Faster amortized bootstrapping using the incomplete NTT for free

Thales B. Paiva[1], Gabrielle De Micheli[1], Syed Mahbub Hafiz[1]
Marcos A. Simplicio Jr.[1,2] and Bahattin Yildiz[1]

[1] Future Security Team, LG Electronics USA, Inc.
[2] Universidade de São Paulo, São Paulo, Brazil
{thales.paiva,gabrielle.demicheli,syedmahbub.hafiz,bahattin.yildiz}@lge.com
msimplicio@larc.usp.br

Abstract. Amortized bootstrapping techniques have been proposed for FHEW/TFHE to efficiently refresh multiple ciphertexts simultaneously within a polynomial modulus. Although recent proposals have very efficient asymptotic complexity, reducing the amortized cost essentially to $\tilde{O}(1)$ FHE multiplications, the practicality of such algorithms still suffers from substantial overhead and high decryption failure rates (DFR). In this study, we improve upon one of the state-of-the-art amortized bootstrapping algorithms (Guimarães et al., ASIACRYPT 2023) for FHEW/TFHE-like schemes by introducing an alternative algorithmic strategy. Specifically, we combine Guimarães et al.'s strategy based on a two-part NTT with an incomplete Number Theoretic Transform (NTT) algorithm. The resulting construction is such that the multiplication of higher-degree polynomials that would usually create a bottleneck in an incomplete NTT setting actually comes for free. As a result, we demonstrate a 2.12× speedup compared to the algorithm of Guimarães et al. and a 1.12× improvement over the state-of-the-art (sequential) TFHE-rs while achieving a DFR close to 2^{-32} for 7-bit messages, although the DFR is higher for 8-bit messages. We also explore trade-offs between execution time and DFR, identifying parameter sets that improve the execution time of Guimarães et al. by 1.41×, while simultaneously reducing the DFR by a factor of 2^{-22} for 8-bit messages.

Keywords: Fully homomorphic encryption · FHEW/TFHE · amortized bootstrapping · incomplete NTT · efficient implementation

1 Introduction

Fully homomorphic encryption (FHE) was first introduced by Gentry in [Gen09] and has since been extensively studied and improved both in terms of efficiency and security. Although most modern FHE schemes are based on the Ring Learning With Errors (RingLWE) problem [SSTX09, LPR10], different strategies have been adopted in their construction. In schemes such as BGV/BFV [BGV14, Bra12, FV12], ciphertexts encrypt vectors of values, while operations on ciphertexts (e.g., addition and multiplication) are performed component-wise using ring operations. Such schemes benefit from an amortized operation cost by simultaneously performing computations on all the entries of the ciphertext vector. However, this comes at the cost of an expensive bootstrapping procedure and large parameters. Such large parameters also imply the need for stronger security assumptions, like relying on worst-case lattice-based problems with super-polynomial approximation factors.

In contrast, schemes such as FHEW/TFHE [DM15, CGGI20] operate at the bit level, and ciphertexts are LWE encryptions without additional ring structure. Nonetheless, the ring structure can still be explored in the context of programmable bootstrapping, where

an encryption of $f(m)$ is computed for any function f and message m. By combining homomorphic LWE addition and programmable bootstrapping, FHEW/TFHE can evaluate any circuit, i.e., perform arbitrary homomorphic computations. The resulting schemes benefit from simpler and, in principle, faster bootstrapping algorithms than BGV/BVF. Moreover, they use much smaller parameters, allowing for (weaker) security assumptions based on worst-case lattice-based problems with polynomial approximation factors – similarly to the assumptions found in general lattice-based public key encryption schemes.

However, when closely analyzing the bootstrapping procedures of FHEW/TFHE and BGV/BFV, which are the main bottlenecks in those schemes, the latter has one clear perk: the amortization resulting from packing several messages into a large ring ciphertext greatly outweighs its higher execution costs. In light of this observation, amortized bootstapping algorithms were also proposed to refresh multiple FHEW/TFHE ciphertexts at once. The ultimate goal of such proposals is to bridge the gap between the BGV/BFV and FHEW/TFHE strategies, while preserving the latter's weaker security assumptions. Our work follows this trend, contributing to a line of investigation that attempts to answer the following research question: *can we bring amortized bootstrapping algorithms to be as efficient as (or more efficient than) their sequential counterparts?*

