding vector in the coefficient embedding.

Another representation of a ring element is with its canonical embedding $\sigma : K \to \mathbb{C}^n$ which endows K, the d^{th} cyclotomic number field, with a geometry. Note that the ring of integers of K corresponds to the d^{th} cyclotomic ring $\mathbb{Z}[X]/\Phi_d(X)$. We know that K has exactly $\phi(d)$ ring homomorphisms, also called embeddings, $\sigma_i : K \to \mathbb{C}$. The canonical embedding is then defined as the map $\sigma(a) = (\sigma_i(a))_{i \in \mathbb{Z}_d^*}$ for $a \in K$. The norm usually considered when using the canonical embedding is the ℓ_∞ norm $||\sigma(a)||_\infty = \max_i |\sigma_i(a)|$. More generally, for any $a \in K$ and any $p \in [1, \infty]$, the ℓ_p norm is defined as $||a||_p = ||\sigma(p)||_p$. Since the σ_i are ring homomorphisms, we then have for any $a, b \in K$ the inequality $||a \cdot b||_p \leq ||a||_\infty \cdot ||b||_p$.

Working with prime cyclotomics, or more generally with non-power-of-two cyclotomics can be rather cumbersome, in particular, when considering the canonical embedding and not just the coefficient embedding. We know that any two embeddings are related to each other by a fixed linear transformation on $\mathbb{R}^d$. For power-of-2 cyclotomics, the transformation is even an isometry and thus the coefficient and canonical embeddings are equivalent up to a $\sqrt{d}$ factor. In this work, we will be working with both $\mathcal{R}_d$, the d^{th} cyclotomic ring modulo q for which we will use the notation $\mathcal{R}_{in}$ and the q^{th} cyclotomic modulo Q for a prime q and a positive integer $Q > 0$, which we will denote $\mathcal{R}_{reg}$. The latter is a prime cyclotomic ring where the two embeddings cannot be easily interchanged. This will in particular affect the error growth analysis as we later explain in Section 3.2.

2.2 Encryption schemes and operations

We recall definitions and notations for the standard LWE encryption scheme used in the bootstrapping algorithm. We also extend our description to the ring version of LWE and introduce two related schemes, GadgetRLWE and RGSW, both used in our algorithm.

LWE: Consider some positive integers n and q. Let $\mathbf{sk} \leftarrow \chi$ be a secret key sampled from a distribution χ and $m \in \mathbb{Z}$ a message. The LWE encryption of the message m under the secret key $\mathbf{sk}$ is given by

$$\mathrm{LWE}_{q,\mathbf{sk}}(m) = [\mathbf{a}^T, b] \in \mathbb{Z}_q^{1 \times (n+1)},$$

where $\mathbf{a} \leftarrow \mathbb{Z}_q^n$, $b = -\mathbf{a} \cdot \mathbf{sk} + e + m \in \mathbb{Z}_q$ and $e \leftarrow \chi'$ is the error, sampled from a distribution χ', and ciphertexts are represented as *row* vectors.

RLWE: The ring version of LWE considers the ring $\mathcal{R}_q$. Let $\mathbf{sk} \leftarrow \chi$ be a secret key sampled from a distribution χ and $\mathbf{m} \in \mathcal{R}_q$ a message. The RLWE encryption of the message $\mathbf{m}$ under the secret key $\mathbf{sk}$ is given by

$$\mathrm{RLWE}_{q,\mathbf{sk}}(\mathbf{m}) = [\mathbf{a}, \mathbf{b}] \in {\mathcal{R}_q}^{1 \times 2},$$

where $\mathbf{a} \leftarrow \mathcal{R}_q$ and $\mathbf{b} = -\mathbf{a} \cdot \mathbf{sk} + \mathbf{e} + \mathbf{m}$ and $e_i \leftarrow \chi'$ for each coefficent e_i of the error. When the context is clear, we do not specify the modulus q or the secret key $\mathbf{sk}$.

Gadget RLWE or RLWE' Consider a gadget vector $\mathbf{v} = (v_0, v_1, \cdots, v_{k-1})$. Gadget RLWE or equivalently refered to as RLWE' is expressed as a vector of RLWE ciphertexts of the form

$$\mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{m}) = (\mathrm{RLWE}_{\mathbf{sk}}(v_0 \cdot \mathbf{m}), \mathrm{RLWE}_{\mathbf{sk}}(v_1 \cdot \mathbf{m}), \cdots, \mathrm{RLWE}_{\mathbf{sk}}(v_{k-1} \cdot \mathbf{m})) \in R_q^{k \times 2}$$

i.e., matrices with k rows, each representing a basic RLWE ciphertext. We remark that RLWE ciphertext can be regarded as a special case of RLWE$'$ instantiated with a trivial gadget $\vec{v} = (1)$. So, anything we say about RLWE$'$ applies to RLWE as well.

RingGSW Given a message $\mathbf{m} \in \mathcal{R}_q$, we define

$$\mathrm{RGSW}_{\mathbf{sk}}(\mathbf{m}) = (\mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{sk} \cdot \mathbf{m}), \mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{m})) \in {\mathcal{R}_q}^{2k \times 2}.$$

We now summarize the operations that can be done with the different schemes presented above and focus in particular on the operations used in our algorithm. The main operation in our algorithm that serves as a building block for other operations is the scalar multiplication by arbitrary ring elements. In order to compute this multiplication, one uses RLWE' with gadget vector $\mathbf{v} = (v_0, v_1, \cdots, v_{k-1})$. The scalar multiplication is denoted as $\mathcal{R} \odot \mathrm{RLWE}'$ and corresponds to $\odot : \mathcal{R} \times \mathrm{RLWE}' \to \mathrm{RLWE}$ defined as

$$\begin{aligned} \mathbf{t} \odot \mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{m}) &:= \sum_{i=0}^{k-1} \mathbf{t}_i \cdot \mathrm{RLWE}_{\mathbf{sk}}(v_i \cdot \mathbf{m}) \\ &= \mathrm{RLWE}_{\mathbf{sk}}\left(\sum_{i=0}^{k-1} v_i \cdot \mathbf{t}_i \cdot \mathbf{m}\right) = \mathrm{RLWE}_{\mathbf{sk}}(\mathbf{t} \cdot \mathbf{m}) \end{aligned}$$

where $\sum_i v_i \mathbf{t}_i = \mathbf{t}$ is the gadget decomposion of $\mathbf{t}$ into "short" vectors $\mathbf{t}_i$, for an appropriate notion of "short" depending on the gadget $\mathbf{v}$. Each operation performed with ciphertexts increases the error. When performing many of these operations, as in our bootstrapping algorithm, it is crucial to keep track of the error growth. More details will be given in Section 3.2. For now, we simply state that each error e_i in $\mathrm{RLWE}_{\mathbf{sk}}(v_i \cdot \mathbf{m})$, after the scalar multiplication, becomes $\sum_{i=0}^{k-1} \mathbf{t}_i \cdot \mathbf{e}_i$.

The RLWE and RLWE$'$ schemes only support multiplication by constant values. In order to obtain multiplication by ciphertexts, we need to consider the RGSW scheme. Let us now consider the multiplication $\star : \mathrm{RLWE} \times \mathrm{RGSW} \to \mathrm{RLWE}$ defined as

$$\begin{aligned} \mathrm{RLWE}_{\mathbf{sk}}(\mathbf{m}_1) \star \mathrm{RGSW}_{\mathbf{sk}}(\mathbf{m}_2) &:= \mathbf{a} \odot \mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{s} \cdot \mathbf{m}_2) + \mathbf{b} \odot \mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{m}_2) \\ &= \mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{a} \cdot \mathbf{s} \cdot \mathbf{m}_2 + \mathbf{b} \cdot \mathbf{m}_2) \\ &= \mathrm{RLWE}_{\mathbf{sk}}(\mathbf{m}_1 \cdot \mathbf{m}_2 + \mathbf{e}_1 \cdot \mathbf{m}_2) \end{aligned}$$

for $\mathrm{RLWE}(\mathbf{m}_1) := (\mathbf{a}, \mathbf{b})$. The output of this multiplication is an RLWE ciphertext encrypting the message $\mathbf{m}_1 \cdot \mathbf{m}_2 + \mathbf{e} \cdot \mathbf{m}_2$. The error thus additively increases by $\mathbf{e}_1 \cdot \mathbf{m}_2$. If the error term $\mathbf{e}_1 \cdot \mathbf{m}_2$ is sufficiently small, then this approximately results in an RLWE encryption of the product of the two messages.

This multiplication can be extended to RLWE$'$ ciphertext multiplication $\star' : \text{RLWE}' \times \text{RGSW} \to \text{RLWE}'$ defined as

$$\begin{aligned}&\text{RLWE}'_{\mathbf{sk}}(\mathbf{m}_1) \star' \text{RGSW}_{\mathbf{sk}}(\mathbf{m}_2)\\ &:= (\text{RLWE}_{\mathbf{sk}}(v_0 \cdot \mathbf{m}_1) \star \text{RGSW}_{\mathbf{sk}}(\mathbf{m}_2), \cdots, \text{RLWE}_{\mathbf{sk}}(v_{k-1} \cdot \mathbf{m}_1) \star \text{RGSW}_{\mathbf{sk}}(\mathbf{m}_2))\\ &\approx \text{RLWE}'_{\mathbf{sk}}(\mathbf{m}_1 \cdot \mathbf{m}_2).\end{aligned}$$

Each component $\text{RLWE}_{\mathbf{sk}}(v_i \cdot \mathbf{m}_1) \star \text{RGSW}_{\mathbf{sk}}(\mathbf{m}_2)$ of the result has the same error growth as a $\star$ operation in the RLWE $\star$ RGSW case. In particular, it includes an error term $\mathbf{e}_{1,i} \cdot \mathbf{m}_2$ that requires the second message $\mathbf{m}_2$ to be small.

The $\star'$ operation corresponds to k times the $\star$ operations, and thus a total of $2k$ $\odot$ operations.

2.3 Using ring automorphisms

Similarly as in [11], we use ring automorphisms to perform scalar multiplication with registers. Recall that an automorphism is a bijective maps from the ring $\mathcal{R}$ to itself such that for a given $t \in \mathbb{Z}_q^*$, we have $\mathbf{a}(X) \mapsto \mathbf{a}(X^t)$.

Automorphism in RLWE *and* RLWE$'$*:* Consider the following RLWE ciphertext $(\mathbf{a}(X), \mathbf{b}(X))$ which encrypts a given message $\mathbf{m}(X)$ under a certain key $\mathbf{sk}$, *i.e.,* $(\mathbf{a}(X), \mathbf{b}(X)) = \text{RLWE}_{\mathbf{sk}}(\mathbf{m}(X))$. We also consider a switching key $\mathbf{ak}_t = \text{RLWE}'_{\mathbf{sk}}(\mathbf{sk}(X^t))$, which is used to map ciphertexts $[\mathbf{a}, \mathbf{b}]$ from key $\mathbf{sk}(X^t)$ to $\mathbf{sk}$. Given an automorphism $\psi_t : \mathcal{R} \to \mathcal{R}$ such that $\mathbf{a}(X) \mapsto \mathbf{a}(X^t)$, we recall the procedure `Evalauto` given in [11]:

1. apply ψ_t to each of the RLWE components. One obtains
$$(\mathbf{a}(X^t), \mathbf{b}(X^t)) = \text{RLWE}_{\mathbf{sk}(X^t)}(\mathbf{m}(X^t))$$
2. apply a key switching function
$$[\mathbf{a}, \mathbf{b}] \mapsto \mathbf{a} \odot \mathbf{ak}_t + [\mathbf{0}, \mathbf{b}]$$
to obtain a ciphertext $\text{RLWE}_{\mathbf{sk}(X)}(\mathbf{m}(X^t))$

The same application can be done on RLWE$'$ ciphertexts. The only difference comes during the second step where we require k key switching, one for each RLWE ciphertext. The only $\mathcal{R} \odot \text{RLWE}'$ operation comes from key switching. Hence, for automorphism on RLWE, we have a single $\mathcal{R} \odot \text{RLWE}'$ operation and when considering automorphisms on RLWE$'$ we have k $\mathcal{R} \odot \text{RLWE}'$ operations, where k is the length of the gadget.