Related Works. The first amortized method for FHEW-like ciphertexts was proposed by Micciancio and Sorrell in [MS18]. The work shows that the cost of bootstrapping n ciphertexts can be reduced from $\tilde{O}(n)$ cryptographic operations to $\tilde{O}(n^\epsilon)$, for any $\epsilon > 0$. The main idea of their method is to pack n LWE input ciphertexts into a single RingLWE ciphertext, and then perform the bootstrapping operation over the latter to amortize the overall cost over n messages. The main bottleneck of the resulting bootstrapping procedure is that it requires a homomorphic polynomial multiplication. Since FHEW [DM15] uses RingGSW registers to encrypt messages m in the exponent as monomials X^m, homomorphic addition is rather straightforward, as it translates to ciphertext multiplication. However, other homomorphic operations required by fast polynomial multiplication algorithms, such as subtraction or multiplication by so-called "twiddle factors", become much more challenging. To address this issue, the original algorithm employs (a variant of) the Nussbaumer Transform [Nus80] to perform fast polynomial multiplication. Although the proposed method offers a major asymptotic performance improvement, the algorithm is hardly practical due to a large constant $2^{O(1/\epsilon)}$ hidden in the asymptotic notation.

In subsequent works [GPVL23, DMKMS24], the Nussbaumer Transform is replaced with the much more practical Number Theoretic Transform (NTT), where the multiplication by twiddle factors is done via exponentiations, taking advantage of ring automorphisms. This approach for homomorphic multiplication is not new, as it was used in FHEW-related algorithms in [BDF18, LMK+23], for example. The result is that [GPVL23, DMKMS24] provide a much smaller overhead than [MS18], effectively reducing the underlying costs from $2^{O(1/\epsilon)} \cdot n^{1/\epsilon}$ to $(1/\epsilon) \cdot n^{1/\epsilon}$. Although both algorithms follow the same structure, [DMKMS24] centers its algorithm around RingLWE registers to perform exponentiations, while [GPVL23] extends the automorphism multiplication technique to work directly on RingGSW ciphertexts. Moreover, the latter work proposes a concrete C++ implementation of their algorithm, which is the main target of our proposed improvements.

In another recent result, Liu and Wang [LW23a, LW23b] introduced a new algebraic framework for batch homomorphic computation relying on the tensoring of three rings. This alternative framework improves the efficiency of amortized boostrapping algorithms, using $\tilde{O}(1)$ homomorphic multiplications while still considering a polynomial modulus. While the asymptotic cost is (almost) optimal, it is shown in [DMKMS24] that the batch algorithm presented in [LW23b] suffers from the same limitations as [MS18]: its actual overhead is very high, making the algorithm impractical for concrete parameters. To overcome this issue while preserving the same asymptotic complexity as [LW23b], Liu and

Wang [LW23c] proposed an alternative algorithm that is efficient for concrete parameters. Their strategy relies on a scheme-switching technique from FHEW/TFHE to BFV/BGV ciphertexts, whose drawback is the need for using superpolynomial modulus instead of the polynomial moduli commonly found in all aforementioned schemes.

Concurrently to our work, Guimarães and Pereira [GP25] improved upon their previous amortized bootstrapping algorithm [GPVL23] by introducing a novel multiplication technique based on sparse polynomials. To do so, they rely on a stronger RLWE assumption with sparse secrets and, while their practical performance is considerably improved, the resulting amortized asymptotic complexity is worse than in their original work.

While many of the aforementioned works explore amortized bootstrapping, most lack practical and efficient implementations. Notably, [DMKMS24] and [LW23a, LW23b] provide nice asymptotic improvements, but remain impractical and unimplemented. Conversely, [GPVL23] stands out as both theoretically efficient and practically relevant, including an open-source implementation.

Contributions. In this paper, we enhance the amortized bootstrapping approach by Guimarães et al. [GPVL23] by targeting its NTT-based homomorphic polynomial multiplication required for homomorphic decryption. Specifically, their algorithm operates in cyclotomic rings $\mathbb{Z}_p[X]/(X^N+1)$, where N is a power of two and p is a prime modulus. When $p \equiv 1 \pmod{2N}$, such rings are known to support fast NTT computations using FFT-like iterative algorithms. However, [GPVL23] observes that these cannot be used in a homomorphic setting because they lead to a large error growth, raising the scheme's decryption failure rate (DFR). They then propose a two-part NTT to provide a good balance between performance and error growth.