2.4 Homomorphic operations on registers

Following the FHEW framework, we use cryptographic registers that encrypt a $\mathbb{Z}_q$ element "in the exponent". In other words, a register storing $m \in \mathbb{Z}_q$ is an encryption of $X^m \in \mathcal{R}_{reg}$. In our algorithm, the encryption scheme will sometimes be RLWE$'$ and sometimes RGSW. Some operations require one scheme or the other. In order to perform some of these operations, we will need to scheme-switch from RLWE$'$ to RGSW. We describe our scheme-switching technique in Section 3.1. In our bootstrapping algorithm, we will primarily use three operations on registers, *i.e.,* either RLWE$'$ or RGSW ciphertexts. We have already mentioned these operations and recall them now:

– $\star'$: RLWE$'$ $\times$ RGSW $\to$ RLWE$'$ multiplications: this operation allows to multiply two ciphertexts, which in the exponent acts like and addition.
– Scheme-switching: this operation converts an RLWE$'$ register into an RGSW register.
– Automorphisms: this operation allows us to multiply the exponent of a RLWE$'$ ciphertext by some (known) value, and correponds to multiplication by a constant.

Note that automorphisms can only operate on RLWE$'$ registers, not an RGSW ones. (This is because RGSW does not directly support the key switching operation required by the second step of the homomorphic automorphic application algorithm. See Section 3.1 for details.) On the other hand, multiplication requires one of the two registers to be in RGSW format. The scheme switching operation is used to combine the two operations, keeping all registers in RLWE$'$ form, and convert them to RGSW only when required for multiplication.

In order to analyse the performance and the correctness of our algorithm we will analyse these three operations in terms of number of $\mathcal{R} \odot$ RLWE$'$ operations needed to compute them and the related error growth (see Table 2).

2.5 Standard and primitive (inverse) FFT

We only mention in this paper some relevant facts about FFT algorithms that are useful for our algorithm. Note that when refering to FFT and related algorithms, we actually refer to the Number Theoretic Transform (NTT) algorithm.

An FFT algorithm can either evalute a polynomial at all N^{th} roots of unity for a given N or only the primitive ones. The former case is refered to as a standard/cyclic FFT whereas the latter case is called a primitive/cyclotomic FFT. In the case of a standard FFT, the inverse direction reconstructs from these evaluations a polynomial mod $X^N - 1$. When multiplying two polynomials $a(x)$ and $z(x)$ modulo a cyclotomic polynomial, using a standard FFT (and its inverse) requires a "final reduction" step to take polynomials modulo $(X^N - 1)$ to polynomials modulo $\Phi_N(X)$. This "final reduction" increases the multiplicative depth of the circuit and, in our case, prevents some useful optimizations (namely, using RLWE instead of RLWE$'$ in the last FFT layer as we will explain in Section 4.2). We can avoid the final reduction step by using a primitive/cyclotomic FFT, which we recall only evaluates the polynomials at the primitive N^{th} roots of unity ω^i for $i \in \mathbb{Z}_N^*$. The inverse FFT then reconstructs from these evaluations a polynomial modulo the N^{th} cyclotomic polynomial $\Phi_N(X)$. We note however that, unlike with a standard FFT, the forward and inverse directions are not interchangeable. We focus the rest of the discussion on the case of power-of-two cyclotomics (which is the case we will use in this paper) where $N = d = 2^{\log_2 N}$, and $\phi(d) = d/2$. For the forward direction, let $0 \le i < \phi(d) = d/2$, and ω be a primitive d^{th} root of unity. Then the Fourier coefficients $\hat{f} := (\hat{f}_0, \hat{f}_1, ..., \hat{f}_{d/2})$ of a polynomial $f(X) = \sum_{i=0}^{d/2-1} f_i \cdot X^i \pmod{X^{d/2}+1}$ are computed as

$$\hat{f}_i := \sum_{j=0}^{d/2-1} f_j \omega^{(2i+1)j},$$

and the inverse FFT of $\hat{f}$ can be computed as

$$\hat{\hat{f}}_\ell := \frac{2}{d} \cdot \sum_{i=0}^{d/2-1} \hat{f}_i \omega^{-(2i+1)\ell} = \frac{2}{d} \cdot \omega^{-\ell} \sum_{i=0}^{d/2-1} \hat{f}_i \omega^{-2i\ell}$$

for each $0 \leq \ell < d/2$ and output $\hat{\hat{f}}(X) = \sum_{i=0}^{d/2-1} \hat{\hat{f}}_i \cdot X^i$. It is easy to verify that these operations are inverses of each other, *i.e.,* $\hat{\hat{f}} = f$. The FFT also preserves both addition and multiplication, *i.e.,* $\widehat{\hat{f} + \hat{g}} = f + g$ and $\widehat{\hat{f} \circ \hat{g}} = f \cdot g$ where $\circ$ denotes the component-wise multiplication of input vectors. Moreover, one notices that the inverse operation can be computed as a standard/cyclic length-$\phi(d)$ FFT (using ω^{-2} as the $\phi(d)^{th}$ root of unity) followed by a multiplication by a power of ω.

In this work, we mainly focus on homomorphic computation of the inverse FFT (while the forward FFT is done in cleartext), so the constant multiplication by $2/d$ becomes a (minor) computational overhead. We can easily remove this overhead by moving the constant from the inverse FFT to the forward FFT. If we move the constant $2/d$, we get

$$\mathrm{FFT}(f)_i := \frac{2}{d} \cdot \hat{f}_i = \frac{2}{d} \cdot \sum_{j=0}^{d/2-1} f_j \omega^{(2i+1)j},$$

$$\mathrm{FFT}^{-1}(\mathrm{FFT}(f))_\ell := \frac{d}{2} \cdot \widehat{\mathrm{FFT}(f)}_\ell = \sum_{i=0}^{d/2-1} \mathrm{FFT}(f)_i \omega^{-(2i+1)\ell}.$$

In this case, FFT^{-1} is still the inverse of FFT and addition is preserved in the same manner. However, note that there is a slight difference in multiplication: $\mathrm{FFT}^{-1}(\mathrm{FFT}(f) \circ \hat{g}) = \mathrm{FFT}^{-1}(\mathrm{FFT}(f \cdot g)) = f \cdot g$.

In our algorithm, we will consider partial (primitive) FFT, denoted by PFT, where instead of reducing modulo $(X - \zeta)$ (*i.e.,* evaluating the polynomials at $X = \zeta$, we reduce modulo $(X^k - \zeta)$. In this reduction, an X^i term will not interact with an X^j term unless $i \equiv j \mod k$. Hence, an equivalent description of a partial FTT is doing k FFTs in parallel, each with $1/k$ as many terms. More precisely, one FFT will operate on the terms which are 0 modulo k, one FFT on just the terms that are 1 modulo k, and so on. This also applies to the inverse direction. In our algorithm, we will thus divide by $\phi(d)/k$ and not $\phi(d)$.

2.6 Summary of notations

We summarize the notations used throughout the paper in Table 1. For simplicity of exposition, in this paper we use a standard power-of-B gadget $(1, B, B^2, \ldots, B^{d_B-1})$. In practice, this can be replaced by a CRT gadget which typically supports more efficient implementation.

3 Novel techniques

In this section, we introduce some novel techniques related to scheme switching and error analaysis. We first introduce a new variant of scheme-switching. We then introduce an error analysis in the context of prime cyclotomics. Indeed, our algorithm will use a prime cyclotomic for the registers, whereas common FHE schemes use power-of-2 cyclotomic rings for which the error analysis differs, as we will explain.

3.1 RLWE′ to RGSW scheme switching

When an automorphism is applied to a ciphertext, it modifies both the encrypted message and the encryption key. (This applies to RLWE, RLWE' and RGSW ciphertexts alike.) Therefore, in

	Notation	Description
Modulus	q_{plain}	Ciphertext modulus for standard LWE.
	q	Prime ciphertext modulus for input RLWE ciphertext. Plaintext modulus for registers.
	Q	Ciphertext modulus used in RLWE$'$ / RGSW registers.
Rings	$\mathcal{R}_{in}$	dth cyclotomic ring (mod q), $\frac{\mathbb{Z}_q[x]}{\Phi_d(x)} \simeq \mathbb{Z}_q^{\phi(d)}$.
	d	power-of-2 degree of $\mathcal{R}_{in}$.
	$\mathcal{R}_{reg}$	qth prime cyclotomic ring mod Q used by the registers.
FFT	k	degree at which we stop the partial FFT.
	$\phi(d)$	the number of Plain-LWE ciphertexts that are packed into an RLWE ciphertext; the number of coefficients in an $\mathcal{R}_{in}$ element; the number of coefficients in the input polynomial of the FFT; the number of registers in any layer of the IFFT; the number of registers output by the IFFT.
	$N = \phi(d)/k$	the number of degree-$(k-1)$ polynomials output by the partial FFT.
	$\omega \in \mathbb{Z}_q$	a primitive (d/k)th root of unity in $\mathbb{Z}_q$ for use in the FFT.
	r_i	radix for FFT layer.
	ℓ	number of FFT layers.
Secret keys	$\mathbf{s}_p \in \mathbb{Z}_{q_{plain}}^{n_{plain}+1}$	Plain LWE secret key.
	$z \in \mathcal{R}_{in}$	RLWE secret key for the input (packed) RLWE ciphertext.
	$s \in \mathcal{R}_{reg}$	RGSW secret key used for registers.
Gadget decomposition	B	Base for the powers-of-B gadget used in registers.
	d_B	$\lceil \log_B(Q) \rceil$, the length of the PowersOfB gadget.
Error variance	$\sigma^2_{\odot}$	The (expected) factor by which an $\mathcal{R}_{reg} \odot$ RLWE$'$ operation scales up the error variance in a ciphertext.
	$\sigma^2_{\odot,RGSW}$	The resulting error variance of the $\odot$ operation on each RLWE$'$ component of RGSW$(\mathbf{m}_2)$.
	$\sigma^2_{\odot,eval\_key}$	The resulting error variance of the $\odot$ operation on the evaluation key with error variance $\sigma^2_{eval\_key}$.
	$\sigma^2_{\odot,aut\_key}$	The resulting error variance of the $\odot$ operation on the automorphism key RLWE$'_{\mathbf{sk}}(\psi(\mathbf{sk}))$.
	σ^2_{in}	The error variance of the input to an operation.