In contrast, our proposal for homomorphic polynomial multiplication leverages the concept of incomplete NTT [LS19, ABD+21, Kan22, DMKMS24, HYJ+25]. This requires a redesign of the homomorphic decryption algorithm to account for more complex multiplication in the NTT domain, as well as a different two-part inverse NTT (INTT) algorithm. This multiplication of higher-degree polynomials usually creates a bottleneck in an incomplete NTT setting, actually comes *for free* in our novel construction. We can then leverage one important property of the incomplete NTT: it allows for relaxed constraints and supports a wider set of parameters. Specifically, while the complete NTT in cyclotomic rings of the form $\mathbb{Z}_p[X]/(X^N+1)$ requires $p \equiv 1 \pmod{2N}$, the ℓ-incomplete NTT only requires the less restrictive $p \equiv 1 \pmod{2N/2^\ell}$. This flexibility allows using a smaller prime modulus p, which is one source of performance gains in the amortized bootstrapping algorithm we propose. At the same time, the broader choices of p allow us to explore new tradeoffs between DFR and performance.

In summary, our contributions are as follows:

1. *Algorithmic contribution.* We introduce an improved amortized bootstrapping algorithm that exploits incompleteness in homomorphic (inverse) NTT operations without incurring any additional overhead. Specifically, we propose an alternative two-part incomplete NTT that generalizes the two-part inverse NTT used in [GPVL23], while preserving the same algorithmic complexity. This generalization enables using a radix-2 incomplete NTT formulation, setting the stage for different performance gains and speed-DFR tradeoffs.

2. *Performance improvements and tradeoffs.* We explore different parameter settings and introduce concrete, optimized amortized bootstrapping instantiations that are faster than the state of the art. In particular, our experimental results show up to a 2.12× improvement in execution time compared to [GPVL23] and a 1.12× speedup compared to the state-of-the-art (sequential) TFHE-rs [Zam22] for a DFR close to 2^{-32} for 7-bit messages, although the DFR is higher for 8-bit messages. In

addition, we analyze the tradeoff between execution time and DFR in our algorithm and identify configurations where execution time improves by 1.41× over [GPVL23], while simultaneously reducing the DFR from 2^{-17} to 2^{-39} for 8-bit messages. Finally, we demonstrate that DFR can be reduced below that of TFHE-rs ($2^{-131} < 2^{-128}$) at the cost of a 2.9× performance slowdown for 8-bit messages.

3. *Reproducible results with open-source implementation.* We provide a C++ implementation of our algorithm, built on top of the codebase from [GPVL23]. The code is available at https://github.com/thalespaiva/incomplete_ntt_amortized_bt.

Organization. Section 2 reviews the necessary mathematical background and theory relevant to our work, including the concept of incomplete NTT and amortized bootstrapping. Section 3 gives an overview of the amortized bootstrapping strategy proposed in [GPVL23], discussing their two-part inverse NTT along with security considerations and implementation aspects. Section 4 details our bootstrapping algorithm, including our alternative two-part inverse incomplete NTT. Section 5 describes our implementation and experimental results, which are compared with state-of-the-art sequential and amortized approaches. Finally, Section 6 presents our conclusions and ideas for future work.

2 Preliminaries

Throughout this paper, we write polynomials $\boldsymbol{x} = \sum x_i X^i$ in ***italic bold*** to differentiate them from vectors $\mathbf{v}$ that are written in **bold**. For a polynomial $\boldsymbol{x}$ we use x_i when referring to its i^{th} coefficient, or the equivalent notation $\boldsymbol{x}[i]$ when doing so in a pseudo-code description. For matrices, we denote as $\mathbf{X}[i]$ their i^{th} row. We use lg to denote the logarithm in base 2.