Table 1. Summary of notations used in the paper

order to use automorphisms to operate homomorphically on ciphertexts, one needs a method to switch back to the original key. For RLWE and RLWE' ciphertexts, this is provided by a standard key switching operation as described in the previous section. However, for RGSW encryption, this does not quite work. The reason is that a RGSW encryption can be interpreted as a pair of RLWE' ciphertexts encrypting $\mathbf{m}$ and $\mathbf{m} \cdot \mathbf{sk}$. The first component does not pose any problem, as it can be transformed using a standard RLWE' key switching operation. However, key switching cannot be directly applied to the second component, because it encrypts a *key-dependent* message $\mathbf{m} \cdot \mathbf{sk}$. So, RGSW key-switching would require not only to modify the encryption key, but also to change the message from $\mathbf{m} \cdot \mathbf{sk}$ to $\mathbf{m} \cdot \mathbf{sk'}$, where $\mathbf{sk'}$ is the new key. For this reason, key switching (and homomorphic automorphism evaluation), is directly applicable only to RLWE and RLWE' ciphertexts. On the other hand, RGSW ciphertexts are required to perform homomorphic multiplications when both multiplicands are encrypted. We address this problem by providing a method to convert RLWE' ciphertexts to RGSW ones, which we call *scheme switching*. Let us now describe how this is done.

Since $\mathrm{RGSW}_{\mathbf{sk}}(\mathbf{m}) = (\mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{sk} \cdot \mathbf{m}), \mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{m}))$ and we are given $\mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{m})$, we just need a way to compute $\mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{sk} \cdot \mathbf{m})$. To do so, we will use $\mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{sk^2})$ given as part of the evaluation key. We will operate in parallel on each of the $\mathrm{RLWE}_{\mathbf{sk}}(v_i \cdot \mathbf{m})$ ciphertexts that make up the $\mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{m})$ ciphertext, lifting each $\mathrm{RLWE}_{\mathbf{sk}}(v_i \cdot \mathbf{m})$ to $\mathrm{RLWE}_{\mathbf{sk}}(v_i \cdot \mathbf{sk} \cdot \mathbf{m})$. More precisely, for each $\mathrm{RLWE}_{\mathbf{sk}}(v_i \cdot \mathbf{m}) := (\mathbf{a}, \mathbf{b})$, we compute

$$\mathbf{a} \odot \mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{sk}^2) + (\mathbf{b}, 0).$$

By regarding $(\mathbf{b}, 0)$ as a noiseless RLWE encryption of $\mathbf{b} \cdot \mathbf{sk}$ under the secret key $\mathbf{sk}$, this computation gives $\mathrm{RLWE}_{\mathbf{sk}}(\mathbf{a} \cdot \mathbf{sk}^2 + \mathbf{b} \cdot \mathbf{sk}) = \mathrm{RLWE}_{\mathbf{sk}}((\mathbf{a} \cdot \mathbf{sk} + \mathbf{b}) \cdot \mathbf{sk}) = \mathrm{RLWE}_{\mathbf{sk}}((v_i \cdot \mathbf{m} + \mathbf{e}) \cdot \mathbf{sk})$. Hence we do get $\mathrm{RLWE}_{\mathbf{sk}}(v_i \cdot \mathbf{sk} \cdot \mathbf{m})$ as desired, but with an additional error $\mathbf{e} \cdot \mathbf{sk}$ scaled up by $\mathbf{sk}$ from the input RLWE' ciphertext error $\mathbf{e}$. We will choose the secret key $\mathbf{sk}$ with small norm (e.g., binary) so that this multiplicative error growth remains small. More details about the full error growth for this scheme switching will be given in Section 3.2.

When our scheme switching method is used in conjuction with key switching, it allows a small optimization. Say we are given a $\mathrm{RLWE}'_{\mathbf{sk'}}(\mathbf{m})$, and we want to turn it into a RGSW encryption under $\mathbf{sk}$. This can be done in two steps, by first performing key-switching to $\mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{m})$, and then using the scheme switching key $\mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{sk^2})$ to compute $\mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{m} \cdot \mathbf{sk})$. The optimization consists in using a modified scheme switching key $\mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{sk'} \cdot \mathbf{sk})$ to turn the input ciphertext (encrypted under $\mathbf{sk'}$) directly into $\mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{m} \cdot \mathbf{sk})$, performing key switching $\mathbf{sk'} \to \mathbf{sk}$ and homomorphic multiplication by $\mathbf{sk}$ at the same time. Notice that the running time is about the same as before because we still need another key switching $\mathbf{sk'} \to \mathbf{sk}$ to compute the other component of the output RGSW ciphertext. However, combining key switching and multiplication in a single operation allows to slightly reduce the noise growth. In order to give a more modular presentation, we will ignore this optimization in the description of our algorithm.

3.2 Error growth in prime cyclotomics

Analysing the error growth in bootstrapping algorithms is crucial for the correctness of the scheme as it allows to set the proper modulus sizes and show the implementation can be run with concrete parameters. It is a standard practice in lattice cryptography to estimate the error growth during homomoprhic operations under the heuristic assumption that the noise in ciphertexts behaves like independent gaussian (or subgaussian) random variables, with standard deviation that depends on the computation leading to the ciphertext. In order to fairly compare our algorithm to previous work, in this paper we use a similar technique and compute the total error estimation based on the error variance introduced by a single $\mathcal{R}_{reg} \odot \mathrm{RLWE}'$ operation. In previous works, where a power-of-2 cyclotomic is being used, this value is equal to $\frac{1}{12} d_B q B^2 \sigma^2_{\text{input}}$ where B is the base for the power-of-B gadget, d_B is the length of the gadget and σ^2_{input} is the error variance of the input RLWE' ciphertext considered. Because $\mathcal{R}_{reg}$ is a prime cyclotomic, the analysis of this variance differs in our case as we do not directly have a bound on the ℓ_∞ norm in the canonical embedding of a ring element for which we know a bound on each coefficient. We thus propose the following theorem.

Theorem 1. *For an odd prime q and a positive integer Q, let $\mathcal{R}_{reg}$ be the q^{th} cyclotomic ring modulo Q used for registers, and d_B be the length of the gadget decomposition. For an* RLWE' *ciphertext defined over $\mathcal{R}_{reg}$, the error variance of the result of a single $\mathcal{R}_{reg} \odot$* RLWE' *operation is bounded by*

$$\sigma^2_{\odot} \leq 2 d_B q \sigma^2_r \sigma^2_{input},$$

where σ^2_{input} is the error variance of the input RLWE' *ciphertext, and σ^2_r is the variance of the gadget decomposition of the input $\mathcal{R}_{reg}$ ring element.*

Proof. We model an element $r \in \mathcal{R}_{reg}$ as sampled uniformly at random — this is a reasonable model because in our algorithm the $\mathcal{R}_{reg}$ elements always either come from a ciphertext (and hence are uniform) or are simply an integer constant (leading to even smaller error growth). The gadget decomposition of $r \in \mathcal{R}_{reg}$, denoted $\mathbf{G}^{-1}(r)$, then consists of d_B ring elements $r_1, \ldots, r_{d_B}$ (which we model as independently distributed). Note that we will use in our algorithm a balanced base-B digit decomposition but we leave the decomposition unspecified here for sake of generality. We model the error vector $\vec{e}_{\mathrm{RLWE}'} = (e_1, \ldots, e_{d_B})$ where each component is the error of each RLWE ciphertext in the input RLWE$'$ ciphertext, as independent random variables with variance $\sigma^2_{\mathrm{input}}$. The output error can then be computed as an inner product

$$e_{\mathrm{output}} = \langle \mathbf{G}^{-1}(r), \vec{e}_{\mathrm{RLWE}'} \rangle = \sum_{i=1}^{d_B} r_i \cdot e_i.$$

We will start by considering a single multiplication $v_i := r_i \cdot e_i \pmod{\Phi_q(X)}$. Recall that r_i and e_i are both ring elements of $\mathcal{R}_{reg}$, *i.e.,* polynomials of degree $q-2$ (as $\mathcal{R}_{reg} = \mathbb{Z}_Q[X]/(\Phi_q(X))$ and $\Phi_q(X) = 1 + X + \cdots + X^{q-1}$). We want to compute the variance of each coefficient of

$$v_i = (r_{i,0} + r_{i,1}X + \cdots + r_{i,q-2}X^{q-2}) \cdot (e_{i,0} + e_{i,1}X + \cdots + e_{i,q-2}X^{q-2}) \pmod{\Phi_q(X)}.$$

For simplicity of notation in the formulas below, we will consider r_i and e_i to be polynomials of degree $q-1$ (instead of $q-2$) with leading coefficients 0, *i.e.,* the trivial terms $r_{i,q-1} = e_{i,q-1} := 0$. By computing $r_i \cdot e_i \pmod{X^q - 1}$ first and then taking the result modulo $\Phi_q(X)$, we can easily obtain the ℓ^{th} coefficient of v_i, which we denote $v_i^{(\ell)}$, for $0 \le \ell \le q-2$. First, note that the ℓ^{th} coefficient of $v_i' := r_i \cdot e_i \pmod{X^q - 1}$ is given by

$$v_i'^{(\ell)} = \sum_{j=0}^{q-1} r_{i,j} \cdot e_{i,\ell-j},$$

where the subscripts of e are defined modulo q, *i.e.,* $e_{i,\ell-j} := e_{i,q+\ell-j}$ if $\ell < j$. Then, since $X^{q-1} = -X^{q-2} - \cdots - 1 \bmod \Phi_q(X)$, the ℓ^{th} coefficient of v_i modulo $\Phi_q(X)$ is computed as

$$v_i^{(\ell)} = v_i'^{(\ell)} - v_i'^{(q-1)} = \sum_{j=0}^{q-1} r_{i,j} \cdot (e_{i,\ell-j} - e_{i,q-j-1}).$$

Let $X_{i,j}^{(\ell)} := r_{i,j} \cdot (e_{i,\ell-j} - e_{i,q-j-1})$ for $0 \le j \le q-1$ and hence $v_i^{(\ell)} = \sum_{j=0}^{q-1} X_{i,j}^{(\ell)}$. Since $r_{i,q-1} = 0$ is a constant value, we trivially have that $\mathsf{var}(X_{i,q-1}^{(\ell)}) = 0$. When $0 \le j \le q-2$, the variance of each $X_{i,j}^{(\ell)}$ equals to

$$\mathsf{var}(X_{i,j}^{(\ell)}) = \mathsf{var}(r_{i,j}) \cdot \mathsf{var}(e_{i,\ell-j} - e_{i,q-j-1}) = \begin{cases} \sigma_r^2 \sigma^2_{\mathrm{input}} & \text{if } j = 0 \text{ or } \ell + 1 \\ 2\sigma_r^2 \sigma^2_{\mathrm{input}} & \text{else} \end{cases}.$$