2.1 Rings and cryptographic schemes that use them

2.1.1 Rings

The amortized boostrapping algorithm given in [GPVL23], which is the basis for our work, operates over a circulant ring of the form $\mathbb{Z}[X]/(X^p - 1)$ for some prime p. Using the same notation as [GPVL23], we refer to this ring as $\tilde{\mathcal{R}}$. When the coefficients of the elements of $\tilde{\mathcal{R}}$ are taken modulo Q for any positive integer Q, we write $\tilde{\mathcal{R}}_Q = \mathbb{Z}_Q[X]/(X^p - 1)$. Besides using $\tilde{\mathcal{R}}$, the same algorithm also relies on power-of-two cyclotomic rings of the form $\mathbb{Z}[X]/(X^N + 1)$ where $N = 2^k$ for some $k \in \mathbb{N}$. We denote this second ring $\hat{\mathcal{R}}$, or $\hat{\mathcal{R}}_Q$ if taken modulo Q.

2.1.2 Standard encryption schemes

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 the circulant version of LWE and GSW, which are used along the discussion.

LWE. Consider positive integers n and q. Let $\mathbf{s} \leftarrow \chi$ be a secret key sampled from a distribution χ, $m \in \mathbb{Z}$ an arbitrary message, and $\Delta \in \mathbb{Z}$ a scaling factor. The LWE encryption of message m under the secret key $\mathbf{s}$ is given by

$$\mathrm{LWE}_{q,\mathbf{s}}(\Delta \cdot m) = \left[\mathbf{a}^\top, b\right] \in \mathbb{Z}_q^{1\times(n+1)},$$

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

Ring LWE. The ring version of LWE considers the ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(\Phi_m(X))$, where Φ_m is the m^{th} cyclotomic ring. Let $\boldsymbol{sk} \leftarrow \chi$ be a secret key sampled from a distribution χ, $\boldsymbol{m} \in \mathcal{R}_q$ a message and $\Delta \in \mathbb{N}$ a scaling factor. The RingLWE encryption of the message $\boldsymbol{m}$ under the secret key $\boldsymbol{sk}$ is given by

$$\text{RingLWE}_{q,\boldsymbol{sk}}(\Delta \cdot \boldsymbol{m}) = [\boldsymbol{a}, \boldsymbol{b}] \in \mathcal{R}_q^{1\times 2},$$

where $\boldsymbol{a} \leftarrow \mathcal{R}_q$, $\boldsymbol{b} = -\boldsymbol{a} \cdot \boldsymbol{sk} + \boldsymbol{e} + \Delta \cdot \boldsymbol{m}$ and the error $\boldsymbol{e}$ is such that each of its coefficients $e_i \leftarrow \chi'$ is sampled from distribution χ'. In order to decrypt a RingLWE ciphertext, one computes the quantity $\boldsymbol{b} + \boldsymbol{a} \cdot \boldsymbol{sk} = \boldsymbol{e} + \Delta \cdot \boldsymbol{m}$ and performs a rounding operation to recover the message $\boldsymbol{m}$. Most schemes, including the bootstrapping algorithms considered in this work, consider $\Phi_m(X) = X^N + 1$ for $N = 2^k$ and any $k \in \mathbb{N}$, so the RingLWE problem is built with $\hat{\mathcal{R}}$.

Circulant LWE. As aforementioned, the algorithm introduced in [GPVL23] relies on circulant rings and, thus, considers the hardness of a variant of LWE called circulant-LWE (CLWE). This variant was introduced in [BDF18] and proven to be as hard as the Ring LWE variant on prime-order cyclotomic polynomials. It essentially considers the ring $\tilde{\mathcal{R}}$ instead of $\hat{\mathcal{R}}$, while sampling is done by picking RingLWE samples and projecting them onto $\tilde{\mathcal{R}}$.

GSW. Consider a modulus q and a dimension parameter N. The GSW scheme encrypts a message $\mathbf{m}$ when $\mathbf{C} \cdot \mathbf{v} = \mathbf{m} \cdot \mathbf{v} + \mathbf{e}$, where the ciphertext $\mathbf{C}$ is an $N \times N$ matrix over $\mathbb{Z}_q$, the secret key $\mathbf{v}$ is an N-dimensional vector over $\mathbb{Z}_q$ and $\mathbf{e}$ is a small error vector [GSW13]. In [GPVL23], the authors consider a double-CRT version of GSW, motivated by the fact that the modulus employed is quite large. More precisely, the modulus Q is decomposed into a product of smaller primes q_i, and CRT is used to operate independently modulo each q_i. Moreover, the polynomials in the ciphertexts are all represented in the FFT domain, whence the name double-CRT or RNS representation (see [GPVL23, Section 2.5] for more details).

2.2 Standard operations

Modulus switching. Modulus switching is a standard operation that changes the modulus of a ciphertext. For RingLWE ciphertext, the operation takes $(\boldsymbol{a}, \boldsymbol{b}) = \text{RingLWE}_{q,\boldsymbol{sk}}(\Delta \cdot \boldsymbol{m})$ and outputs a ciphertext $(\boldsymbol{a}', \boldsymbol{b}') = \text{RingLWE}_{q',\boldsymbol{sk}}(\Delta \cdot \boldsymbol{m})$ for a new (usually smaller) modulus q'. We refer to [GPVL23, Algorithm 2] for the pseudo-code of the modulus switching algorithm. In their bootstrapping procedure, the modulus switching operation is executed on RingLWE ciphertexts only.

Key switching. Key switching is another standard operation that allows replacing the secret key $\boldsymbol{sk}$ for another key $\boldsymbol{zk}$. In the RingLWE case, the procedure takes as input a ciphertext $(\boldsymbol{a}, \boldsymbol{b}) = \text{RingLWE}_{q,\mathbf{sk}}(\Delta \cdot \boldsymbol{m})$ and a public key-switching key generated from the secret key $\boldsymbol{sk}$, and then outputs a ciphertext $(\boldsymbol{a}', \boldsymbol{b}') = \text{RingLWE}_{q,\boldsymbol{zk}}(\Delta \cdot \boldsymbol{m})$. The specific case of circulant rings defined modulo $X^p - 1$ can pose some security issues, though. Indeed, [BDF18] proves that the GSW scheme over such circulant rings is secure when only encryptions of powers of X are considered. However, in key switching, one must consider the encryption of the new secret key $\boldsymbol{zk}$, which is not a power of X. To address this, [GPVL23, Section 3.5] introduces two algorithms specifically for GSW key switching with a noise-memory tradeoff – we refer the reader to [GPVL23, Algorithms 5 and 6] for details.

Automorphisms. Ring automorphisms can be used to perform scalar multiplication with registers. Recall that an automorphism is a bijective map from a ring $\mathcal{R}$ to itself, such that, for a given $t \in \mathbb{Z}_q^*$, we have $\boldsymbol{a}(X) \mapsto \boldsymbol{a}(X^t)$. Both [LMK+23] and [DMKMS24] apply ring automorphisms on RingLWE ciphertexts. In [GPVL23, Section 3.6], though, automorphisms are defined over GSW ciphertexts. Consider the GSW ciphertext $\mathbf{C}$ (a matrix) defined over the ring $\tilde{\mathcal{R}}_Q$, which encrypts a given message $\boldsymbol{m}$ under a certain key $\boldsymbol{sk}$. Let $\boldsymbol{ak}_t$ be a switching key. An automorphism $\psi_t : \tilde{\mathcal{R}}_Q \to \tilde{\mathcal{R}}_Q$ can be applied to each row of $\mathbf{C}$ such that $\psi_t(\mathbf{C}) \in \tilde{\mathcal{R}}_Q\mathrm{GSW}_{\psi_t(\boldsymbol{sk})}(\psi_t(\boldsymbol{m}))$. Applying key switching with $\boldsymbol{ak}_t$ on $\psi_t(\mathbf{C})$ allows to recover a ciphertext $\mathbf{C}' \in \tilde{\mathcal{R}}_Q\mathrm{GSW}_{\boldsymbol{sk}}(\psi_t(\boldsymbol{m}))$.