The first variance corresponds to the case where $\mathsf{var}(e_{i,\ell-j} - e_{i,q-j-1}) = \mathsf{var}(e_{i,\ell-j})$ as $e_{i,q-j-1} = 0$ when $j = 0$ or when $\mathsf{var}(e_{i,\ell-j} - e_{i,q-j-1}) = \mathsf{var}(e_{i,q-j-1})$ as $\mathsf{var}(e_{i,\ell-j}) = 0$ when $j = \ell + 1$. Since $\mathsf{var}\left(\sum_{j=0}^{q-1} X_{i,j}^{\ell}\right) = \sum_{j=0}^{q-1} \mathsf{var}(X_{i,j}^{(\ell)}) + 2\sum_{0 \le j < k < q} \mathsf{cov}(X_{i,j}^{(\ell)}, X_{i,k}^{(\ell)})$, it now suffices to compute the covariance of each pair. We will first consider the special case where $k = j + \ell + 1$ as it is the only case where common terms appear between $X_{i,j}^{(\ell)}$ and $X_{i,k}^{(\ell)}$. Indeed, we have that for $k = j + \ell + 1$,

$$X_{i,k}^{(\ell)} = r_{i,j+\ell+1} \cdot (e_{i,-j-1} - e_{i,q-j-\ell-2}),$$

where $e_{i,-j-1} = e_{i,q-j-1}$ also appears in $X_{i,j}^{(\ell)}$. However, due to the distributive property of covariance, it holds that

$$\mathsf{cov}(X_{i,j}^{(\ell)}, X_{i,k}^{(\ell)}) = -\mathsf{cov}(r_{i,j} \cdot e_{i,q-j-1}, r_{i,j+\ell+1} \cdot e_{i,q-j-1}) = 0^3.$$

In all other cases we trivially have $\mathsf{cov}(X_{i,j}^{\ell}, X_{i,k}^{(\ell)}) = 0$ since $X_{i,j}^{\ell}$ and $X_{i,k}^{\ell}$ are independent. Note that there exist two j indices ($j = 0, \ell+1$) satisfying $\mathsf{var}(X_{i,j}^{(\ell)}) = \sigma_r^2 \sigma_{\text{input}}^2$ when $0 \le \ell < q-2$, while there exists only one such j index ($j = 0$) when $\ell = q-2$. As a result, we obtain the variance of $v_i^{(\ell)}$ as

$$\mathsf{var}(v_i^{(\ell)}) = \sum_{j=0}^{q-1} \mathsf{var}(X_{i,j}^{(\ell)}) = \begin{cases} (2q-4)\sigma_r^2 \sigma_{\text{input}}^2 & \text{if } 0 \le \ell < q-2 \\ (2q-3)\sigma_r^2 \sigma_{\text{input}}^2 & \text{if } \ell = q-2 \end{cases}.$$

Finally, the variance of each coefficient of e_{output} denoted by $\sigma_{\odot}^2$ is bounded by $2d_B q \sigma_r^2 \sigma_{\text{input}}^2$. □

Corollary 1. *For an odd prime q and a positivie integer Q, let $\mathcal{R}_{reg}$ be the q^{th} cyclotomic ring modulo Q used for registers, and d_B be the length of a balanced base-B gadget decomposition with uniform coefficients in $[-B/2, B/2)$. For an* RLWE′ *ciphertext defined over $\mathcal{R}_{reg}$, the error variance of the result of a single $\mathcal{R}_{reg} \odot$ RLWE′ operation is bounded by*

$$\sigma_{\odot}^2 \le \frac{B^2}{6} d_B q \sigma_{input}^2.,$$

where σ_{input}^2 is the error variance of the input RLWE′ *ciphertext.*

Proof. If one considers a balanced base-B digit decomposition of $r \in \mathcal{R}_{reg}$ which consists of d_B ring elements $r_1, \ldots, r_{d_B}$ whose coefficients are each uniform in $[-B/2, B/2)$, then the variance of the gadget decomposition of the input $\mathcal{R}_{reg}$ ring element satisfies $\sigma_r^2 = B^2/12$. By replacing this value in the upper bound for $\sigma_{\odot}^2$ given in Theorem 1, we get

$$\sigma_{\odot}^2 \le \frac{B^2}{6} d_B q \sigma_{input}^2.$$

□

Remark 1. Note that the variance $\sigma_{\odot}^2$ considered for error analysis in power-of-2 cyclotomic is $\sigma_{\odot}^2 = \frac{B^2}{12} d_B N \sigma_{input}^2$, (see [11, Section 4.2]), where N is a power of two and the $2N^{th}$ cyclotomic ring is considered. Interestingly, our analysis for prime cyclotomic rings only shows a difference by a factor 2.

Error growth in previous operations We now describe the error growth for the main operations used in our algorithm as a function of $\sigma_{\odot}^2$.

[3] In general, it holds that $\mathsf{cov}(XY, XZ) = E(X^2)E(Y)E(Z) - E(X)^2 E(Y)E(Z)$ for any random variables X, Y and Z. Therefore, if $E(Y) = E(Z) = 0$, then $\mathsf{cov}(XY, XZ) = 0$.

RGSW $\times$ RLWE$'$ *multiplication:* Recall that a multiplication between $\mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{m}_1)$ and $\mathrm{RGSW}_{\mathbf{sk}}(\mathbf{m}_2) = (\mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{m}_2), \mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{sk} \cdot \mathbf{m}_2))$ is computed as $\mathbf{a} \odot \mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{sk} \cdot \mathbf{m}_2) + \mathbf{b} \odot \mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{m}_2)$ for each RLWE component $(\mathbf{a}, \mathbf{b})$ of $\mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{m}_1)$. From this description, we easily see that two $\odot$ computations are performed to which should be added the error coming from the RGSW ciphertext itself multiplicatively. Finally, as already mentioned when describing the operation, the error also additively increases by $\mathbf{e}_{\mathrm{RLWE}'} \cdot \mathbf{m}_2$. Since $\mathbf{m}_2$ is a monomial, we simply add $\sigma^2_{\mathrm{RLWE}'}$. Therefore, the total error variance is equal to $2\sigma^2_{\odot,RGSW} + \sigma^2_{RLWE'}$ where $\sigma^2_{\odot,RGSW}$ denotes the resulting error variance of the $\odot$ operation on each RLWE$'$ component of $\mathrm{RGSW}(\mathbf{m}_2)$.

RLWE'-to-RGSW Scheme Switching: Recall that the operation can be described as $(\mathbf{b}, 0) + \mathbf{a} \odot \mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{sk}^2)$ for each RLWE component $(\mathbf{a}, \mathbf{b})$ of the input RLWE$'$ ciphertext. There are two sources of error. Firstly, an additive error growth comes from the $\odot$ operation in $\mathbf{a} \odot \mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{sk}^2)$. Since $\mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{sk}^2)$ is a fresh encryption that comes from the evaluation key, the error variance is relatively small. We thus have an additive error growth with variance $\sigma^2_{\odot,eval\_key}$ which denotes the resulting error variance of the $\odot$ operation on the evaluation key with error variance $\sigma^2_{eval\_key}$.

Secondly, a multiplicative error growth comes from the fact that the existing error in the RLWE$'$ ciphertext gets scaled by $\mathbf{sk}$. The secret key $\mathbf{sk}$ is not a scalar but rather a ring element and recall we work in a prime cyclotomic. We know the error variance scales by a factor of no more than $\ell_1(\mathbf{sk})$, where the norm is with respect to the canonical embedding for $\mathbf{sk}$.

Combining these two sources of error under the assumption that each coefficient of $\mathbf{sk}$ is binary/ternary, then the error variance of the output is $\ell_1(\mathbf{sk}) \cdot \sigma^2_{RLWE'} + \sigma^2_{\odot,eval\_key}$.

RLWE' Automorphism: Applying an automorphism ψ itself does not change the error. The following key-switching operation however introduces an additive error growth with variance $\sigma^2_{\odot,aut\_key}$ which denotes the resulting error variance of the $\odot$ operation on the automorphism key $\mathrm{RLWE}'_{\mathbf{sk}}(\psi(\mathbf{sk}))$.

We summarize these error growth in Table 2.

Operation	Computation (for each RLWE $(\mathbf{a}, \mathbf{b})$ of RLWE$'$)	$\#\mathcal{R} \odot$ RLWE$'$	Error Variance
RLWE$'$ $\times$ RGSW	$\mathbf{a} \odot \mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{s} \cdot \mathbf{m}_2) + \mathbf{b} \odot \mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{m}_2)$	$2k$	$2\sigma^2_{\odot,RGSW} + \sigma^2_{RLWE'}$
SchemeSwitch	$\mathbf{a} \odot \mathrm{RLWE}'_{\mathbf{sk}}(\mathbf{sk}^2) + (\mathbf{b}, 0)$	k	$\ell_1(\mathbf{sk}) \cdot \sigma^2_{RLWE'} + \sigma^2_{\odot,eval\_key}$
Automorphism	$\psi(\mathbf{a}) \odot \mathrm{RLWE}'_{\mathbf{sk}}(\psi(\mathbf{sk})) + (0, \psi(\mathbf{b}))$	k	$\sigma^2_{RLWE'} + \sigma^2_{\odot,aut\_key}$

Table 2. Summary of register operations with $\odot$ operation count and error growth.

4 Description of the algorithm

The overall algorithm, at a high level, can be subdivided into various steps:

- Step 1: a *packing* step takes as input $\phi(d)$ LWE ciphertexts and "combines" them into a single RLWE ciphertext $(\mathbf{a}, \mathbf{b}) \in \mathcal{R}_{in} \times \mathcal{R}_{in}$.

– Step 2: a *homomorphic decryption* of the RLWE ciphertext consists in computing (an encryption of) the ring element $(\mathbf{a} \cdot \mathbf{z} + \mathbf{b}) \in \mathcal{R}_{in}$, homomorphically, given (as a bootstrapping key) a encryption of $\mathbf{z}$.
– Step 3: an *msbExtract* step recovers the $\phi(d)$ LWE ciphertexts with reduced noise.

Step 1 and Step 3, except for the use of different rings, are very similar to previous work [14]. We describe the 3 steps in detail with a particular emphasis on Step 2 which is the main novelty of this paper.

4.1 Packing

The very first step of bootstrapping procedure consists in taking a set of LWE ciphertexts and pack them into a single RLWE ciphertext. More precisely, the packing algorithm takes as input $\phi(d)$ LWE ciphertexts encrypting messages $m_i \in \mathbb{Z}$ as well as an RLWE$'$ encryption RLWE$'(s_i)$ of each coefficients of the plain LWE secret key $\mathbf{s}_p = (s_0, s_1, \cdots, s_{n_{plain}}) \in \mathbb{Z}_{q_{plain}}^{n_{plain}+1}$, in the dth cyclotomic ring with modulus q_{plain} and outputs $(\mathbf{a}, \mathbf{b}) \in \mathcal{R}_{in} \times \mathcal{R}_{in}$ encrypting the message $\mathbf{m}(X) = \sum_i m_i X^{i-1}$. The pseudo-code is given in Algorithm 1.

Algorithm 1 Ring packing

Input: $\phi(d)$ plain LWE ciphertexts $(\vec{a}_i, b_i) \in \mathbb{Z}_{q_{plain}}^{n_{plain}} \times \mathbb{Z}_{q_{plain}}$, RLWE$'(s_i)$
Output: RLWE ciphertext in $\mathcal{R}_{in}$.
for $0 \leq i < n_{plain}$ **do**
  let $r_i = a_{0,i} + a_{1,i}X + a_{2,i}X^2 + \cdots + a_{\phi(d)-1,i}X^{\phi(d)-1}$ in $(\mathcal{R}_{in})_{q_{plain}}$.
end for
$r' = (0, (b_0 + b_1 X + b_2 X^2 + \cdots + b_{\phi(d)-1}X^{\phi(d)-1}))$ ▷ (Noiseless RLWE$'$ ciphertext)
$ct \leftarrow r' + \sum_{i=0}^{n_{plain}-1} r_i \odot \text{RLWE}'(s_i)$
return ModSwitch$_{q_{plain} \to q}(ct)$

For simplicity, we first built a ring ciphertext ct modulo q_{plain} (i.e., the original input modulus) and then switch the modulus to q. Alternatively, one can directly compute a ring ciphertext modulo q by using a packing key $\{\text{RLWE}'(s_i)\}_i$ already encrypted under modulus q. The packing key may also use a different gadget (e.g., the power-of-two gadget, instead of powers-of-B) than other ciphertexts used later in the algorithm.

Since this part of the algorithm is essentially identical to previous work [14], we omit these details, and move on to the second step.

4.2 Linear step

This step of the algorithm takes as input a single RLWE ciphertext $(\mathbf{a}, \mathbf{b}) \in {\mathcal{R}_{in}}^2$ and outputs $\phi(d)$ RLWE ciphertexts, each encrypting a coefficient of $(\mathbf{a} \cdot \mathbf{z} + \mathbf{b}) \in \mathcal{R}_{in}$ (recall that an element of $\mathcal{R}_{in}$ is a polynomial of degree $\phi(d)$). It can be further subdivided into two computations: a (homomorphic) polynomial multiplication between $\mathbf{a}$ and (an encryption of) $\mathbf{z}$, where each coefficient of the polynomials (describing the key $\mathbf{z}$, all intermediate results, and the final ring element)is a distinct ciphertext, and the addition of the ring element $\mathbf{b}$. We now provide a detailed explanation of these computations along with a pseudo-code of the various steps of the algorithm.

An FFT-based polynomial multiplication. For this step, the algorithm uses a standard FFT-based method summarized in Figure 4.2. More precisely, we perform the following steps:

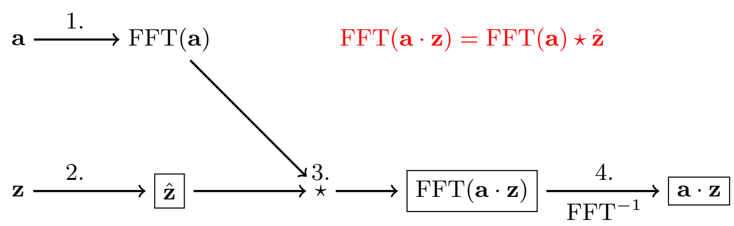

Fig. 1. High level description of the linear step of our algorithm. The notation $\star$ refers to pointwise multiplication. The boxed information refers to encrypted data where homomorphic operations are required. Each step $i.$ is described in more details in the paper.

1. Compute a partial FFT of $\mathbf{a} \in \mathcal{R}_{in}$, *i.e.*, PFT($\mathbf{a}$) in cleartext form. Let $k-1$ be the degree of the polynomials outputted by PFT. Note that a full (non-partial) FFT would have $k=1$ as the algorithm recurses until the input polynomial is reduced modulo all $\phi(d)$ linear factors of $\Phi_d(X)$. When computing PFT, the algorithm outputs $\phi(d)/k$ polynomials of degree $k-1$ (and hence does not recurse all the way down to the linear factors). In other words, this corresponds to evaluating the CRT isomorphism
$$\frac{\mathbb{Z}_q[X]}{\left(X^{d/2}+1\right)} \simeq \left(\frac{\mathbb{Z}_q[X]}{(X^k-\zeta_0)}\right) \times \cdots \times \left(\frac{\mathbb{Z}_q[X]}{\left(X^k-\zeta_{\phi(d)/k-1}\right)}\right)$$
where the ζ_i are the solutions to $(\zeta^k)^{\phi(d)} = -1$, namely the primitive $(d/k)-$th roots of unity modulo q. This step thus outputs a list of $\phi(d)/k$ polynomials $\{\tilde{\mathbf{a}}_i\}_{0\le i<\phi(d)/k}$, where each $\tilde{\mathbf{a}}_i = \mathbf{a} \mod (X^k-\zeta_i)$ is a polynomial with k coefficients. Note that this computation is done in the clear, and thus no homomorphic operations are needed.
Recall that when computing an inverse PFT, one must divide the polynomials $\tilde{\mathbf{a}}_i$ by $\phi(d)/k \pmod q$. In order to be able to compute this division in the clear rather than homomorphically, this step can be done now (Refer to Section 2.5). Hence the polynomials are updated to $\tilde{\mathbf{a}}_i \leftarrow \tilde{\mathbf{a}}_i/(\phi(d)/k) \pmod q$.
2. The evaluation key contains RGSW registers of PFT($\mathbf{z}$). Similarly as before, let $\tilde{\mathbf{z}}_i = \mathbf{z} \mod (X^k - \zeta_i)$, where each $\tilde{\mathbf{z}}_i$ is a polynomial with k coefficients. Let $\tilde{z}_i^{(j)}$ be the j^{th} coefficient of $\tilde{\mathbf{z}}_i$. Then the evaluation key contains the list of RGSW $\left(X^{\tilde{z}_i^{(j)}}\right)$ for $0 \le i < \phi(d)/k$ and $0 \le j < k$.
3. We now want to homomorphically compute PFT($\mathbf{a}\cdot\mathbf{z}$) from PFT($\mathbf{a}$) and the RGSW registers of PFT($\mathbf{z}$). Note that the polynomial multiplication in $\mathcal{R}_{in}$ corresponds to component-wise multiplication in PFT representation, *i.e.*, $\mathrm{PFT}(\mathbf{a}\cdot\mathbf{z}) = (\tilde{\mathbf{a}}_0\cdot\tilde{\mathbf{z}}_0, \tilde{\mathbf{a}}_1\cdot\tilde{\mathbf{z}}_1, ..., \tilde{\mathbf{a}}_{\phi(d)/k-1}\cdot\tilde{\mathbf{z}}_{\phi(d)/k-1})$. For ease of notation, let us fix i (we drop the subscript i) and consider a single multiplication of $\tilde{\mathbf{a}} := \sum_{j=0}^{k-1}\tilde{a}_jX^j$ and $\tilde{\mathbf{z}} := \sum_{j=0}^{k-1}\tilde{z}_jX^j$ modulo $(X^k-\zeta)$. More precisely, we want to homomorphically compute
$$(\tilde{a}_0 + \tilde{a}_1X + \cdots + \tilde{a}_{k-1}X^{k-1})(\tilde{z}_0 + \tilde{z}_1X + \cdots + \tilde{z}_{k-1}X^{k-1}) \mod (X^k-\zeta).$$

where each coefficient $\tilde{z}_j$ is encrypted as an RGSW register.
Each coefficient of the resulting product can be computed as follows. For $j = 0, \cdots, k-1$, the j-th coefficient of $\mathbf{v} := \tilde{\mathbf{a}} \cdot \tilde{\mathbf{z}}$ is equal to

$$v_j = \tilde{z}_0\tilde{a}_j + \tilde{z}_1\tilde{a}_{j-1} + \cdots + \tilde{z}_{j-1}\tilde{a}_1 + \tilde{z}_j\tilde{a}_0 + \zeta\left(\tilde{z}_{j+1}\tilde{a}_{k-1} + \tilde{z}_{j+2}\tilde{a}_{k-2} + \cdots + \tilde{z}_{k-1}\tilde{a}_{j+1}\right),$$

which corresponds to the inner product taken between the vector of coefficients $\vec{\tilde{\mathbf{z}}} = (\tilde{z}_0, \cdots, \tilde{z}_{k-1})$ of the polynomial $\tilde{\mathbf{z}}$ and the new vector $\vec{\mathbf{c}} = (\tilde{a}_j, \tilde{a}_{j-1}, \ldots, \tilde{a}_0, \zeta\tilde{a}_{k-1}, \ldots, \zeta\tilde{a}_{j+1})$. We emphasize again the fact that the coefficients of $\vec{\mathbf{c}}$ are in the clear, whereas the coefficients of $\vec{\tilde{\mathbf{z}}}$ are not. So, it is easy to multiply $\vec{\mathbf{c}}$ by ζ.
Without loss of generality, let us assume all the coefficents c_i of $\vec{\mathbf{c}}$ are nonzero and thus invertible. (Here we use the fact that q is a prime. So, all nonzero elements are invertible modulo q and multiplication (in the exponent) can be implemented using an automorphism of the prime cyclotomic ring.) Then we can compute the inner product in a telescoping manner as

$$v_j = \left(\left(\ldots\left(\left(\tilde{z}_0 c_0 c_1^{-1} + \tilde{z}_1\right) c_1 c_2^{-1} + \tilde{z}_2\right) c_2 c_3^{-1} + \ldots\right) c_{k-2} c_{k-1}^{-1} + \tilde{z}_{k-1}\right) c_{k-1}.$$

This will end up being the most efficient way to compute this inner product homomorphically. Let us now explicit how one coefficient corresponding to a monomial X^j can be computed homomorphically (this computation will have to be repeated for all k coefficients of a single product as well as for all $\phi(d)/k$ pairs of $(\tilde{\mathbf{a}}_i, \tilde{\mathbf{z}}_i)$ polynomials).

(a) Let *accum* be an RLWE′ register, initialized as $\mathrm{RLWE}'\left(X^{\tilde{z}_0}\right)$ from the evaluation key.
(b) For $j' \in [0, \ldots, k-2]$, update *accum* as follows:
   i. Apply the automorphism that sends X to $X^{c'_j c^{-1}_{j'+1}}$.
   ii. Do an $\mathrm{RLWE}' \times \mathrm{RGSW}$ multiplication with $\mathrm{RGSW}\left(X^{\tilde{z}_{j'+1}}\right)$ from the evaluation key.
(c) Finally apply the automorphism that sends X to $X^{c_{k-1}}$, then the output is $\mathrm{RLWE}'\left(X^{v_j}\right)$.

Since we repeat (a)-(c) for every coefficient of $\tilde{\mathbf{a}}_i \cdot \tilde{\mathbf{z}}_i$ for $1 \le i \le \phi(d)/k$, the output of this step consists of $\phi(d)$ RLWE′ registers of the form

$$\left\{\mathrm{RLWE}'\left(X^{\mathbf{v}_i^{(j)}}\right)\right\}_{0 \le i < \phi(d)/k, 0 \le j < k},$$

where $\mathbf{v}_i^{(j)}$ denotes the j-th coefficient of $\mathbf{v}_i := \tilde{\mathbf{a}}_i \cdot \tilde{\mathbf{z}}_i \pmod{X^k - \zeta_i}$. This procedure is illustrated in Figure 2, and the corresponding pseudo-code for component-wise multiplication is given in Algorithm 3.

4. We now have the encryption of $\mathrm{PFT}(\mathbf{a} \cdot \mathbf{z})$. It thus remains to perform the inverse of PFT, denoted by PFT^{-1}, in order to recover the resulting polynomial product $\mathbf{a} \cdot \mathbf{z}$, more specifically RLWE encryptions of the coefficients of $\mathbf{a} \cdot \mathbf{z}$.
   Recall from Section 2.5 that the inverse of a primitive FFT of length N (using a $2N$th root of unity ω) can be computed by first taking a standard FFT of length N using ω^{-2} as the Nth root of unity, then multiplying the ith term by ω^{-i}. Moreover, a partial FFT of length $\phi(d)$ that reduces modulo $(X^k - \zeta)$ is equivalent to k full FFTs of length $N = \phi(d)/k$ done in parallel, and the same remains true for the inverse (see section 2.5 for details about the equivalence). Hence, to homomorphically compute PFT^{-1}, we will
   (a) Split the $\phi(d)$ registers output by the pointwise multiplication step into k groups of size N: each group corresponds to the coefficients of the monomial X^j for $0 \le j \le k-1$ of all $\phi(d)/k$ polynomials.