2.3 Incomplete NTT

The number-theoretic transform (NTT) is a finite-field analogue of the fast Fourier transform (FFT) that can be used for swift polynomial multiplication under certain conditions. For negacyclic rings of the form $\mathbb{Z}_p[X]/(X^N + 1)$, NTT-based multiplication requires the existence of a $2N$-root of unity in $\mathbb{Z}_p$, which is guaranteed to exist when $p \equiv 1 \pmod{2N}$. Similar to the FFT, the NTT is efficiently computed using $O(N \log N)$ operations via an iterative procedure. Due to the strict requirements on the modulus p from using the NTT in $\mathbb{Z}_p[X]/(X^N + 1)$, though, many works consider the so-called ℓ-incomplete NTT formulation [HYJ+25, ABD+19, CHK+21, Lip21, ABC19, CP05, Moe76, DMKMS24]. This is a generalization of the NTT where, instead of running all the $\lg N$ iterations of the algorithm, one stops at the $\lg N - \ell$ iteration, for an integer $0 \leq \ell \leq \lg N$. This requires a $2N/2^\ell$-root of unity to exist in $\mathbb{Z}_p$, which relaxes the condition on the modulus to $p \equiv 1 \pmod{2N/2^\ell}$. Consequently, incompleteness enables a strictly wider set of compatible prime moduli, as formally proved in Theorem 3 of [HYJ+25], which also formalizes and systematizes the key properties and advantages of the approach.

Due to cyclotomic polynomial properties, when a $2N/2^\ell$-root of unity exists in $\mathbb{Z}_p$, the polynomial $X^N + 1$ can be factored as $X^N + 1 = (X^{2^\ell} - \psi_1) \dots (X^{2^\ell} - \psi_{N/2^\ell})$, where each ψ_i is a distinct power of the root of unity, efficiently computable. Given a polynomial $\boldsymbol{a} \in \mathbb{Z}_p[X]/(X^N + 1)$, the ℓ-incomplete NTT returns a sequence of $N/2^\ell$ polynomials: $\mathrm{NTT}(\boldsymbol{a}) = (\hat{\boldsymbol{a}}_1, \dots, \hat{\boldsymbol{a}}_{N/2^\ell})$, where each $\hat{\boldsymbol{a}}_i = \boldsymbol{a} \pmod{x^{2^\ell} - \psi_i}$. To compute the product $\boldsymbol{c} = \boldsymbol{ab}$ of two polynomials $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{Z}_p[X]/(X^N + 1)$ using the ℓ-incomplete NTT, we evaluate $\hat{\boldsymbol{a}} = \mathrm{NTT}(\boldsymbol{a})$ and $\hat{\boldsymbol{b}} = \mathrm{NTT}(\boldsymbol{b})$, then, compute the products $\hat{\boldsymbol{c}}_i = \hat{\boldsymbol{a}}_i \hat{\boldsymbol{b}}_i \pmod{x^{2^\ell} - \psi_i}$ for $i = 1, \dots, N/2^\ell$. The process of multiplying polynomials modulo $x^{2^\ell} - \psi_i$ for each ψ_i, which is a frequent operation in the NTT formulation, is commonly called the *base multiplication*. Finally, the product $\boldsymbol{c}$ is computed by taking the inverse incomplete NTT as $\boldsymbol{c} = \mathrm{INTT}(\hat{\boldsymbol{c}})$, where $\hat{\boldsymbol{c}} = (\hat{\boldsymbol{c}}_1, \dots, \hat{\boldsymbol{c}}_{N/2^\ell})$.

The ℓ-incomplete NTT is known to enable faster polynomial multiplication than the complete NTT (i.e., the case where $\ell = 0$) in some cases – typically, when using small values of ℓ such as $\ell = 1, 2$ [ABD+21, Kan22, HYJ+25]. However, in general, increasing ℓ beyond these values degrades performance. The reason is that the complexity of the base multiplication, typically executed using the schoolbook algorithm, grows exponentially in ℓ. This observation is also explicated by Hafiz et al. in [HYJ+25, Section 5].

3 Guimarães et al.'s [GPVL23] amortized boostrapping

The amortized bootstrapping algorithm presented in [GPVL23] follows the same structure as the original amortized bootstrapping algorithm introduced in [MS18], so it can also be decomposed into three main steps:

1. **Packing** takes as input N LWE ciphertexts, each encrypting a (scaled) κ-bit message $m_i \in \mathbb{Z}_{2^\kappa}$ under the same secret key $\mathbf{s}$. In other words, given a set $\{\mathrm{LWE}_{\mathbf{s}}^{p^\star}(\Delta \cdot m_i)\}_{i=1}^N$

for prime modulus $p^\star$ and a scaling factor $\Delta = \lfloor p^\star/2^\kappa \rceil$, a packing key consisting of RingLWE encryptions of coefficients of the LWE secret key, the algorithm outputs a single "packed" RingLWE ciphertext encrypting a (scaled) message $\boldsymbol{m}(X) = \sum_i m_i X^{i-1}$. The algorithm was originally introduced in [MS18, Section 3] and we refer the reader to [MS18, Lemma 4] for a proof of its correctness and the corresponding pseudocode. In [GPVL23], the packing step is not modified from the original work, and the authors report the pseudocode in Appendix E.

2. **Homomorphic decryption** is the core of the bootstrapping algorithm. Homomorphically decrypting the RingLWE ciphertext obtained from packing essentially consists of computing a homomorphic polynomial multiplication, as explained in Section 2.1.

3. **Message extraction** is the last step of amortized bootstrapping, where one recovers the N LWE ciphertexts with reduced noise from the bootstrapped ring ciphertext. This technique was first introduced in [DM15] and adapted to this algorithm (in particular to the cyclic ring setup) in [GPVL23, Algorithm 10].

Let us now focus on the second step, i.e., the homomorphic decryption. The output of the preceding packing step is a RingLWE ciphertext $(\boldsymbol{a}, \boldsymbol{b}) \in \hat{\mathcal{R}}_Q^2$. Recall from Section 2.1 that, in order to (homomorphically) decrypt such a ciphertext encrypted under some key $\boldsymbol{zk}$, one must compute the quantity $\boldsymbol{b} + \boldsymbol{a} \cdot \boldsymbol{zk}$ (with an encryption of $\boldsymbol{zk}$ given as bootstrapping key). For the polynomial multiplication, [GPVL23] replaces the Nussbaumer transform and the SlowMult algorithm from [MS18] by an NTT and a pointwise multiplication. However, for this substitution to work, the algorithm must rely on an accumulator that enables the evaluation of a standard NTT homomorphically. This includes being able to perform a variety of operations in the exponent, such as addition, subtraction, scalar multiplication, and key-switching.

The algorithm introduced in [GPVL23] primarily operates over the circulant ring $\tilde{\mathcal{R}}_Q = \mathbb{Z}_Q[X]/(X^p - 1)$ for some prime p and modulus Q. The registers on which the homomorphic operations are performed are RingGSW ciphertexts over this ring, encrypted with the secret key $\boldsymbol{sk}$. While homomorphic multiplications and external products are common operations already well-defined for (ring) GSW, [GPVL23] extends the use of automorphisms and key switching to the GSW scheme [GPVL23, Section 3.5, Algorithm 5] to enable the use of the homomorphic NTT. Consequently, as the inputs of the polynomial multiplication are elements $\boldsymbol{a}, \boldsymbol{zk}$, both in the negacyclic polynomial ring $\hat{\mathcal{R}}_Q$, the multiplication (and addition of $\boldsymbol{b} \in \hat{\mathcal{R}}_Q$) is done by computing

$$\mathrm{INTT}\left(\mathrm{NTT}(\boldsymbol{a}) \cdot \mathrm{NTT}(\boldsymbol{zk}) + \mathrm{NTT}(\boldsymbol{b})\right) \equiv \boldsymbol{a} \cdot \boldsymbol{zk} + \boldsymbol{b} \in \hat{\mathcal{R}}_Q.$$

The quantities $\mathrm{NTT}(\boldsymbol{a})$ and $\mathrm{NTT}(\boldsymbol{b})$ are first computed along with the bootstrapping keys, which correspond to RingGSW encryptions of $\mathrm{NTT}(\boldsymbol{zk})$. The pointwise multiplication is performed using automorphisms. Finally, the inverse NTT is applied homomorphically in order to recover the GSW ciphertexts encrypting $\boldsymbol{b} + \boldsymbol{a} \cdot \boldsymbol{zk}$. The correctness of the amortized bootstrapping algorithm is given in [GPVL23, Theorem 1] and the noise growth is reported in [GPVL23, Corollary 2]. We report in Table 1 the parameters used in both [GPVL23] and our proposed algorithm.

In what follows, we describe the inverse NTT algorithm used by [GPVL23] in more details, while adapting the presentation to better suit comparisons with our own work.