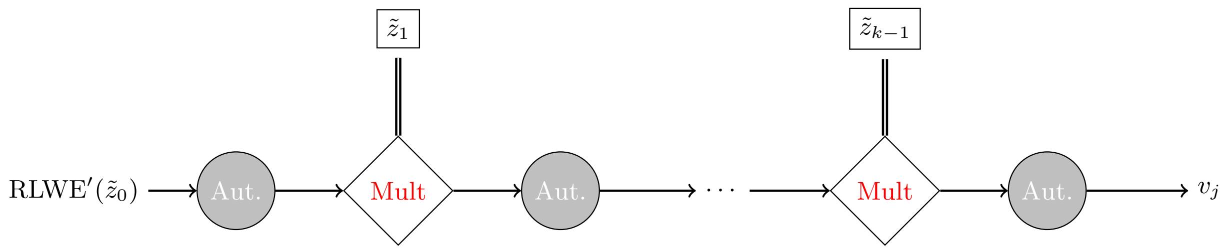

Fig. 2. Homomorphic computation of a X^j coefficient for pointwise multiplication. A single line corresponds to RLWE' ciphertexts and a double line to RGSW ciphertexts. Aut. stands for automorphisms and Mult. for multiplication. Boxed values are encrypted values.

Algorithm 2 Pointwise multiplication between polynomials $\tilde{\mathbf{a}}$ and $\tilde{\mathbf{z}}$.

Input: A set of degree-$(k-1)$ polynomials $\{\tilde{\mathbf{a}}_i\}_{0\le i<N}$, $\{\mathrm{RGSW}(X^{\tilde{z}_i^{(j)}})\}_{0\le i<N, 0\le j<k}$ for $N := \phi(d)/k$, ω: the $2N$-th root of unity mod q
Output: $\phi(d)$ RLWE′ ciphertexts
REG $\leftarrow [0,\ldots,0]$
for all $0 \le i < N$ **do**
    $\zeta \leftarrow \omega^{2i+1}$
    Let $\vec{\tilde{\mathbf{a}}} = (\tilde{a}_{i,0}, \tilde{a}_{i,1}, ..., \tilde{a}_{i,k-1})$ ▷ $\tilde{\mathbf{a}}_i := \sum_{j=0}^{k-1} \tilde{a}_{i,j} X^j$
    for all $0 \le j < k$ **do**
        $\vec{\mathbf{c}} \leftarrow (\tilde{a}_{i,j}, \tilde{a}_{i,j-1}, \ldots, \tilde{a}_{i,0}, \zeta\tilde{a}_{i,k-1}, \zeta\tilde{a}_{i,k-2}, \ldots \zeta\tilde{a}_{i,j+1})$
        accum $\leftarrow$ RLWE′$(X^{\tilde{z}_i^{(0)}})$
        for $j' \leftarrow 0, 1, \ldots, k-2$ **do**
            accum $\leftarrow$ EvalAut(*accum*, $c_{j'} c_{j'+1}^{-1}$)
            accum $\leftarrow$ MulRGSW(RGSW($\tilde{z}_i^{(j'+1)}$), *accum*)
        end for
        accum $\leftarrow$ EvalAut(*accum*, c_{k-1}) ▷ *accum* = RGSW$(X^{(\tilde{\mathbf{a}}_i \cdot \tilde{\mathbf{z}}_i)^{(j)}})$
        REG$[ik+j] \leftarrow$ *accum*
    end for
end for
output REG ▷ Register of all coefficients of $\tilde{\mathbf{a}}_i \cdot \tilde{\mathbf{z}}_i$ for $0 \le i < N$

(b) Homomorphically perform a standard (not primitive) length-N FFT in the forward direction on each group of size N, using ω^{-2} as the Nth root of unity. Overall, this step corresponds to computing k FFTs. We refer the reader to Section 2.5 for more details on the equivalence between partial FFT modulo $X^k - \zeta$ and k-parallel standard FFT.

(c) Multiply (homomorphically, via an automorphism) the ith output register in each group by ω^{-i}, for all i from 1 to $N-1$. More specifically, we apply the automorphism $X \mapsto X^{\omega^{-i}}$, corresponding to multiplication by ω^{-i} in the exponent, followed by a key switching operation.

The following algorithms provide pseudocodes for the above procedure to compute the homomorphic PFT^{-1}. More specifically, Algorithm 3 describes step (a) and then calls Algorithm 4 for each of the groups of registers. Algorithm 4 describes a primitive length-N (inverse) FFT for a single group of size N, consisting of a standard (cyclic) FFT (step (b)) as well as the multiplication by ω^{-i} (step (c)). It remains to describe more precisely what happens in the

Algorithm 3 IFFT stage of bootstrapping (`BootstrapIFFT`)

1: **Input:** a list of $\phi(d)$ registers REG, k, N, a list of radices $\{r_i\}_{0\le i<\ell}$, and ω
Require: ω a primitive $2N$th root of unity mod q, $\prod_{0\le i<\ell} r_i = N$, and $kN = \phi(d) = \mathrm{len}(\mathrm{REG})$
2: **for all** $0 \le j < k$ **do**
3: $\quad \mathrm{REG}[j, k+j, \ldots, (N-1)k+j] \leftarrow \text{N-IFFT}(\mathrm{REG}[j, k+j, \ldots, (N-1)k+j], \{r_i\}, \omega)$
4: **end for**

Algorithm 4 Primitive length-N IFFT for a single group of size N (`N-IFFT`)

1: **Input:** List of RLWE′ registers $\mathrm{REG} = \{\mathrm{RLWE}'(X^{(\tilde{\mathbf{a}}_i \cdot \tilde{\mathbf{z}}_i)^{(j)}})\}_{0\le i<N}$ for some fixed $0 \le j < k$, list of radices $\{r_i\}_{0\le i<\ell}$, primitive $2N$th root of unity ω modulo q.
2: **Output:** list of RLWE registers $\mathrm{REG} = \{\mathrm{RLWE}(X^{(\mathbf{a}_i \cdot \mathbf{z}_i)^{(j)}})\}_{0\le i<N}$.
3: **let** $\omega' = \omega^{-2}$
4: $\mathrm{REG} \leftarrow \mathrm{FFT}(\mathrm{REG}, \{r_i\}_{0\le i<\ell}, \omega')$ ▷ Step 4-(b)
5: **for** $i \leftarrow 1, \ldots N-1$ **do**
6: $\quad \mathrm{REG}[i] \leftarrow \mathit{EvalAut}(\mathrm{REG}[i], \omega^{-i})$ ▷ Step 4-(c)
7: **end for**
8: **return** REG

(cyclic) FFT call, line 4 of Algorithm 4.

Recall that FFT is a recursive algorithm that follows the structure of a remainder tree, see the procedure `FFT` given in Algorithm 6. We will now focus on what happens in a single layer of the FFT as described in the second procedure `FFT Layer` in Algorithm 6.

At a single layer: Let r_i be the radix used for the i-th FFT layer for $0 \le i < \ell$. Then, for $R_i := \prod_{i\le i'<\ell} r_i$, the inputs to the i-th FFT layer are the coefficients of N/R_i polynomials modulo $(X^{R_i} - \omega'^j)$ (for varying values of j), each with R_i coefficients.[4] Hence this corresponds to a total of N coefficients, *i.e.,* N registers. The outputs are the coefficients of N/R_{i+1} polynomials

[4] When $i = 0$, it starts with a single input polynomial modulo $X^N - \omega^0$.

modulo $(X^{R_{i+1}} - \omega'^{j'})$ ranging over all j' such that $r_i \cdot j' \equiv j \mod N$. Note that the total number of coefficients remains the same, *i.e.,* we still have N registers.

Let us now consider a single input polynomial (out of N/R_i), *i.e.,* one of the nodes in the remainder tree, and describe what computations are needed to produce the children nodes. This subroutine is described in Algorithm 5, called `FFT Subroutine` and is repeated for every node (meaning polynomial) of the layer, hence N/R_i times. We illustrate the reduction of this polynomial via an example to better describe the operations needed in this subroutine.

Algorithm 5 `FFT Subroutine`

Input: Radix r which divides R for $R \mid N$, index j, and RLWE$'$ ciphertexts $\{ct_i\}_{0 \le i < R}$ storing coefficients of a single polynomial mod $(x^R - \omega'^j)$
Output: r tuples each of which consists of index j' such that $r \cdot j' \equiv j \mod N$, and R/r RLWE$'$ ciphertexts
for all $0 \le i < R - R/r$ **do**
  $ct_i \leftarrow SwitchToRGSW(ct_i)$
end for
if this is the final FFT layer **then**
  for all $R - R/r \le i < R$ **do**
    let $S = \frac{Q}{4}$ ▷ 4 is the plaintext modulus after bootstrapping
    $ct_i \leftarrow S \odot ct_i$ ▷ ct_i is now RLWE instead of RLWE$'$
  end for
end if
let $\{j'_0, ..., j'_{r-1}\}$ = the set of all j''s satisfying $rj' \equiv j \mod N$
for all $0 \le v \le r - 1$ **do**
  let $\zeta = \omega'^{j'_v}$
  for all $0 \le i < R/r$ **do**
    $accum[v][i] \leftarrow ct_{R-R/r+i}$
    for $\kappa \leftarrow [2, 3, ..., r]$ **do**
      $accum[v][i] \leftarrow EvalAut(accum, \zeta)$
      $accum[v][i] \leftarrow MulRGSW(ct_{R-\kappa \cdot R/r+i}, accum)$
    end for
  end for
end for
output r tuples $(j'_v, accum[v])$ for $0 \le v < r$

Example 1. We describe in this example the reduction from an input polynomial to a single child node for the simple radix-2 FFT. Assume we have as input a polynomial of the form

$$g_0 + g_1X + g_2X^2 + g_3X^3 + g_4X^4 + g_5X^5 + g_6X^6 + g_7X^7$$

and we want to reduce it modulo $(X^2 - \zeta)$. Similarly as for pointwise multiplication, it is possible to compute the coefficient terms for each monomial X^j. In our example, we would have constant coefficient

$$g_0 + \zeta g_2 + \zeta^2 g_4 + \zeta^3 g_6$$

and X coefficient

$$g_1 + \zeta g_3 + \zeta^2 g_5 + \zeta^3 g_7.$$

In the remainder tree, this operation would have to be repeated for r different values of ζ, in particular for this example, four different values.

Algorithm 6 Full radix-r standard FFT (`FFT`)

procedure FFT($\{\text{REG}\}$, $\{r_i\}_{0\leq i<\ell}$, ω')
  $state \leftarrow \{(N, \text{REG})\}$ ▷ List of tuples
  for i in $[0, 1, ..., \ell-1]$ **do**
    $state \leftarrow$ FFT LAYER($r_i, \omega', state$)
  end for
  return REG
end procedure

procedure FFT LAYER(r_i, ω', list of tuples)
  Input: N/R_i tuples of the form $(j, \{ct_{j,0}, \ldots, ct_{j,R_i-1}\})$, where each $ct_{j,v}$ is an RLWE′ ciphertext
  ▷ $ct_{j,v}$ represents the v-th coefficient of a polynomial mod $(X^{R_i} - \omega'^j)$
  Output: N/R_{i+1} tuples of the form $(j', \{ct_{j',0}, \ldots, ct_{j',R_{i+1}-1}\})$.
  ▷ Each input j has r_i corresponding outputs j' such that $r_i \cdot j' \equiv j \bmod N$.
  ▷ $ct_{j',v}$ represents the v-th coefficient of a polynomial mod $(x^{R_{i+1}} - \omega'^{j'})$
  for all $(j, \{ct_{j,0}, \ldots, ct_{j,R_i-1}\})$ in input **do**
    FFT_SUBROUTINE($\{ct_{j,0}, \ldots, ct_{j,R_i-1}\}, r_i, j, \omega'$)
  end for
end procedure

The homomorphic circuit to perform this reduction is illustrated in Figure 3. We recall that the input coefficients g_i (both in the example and in Figure 3) correspond to registers, in particular RLWE′ ciphertexts. The main operations needed for a reduction are scheme-switching for most of the coefficients, multiplication by a power of ζ, which can be done using automorphisms, and addition which corresponds to RLWE′ $\times$ RGSW multiplications.

Remark 2. The scheme switches at the beginning of the circuit convert RLWE′ ciphertexts to RGSW ciphertexts for all coefficients except the last. As mentioned previously, the circuit for the same coefficients is performed for various values of ζ. For all these cases, the scheme-switching operations need only to be performed once (as the coefficients do not change) and thus the cost will be amortized.

Details of this subroutine are given in Algorithm 5. Algorithm 6 provides the pseudocode for all $\phi(d)$ registers (`FFT Layer`) as well as the full FFT algorithm where all layers are considered (`FFT`).

One can note from Algorithm 5 that the case of the last layer of the FFT slightly differs (see line 6). Indeed, it is possible to optimize the running-time of the FFT algorithm by modifying the nature of the elements considered in the very last layer of the FFT. Indeed, one can notice that the outputs of the IFFT only need to be RLWE ciphertexts, not RLWE′ ciphertexts. Hence, one can save operations by using RLWE registers instead of RLWE′ registers when possible. While RLWE cannot be scheme-switched to RGSW without blowing up the error, we can modify the last IFFT layer to use RLWE instead of RLWE′ for the registers that do not get scheme-switched (this corresponds to *accum* in Algorithm 5 or g_r in Figure 3). Concretely, each of the $\phi(d)/r$ non-scheme-switched RLWE′ ciphertexts would be converted to RLWE by an $\mathcal{R} \odot$ RLWE′ operation with $\mathcal{R}$ element $\lceil Q/4 \rfloor$, where 4 is the plaintext modulus the msbExtract stage expects.

This concludes the description of the homomorphic computation of $\mathbf{a} \cdot \mathbf{z}$. The output of this multiplicative step is thus $\phi(d)$ RLWE registers, each encrypting a coefficient of $\mathbf{a} \cdot \mathbf{z} \in \mathcal{R}_{in}$.

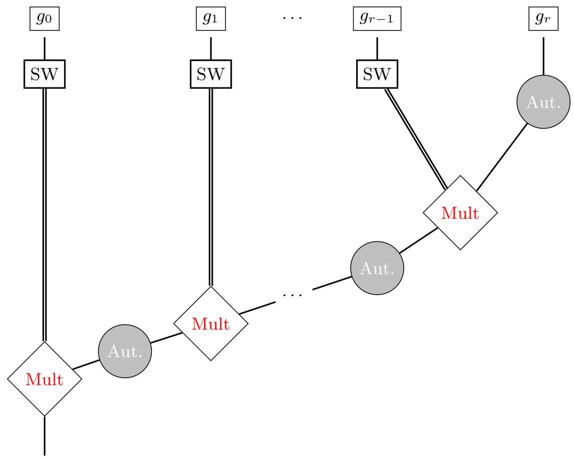

Fig. 3. One layer of FFT for a single input polynomial. A single line corresponds to RLWE ciphertexts and a double line to RGSW ciphertexts. SW stands for scheme-switching, Aut. for automorphisms and Mult for multiplication.

Adding b From the previous step, we have obtained registers encoding the coefficients of $\mathbf{a} \cdot \mathbf{z}$. We also have the polynomial $\mathbf{b}$ in the clear. In order to obtain registers encoding the coefficients of $\mathbf{a} \cdot \mathbf{z} + \mathbf{b}$, we add $\mathbf{b}$ via fixed rotations, *i.e.*, scaling the ciphertext by a monomial. Concretely, to add a coefficient b_i to a register RLWE$(X^{(\mathbf{a} \cdot \mathbf{z})_i})$, we simply scale the RLWE ciphertext by X^{b_i} resulting in RLWE$(X^{(\mathbf{a} \cdot \mathbf{z}+\mathbf{b})_i})$. Since X^{b_i} has norm 1, it does not increases the noise of the register. We thus now have $\phi(d)$ registers encoding the coefficients of $\mathbf{a} \cdot \mathbf{z} + \mathbf{b}$, as expected. This concludes the linear step of the algorithm.

4.3 msbExtract

The linear step outputs $\phi(d)$ RLWE registers each encrypting $\frac{Q}{4} X^{c_i}$, where each $c_i \in \mathbb{Z}_q$ is a noisy (un-rounded) decryption of the ith input ciphertext. We now operate separately on each register (and drop the subscript i) in order to recover from each register a plain-LWE encryption of $f(c)$, for some function f that applies rounding and allows for computation. Output ciphertexts have plaintext modulus 4; to compute NAND gates, it suffices for $f(c)$ to be 1 for $c \in [-q/8, 3q/8)$ and 0 elsewhere (we refer to [13] for more details). Focusing on a single register $(\mathbf{a}, \mathbf{b}) \in {\mathcal{R}_{reg}}^2$, we have

$$\mathbf{b}(X) = -\mathbf{a}(X) \cdot \mathbf{s}(X) + \mathbf{e}(X) + \frac{Q}{4}\mathbf{m}(X) \pmod{Q, \Phi_q(X)}$$

where $\mathbf{m}(X) = X^c$. Looking at a single coefficient of these polynomials, the ring product $\mathbf{a}(X) \cdot \mathbf{s}(X)$ will become a vector inner product between the coefficients of $\mathbf{s}$ and some permuted coefficients of $\mathbf{a}$. Because we use prime q, these polynomials are degree $\leq q-2$, and $X^{q-1} \equiv -1 - X - \cdots - X^{q-2}$. Note that, as for error growth, the fact that we consider prime cyclotomics instead of power-of-2 cyclotomics slightly changes the setting. We get for a single coefficient

$$
\begin{aligned}
b_i = -a_i s_0 - a_{i-1}s_1 - \cdots - a_0 s_i - 0 \cdot s_{i+1} - a_{q-2}s_{i+2} - a_{q-3}s_{i+3} - \cdots - a_{i+2}s_{q-2} \\
+ a_{q-2}s_1 + a_{q-3}s_2 + \cdots + a_2 s_{q-3} + a_1 s_{q-2} \\
+ \frac{Q}{4}m_i + e_i
\end{aligned}
$$

which can be re-written as

$$
\begin{aligned}
\frac{Q}{4}m_i + e_i = b_i + \{(a_i, a_{i-1}, \ldots, a_0, 0, a_{q-2}, a_{q-1}, \ldots, a_{i+2}) \\
- (0, a_{q-2}, a_{q-1}, \ldots, a_1)\} \cdot (s_0, s_1 \ldots, s_{q-2})
\end{aligned}
$$

For $0 \le i \le q-2$, letting $\vec{a_i}$ denote the above vector $(a_i, a_{i-1} - a_{q-2}, \ldots, a_{i+2} - a_1)$, we then have that $(\vec{a_i}, b_i)$ is an LWE encryption with noise e_i under secret key $\mathbf{s}_p = (s_0, \ldots, s_{q-2})$ of message m_i. Note that m_i is 1 if $c = i$, -1 if $c = q-1$, and 0 otherwise. To produce an encryption of $f(c)$, which should be 1 for $c \in (-q/8, 3q/8)$ and 0 elsewhere, we simply sum the relevant $(\vec{a_i}, b_i)$:

$$
\sum_{i=\lceil 7q/8 \rceil}^{q-2} (\vec{a_i}, b_i) + \sum_{i=0}^{\lfloor 3q/8 \rfloor - 1} (\vec{a_i}, b_i)
$$

taking care to ensure the number of summands is 3 mod 4, so that when $c = q-1$ the sum is 1 mod 4 as desired. (When $q \equiv 1 \bmod 8$, this will be the case for the summation written above.)

This gives us an LWE encryption with plaintext modulus 4 and ciphertext modulus Q under a key $\mathbf{s}_p \in \mathbb{Z}^{q-1}$. To conclude bootstrapping, we can keyswitch back to the orignal plain LWE secret key, and modulus switch back down to the original (much smaller than Q) ciphertext modulus.

5 Analysis

In order to evaluate the performance of our algorithm, we analyse its running-time as well as the error growth. We will first show that our homomorphic decryption procedure takes no more than $\mathcal{O}((k + r \cdot \ell)\phi(d)d_B)$ homomorphic operations.

5.1 Counting homomorphic operations

We will evaluate the efficiency of our algorithm by first measuring the time complexity in terms of the number of $\mathcal{R} \odot \mathrm{RLWE}'$ operations performed. We have already summarized in Section 3.2, Table 2 the number of $\mathcal{R} \odot \mathrm{RLWE}'$ operations needed for the main operations used in our scheme: scheme switching, automorphisms (with key switching) and $\mathrm{RGSW} \times \mathrm{RLWE}'$. We now describe the number of $\mathcal{R} \odot \mathrm{RLWE}'$ operations for the various steps of our algorithm.

Pointwise multiplication Based on the description given in Section 4.2, we have the following analysis. For a single coefficient X^j in the computation of the inner product, our algorithm performed k automorphisms and $(k-1)$ $\mathrm{RGSW} \times \mathrm{RLWE}'$ multiplications. Hence the number of $\mathcal{R} \odot \mathrm{RLWE}'$ operations per register is $(3k-2)d_B$. This computation needs to be repeated for all k coefficients of a single product and for $\phi(d)/k$ pairs of $(\tilde{\mathbf{a}}_i, \tilde{\mathbf{z}})$ polynomials. Thus the total number of $\mathcal{R} \odot \mathrm{RLWE}'$ operations for the entire pointwise multiplication algorithm is $(3k-2)\phi(d)d_B$. This result is summarized in Table 3.

Partial inverse FFT Based on the description given in Section 4.2, considering a single register and a single radix-r layer of FFT, the algorithm computes $(r-1)$ automorphisms, $(r-1)$ $\mathrm{RGSW} \times \mathrm{RLWE}'$ multiplications but only amortized $(1-1/r)$ scheme switches as explained in Section 4.2. Thus the total number of operations per layer is

$$\left((r-1)+2(r-1)+\left(1-\frac{1}{r}\right)\right)\phi(d)d_B = \left(3r-2-\frac{1}{r}\right)\phi(d)d_B$$

operations. This result is summarized in Table 3.

Last layer of IFFT optimization: Recall that the outputs of the IFFT only need to be RLWE ciphertexts and not RLWE$'$ ciphertexts. This allowed us to optimize the cost of the last layer of the IFFT by using RLWE registers instead of RLWE$'$ registers when possible. By using this modification, the multiplications and automorphisms in this layer will use a factor of d_B fewer operations. Thus the total number of operations for the last layer is only

$$\left((r-1)+2(r-1)+(1/r)+\left(1-\frac{1}{r}\right)d_B\right)\phi(d) = \left(3r-3+\frac{1}{r}+d_B\left(1-\frac{1}{r}\right)\right)\phi(d).$$

This result is summarized in Table 3.

After the last layer: Recall that the very last step of the homomorphic partial IFFT is to multiply the i^{th} output register in each group by ω^{-i}, via an automorphism that sends X to $X^{\omega^{-i}}$. Operation-wise, this corresponds to one automorphism per register. Since with the last-layer optimization the registers are RLWE instead of RLWE$'$, this corresponds to $\phi(d)$ operations in total.

Adding b Recall that to add b_i to the register $\mathrm{RLWE}(X^{(\mathbf{a}\cdot\mathbf{z})_i})$, we simply scaled the RLWE ciphertext by X^{b_i}. No $\mathcal{R} \odot \mathrm{RLWE}'$ operations are involved. This result is summarized in Table 3.

Steps of the algorithm	$\mathcal{R} \odot \mathrm{RLWE}'$ operations
Partial FFT of $\mathbf{a}$	–
Pointwise multiplication	$(3k-2)\phi(d)d_B$
Partial IFFT (per layer)	$(3r-2-\frac{1}{r})\phi(d)d_B$
Last layer of IFFT	$(3r-3+\frac{1}{r}+d_B(1-\frac{1}{r}))\phi(d)$
Last IFFT step	$\phi(d)$
Adding $\mathbf{b}$	–

Table 3. Summary of $\mathcal{R} \odot \mathrm{RLWE}'$ operation count.

5.2 Error growth

Similarly as for the operation count, we have already summarized in Section 3.2, Table 2 the error growth coming from the scheme switching, automorphisms (with key switching) and $\mathrm{RGSW} \times \mathrm{RLWE}'$

operations. We now describe the error growth resulting from the various steps of our algorithm based on the error variance for each of these operations and the operation count described in the previous section.

Pointwise multiplication Recall from the description given in Section 4.2 that the algorithm starts with an inital RLWE$'$ ciphertext, denoted as *accum*, which is a "fresh" ciphertext from the evaluation key with error variance $\sigma^2_{eval\_key}$. Each automorphism performed during pointwise multiplication adds $\sigma^2_{\odot,aut\_key}$ error variance, and each multiplication with a fresh RGSW ciphertext adds $2\sigma^2_{\odot,\mathrm{RGSW}}$ error variance. Hence, the error variance after pointwise multiplication is

$$(3k-2)(\sigma^2_{\odot,aut\_key} + 2\sigma^2_{\odot,\mathrm{RGSW}}) + \sigma^2_{eval\_key}.$$

This result is summarized in Table 4.

Inverse FFT Again, recall that each automorphism adds $\sigma^2_{\odot,aut\_key}$ error variance. The output of schemeswitching has $\sigma^2_{sw} = \sigma^2_{\odot,eval\_key} + \ell_1(s)\sigma^2_{in}$ error variance. Let σ^2_{accum} be initialized as σ^2_{in}. Each automorphism and RGSW $\times$ RLWE$'$ multiplication (performed a total amount of $r-1$ times) updates the variance as $\sigma^2_{accum} \leftarrow \sigma^2_{accum} + \sigma^2_{\odot,aut\_key}$ and $\sigma^2_{accum} \leftarrow \sigma^2_{accum} + 2\sigma^2_{\odot,sw}$ where $\sigma^2_{\odot,sw} \leq \frac{B^2}{6} d_B q \sigma^2_{sw}$. Hence, in total, the error variance after a radix-r layer becomes

$$\sigma^2_{in} + (r-1)(\sigma^2_{\odot,aut\_key} + 2\sigma^2_{\odot,sw})$$

for $\sigma^2_{\odot,sw} \leq \frac{B^2}{6} d_B q \sigma^2_{sw}$ This result is summarized in Table 4.

After last layer Multiplying the i^{th} output register in each group by ω^{-i} with an automorphism increases (additively) the error variance by $\sigma^2_{\odot,aut\_key}$. This result is summarized in Table 4.

Adding b Scaling by a monomial does not increase the error. This result is summarized in Table 4.

Algorithms	Error growth
Partial FFT of **a**	–
Pointwise multiplication	$(3k-2)(\sigma^2_{\odot,aut\_key} + 2\sigma^2_{\odot,\mathrm{RGSW}}) + \sigma^2_{eval\_key}$.
Partial IFFT (per layer)	$\sigma^2_{in} + (r-1)(\sigma^2_{\odot,aut\_key} + 2(\sigma^2_{\odot,eval\_key} + \ell_1(s)\sigma^2_{in}))$
Last IFFT step	$\sigma^2_{\odot,aut\_key}$
Adding **b**	–

Table 4. Summary of error growth.

5.3 Asymptotic analysis

Let $\lambda = O(n)$ the be security level considered. We study the performance of our algorithm as λ increases, *i.e.,* when n tends to infinity. Recall that the other parameters used in our algorithm are $d_B = \lceil \log_B Q \rceil = O(\log n)$, the number of layers ℓ (*i.e.,* the multiplicative depth) and $k = r = \phi(d)^{1/\ell}$ (where we recall that k is the degree at which we stop the partial FFT and r is the radix for an FFT layer).

Theorem 2. *Let $\phi(d) = O(n)$ be the number of packed ciphertexts and $q, Q = poly(n)$ the moduli of the rings considered. The total cost of bootstrapping (non-amortized) then corresponds to $O(n^{1+\frac{1}{\ell}} \cdot \log n \cdot \ell)$ homomorphic operations (in terms of the number of $\mathcal{R} \odot \mathrm{RLWE}'$ operations).*

Proof. The number of $\odot$ operations in the pointwise multiplication step is $(3k-2)\phi(d)d_B$ which asymptotically corresponds to $O(n^{1+\frac{1}{\ell}} \log n)$. Similarly, the inverse FFT requires $(3r-2-\frac{1}{r})\phi(d)d_B\ell$ operations (without including the last layer modification which asymptotically does not change the result) which asymptotically gives $O(n^{1+\frac{1}{\ell}} \cdot \log n \cdot \ell)$. $\square$

Corollary 2. *The amortized cost per message is $O(n^{\frac{1}{\ell}} \cdot \log n \cdot \ell)$ homomorphic operations (in terms of the number of $\mathcal{R} \odot \mathrm{RLWE}'$ operations).*

5.4 Comparison with previous work

We compare the performance of our algorithm with two lines of work: sequential bootstrapping algorithms such as FHEW/TFHE [4, 5] and the amortized bootstrapping algorithm given in [14]. To compare the performance of our algorithm, we look at asymptotic analyses and report costs in terms of homomorphic operations (for fair comparison with previous works). Table 5 summarizes the comparisons between our work and previous works.

Comparing with sequential FHEW/TFHE : see Table 5.

Comparing with amortized work [14]: The total cost (non-amortized) in [14] is $O(3^{1/\epsilon} n^{1+\epsilon})$ for some $\epsilon < 1/2$.

Scheme	Total cost	Number of messages	Amortized cost
FHEW	$\tilde{O}(n)$	1	$\tilde{O}(n)$
TFHE	$O(n)$	1	$O(n)$
[14]	$\tilde{O}(3^\ell \cdot n^{1+1/\ell})$	$O(n)$	$\tilde{O}(3^\ell \cdot n^{1/\ell})$
our work	$O(\ell \cdot n^{1+1/\ell})$	$O(n)$	$O(\ell \cdot n^{1/\ell})$

Table 5. Comparing asymptotic cost of various bootstrapping algorithms in terms of homomorphic operations, where ℓ corresponds to the recursive depth in each algorithm. For uniformity with previous work, performance is expressed as the number of RGSW $\times$ RGSW products, or equivalent operations. Alternatively, the number of basic $\mathcal{R} \odot \mathrm{RLWE}'$ products can be obtained by multiplying these figures by $O(\log n)$, the length of the gadget vector.

References

1. Alperin-Sheriff, J., Peikert, C.: Practical bootstrapping in quasilinear time. In: Canetti, R., Garay, J.A. (eds.) Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I. Lecture Notes in Computer Science, vol. 8042, pp. 1–20. Springer (2013). , https://doi.org/10.1007/978-3-642-40041-4_1
2. Bonnoron, G., Ducas, L., Fillinger, M.: Large FHE gates from tensored homomorphic accumulator. In: Joux, A., Nitaj, A., Rachidi, T. (eds.) Progress in Cryptology - AFRICACRYPT 2018 - 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7-9, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10831, pp. 217–251. Springer (2018). , https://doi.org/10.1007/978-3-319-89339-6_13
3. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. ACM Trans. Comput. Theory **6**(3), 13:1–13:36 (2014). , https://doi.org/10.1145/2633600
4. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: fast fully homomorphic encryption over the torus. Journal of Cryptology **33**(1), 34–91 (2020)
5. Ducas, L., Micciancio, D.: FHEW: bootstrapping homomorphic encryption in less than a second. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 617–640. Springer (2015)
6. Gentry, C.: A fully homomorphic encryption scheme. Stanford university (2009)
7. Gentry, C., Halevi, S., Smart, N.P.: Fully homomorphic encryption with polylog overhead. In: Pointcheval, D., Johansson, T. (eds.) Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7237, pp. 465–482. Springer (2012). , https://doi.org/10.1007/978-3-642-29011-4_28
8. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7417, pp. 850–867. Springer (2012). , https://doi.org/10.1007/978-3-642-32009-5_49
9. Guimarães, A., Pereira, H.V.L., van Leeuwen, B.: Amortized bootstrapping revisited: Simpler, asymptotically-faster, implemented. Cryptology ePrint Archive, Paper 2023/014 (2023), https://eprint.iacr.org/2023/014, https://eprint.iacr.org/2023/014
10. Halevi, S., Shoup, V.: Faster homomorphic linear transformations in helib. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10991, pp. 93–120. Springer (2018). , https://doi.org/10.1007/978-3-319-96884-1_4
11. Lee, Y., Micciancio, D., Kim, A., Choi, R., Deryabin, M., Eom, J., Yoo, D.: Efficient FHEW bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption. Cryptology ePrint Archive (2022)
12. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. J. ACM **60**(6), 43:1–43:35 (2013). , https://doi.org/10.1145/2535925
13. Micciancio, D., Polyakov, Y.: Bootstrapping in FHEW-like cryptosystems. In: Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography. pp. 17–28 (2021)
14. Micciancio, D., Sorrell, J.: Ring packing and amortized FHEW bootstrapping. In: Chatzigiannakis, I., Kaklamanis, C., Marx, D., Sannella, D. (eds.) 45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic. LIPIcs, vol. 107, pp. 100:1–100:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). , https://doi.org/10.4230/LIPIcs.ICALP.2018.100
15. Stehlé, D., Steinfeld, R., Tanaka, K., Xagawa, K.: Efficient public key encryption based on ideal lattices. In: Matsui, M. (ed.) Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5912, pp. 617–635. Springer (2009). , https://doi.org/10.1007/978-3-642-10366-7_36