3.1 Two-part inverse NTT for controlling error growth

Consider the most costly operation when bootstrapping a ciphertext $(\boldsymbol{a}, \boldsymbol{b}) \in \hat{\mathcal{R}}_Q$ encrypted with a secret key $\boldsymbol{zk}$: the homomorphic multiplication of the polynomials $\boldsymbol{a}$ and $\boldsymbol{zk}$ given

Table 1. Summary of the parameters used in [GPVL23] and in our proposal.

Type	Parameter	Description
Message	κ	Bitsize of the messages
Degree	N	Power-of-two degree of RLWE samples
	p	Prime degree of circulant-LWE samples
Modulus	Q	Ciphertext modulus
	$p^\star$	Plaintext modulus
Rings	$\hat{\mathcal{R}} = \mathbb{Z}[X]/(X^N+1)$	Power-of-two cyclotomic ring
	$\hat{\mathcal{R}}_Q = \mathbb{Z}_Q[X]/(X^N+1)$	Power-of-two cyclotomic ring with coeffients modulo Q
	$\tilde{\mathcal{R}} = \mathbb{Z}[X]/(X^p-1)$	Circulant ring
	$\tilde{\mathcal{R}}_Q = \mathbb{Z}_Q[X]/(X^p-1)$	Circulant ring with coeffients modulo Q
Secret Key	$\mathbf{s}$	LWE secret key
	$\boldsymbol{sk}$	RingGSW secret key in $\tilde{\mathcal{R}}_Q$.
	$\boldsymbol{zk}$	RingLWE secret key in $\hat{\mathcal{R}}_Q$
NTT-related	m	Radix of NTT
	ℓ	NTT incompleteness degree
	α	Balancing parameter for the two-part NTT

an encryption of the coefficients of $\boldsymbol{zk}$. Usually, multiplications in the cyclotomic ring $\hat{\mathcal{R}}_Q = \mathbb{Z}_Q[X]/\left(X^N+1\right)$ can be done very efficiently in $O(N \lg N)$ using the fast INTT. However, this approach can be problematic in an homomorphic setting due to the resulting error growth. Specifically, at each iteration i, for $i = 1$ to $\lg N$, the fast INTT algorithm performs $N/2$ multiplications and N additions. While this would require only $(N/2)\lg N$ multiplications for cleartext operations, in the homomorphic case each INTT iteration requires scalar multiplications on ciphertexts, thus increasing the depth of multiplications. The resulting ciphertexts have a depth of $\lg N$ multiplications, leading to an error growth that is likely to cause decryption failures except for very small values of N.

Let us compare this to a naive INTT implementation using a matrix-vector multiplication. The inverse NTT is a linear operation that can be described with a matrix $\mathbf{M} \in \mathbb{Z}_p^{N\times N}$. With this formulation, the INTT computation takes N^2 multiplications. In this case, all homomorphic multiplications are done in depth 1, and then combined with additions, making it much more manageable with respect to ciphertext error growth. However, on the downside, the large number of multiplications required can make it impractical for larger values of N.

The strategy adopted by Guimarães et al. [GPVL23] is to find a good balance between performance and depth of homomorphic multiplications (with respect to error growth). In particular, they use a two-part INTT computation based on a radix-m formulation of the INTT, which limits the homomorphic multiplication depth to only two levels. Algorithm 1 shows their INTT algorithm without considering homomorphic operations. It takes an input[1] $0 \leq \alpha \leq \lg N$ that defines the radix $m = 2^\alpha$ used for the computation using $N^2/m + Nm = N^2/2^\alpha + N2^\alpha$ operations. After a simple optimization argument, the authors conclude that using $\alpha \approx \lg N/2$, the value that best balances Part 1 and Part 2 in Algorithm 1, is the theoretically optimal choice.

In practice, however, we argue that there are cases where another choice of α may yield better performance. This can be explained by making the following observations. In efficient FHE implementations, it is not uncommon to merge steps between different consecutive procedures to improve performance and error growth. This is done in Guimarães et al.'s implementation, for example, as they partially merge the homomorphic inverse NTT

[1]The algorithm by Guimarães et al. [GPVL23] takes parameter m directly. We introduce the auxiliary variable $\alpha = \lg m$ for easier comparison with our proposed algorithm (described in Section 4.1).

procedure `INTT_gpvl`($\hat{\mathbf{a}}$, α)
- $\hat{\mathbf{a}} \in \mathbb{Z}_p^N$ is the vector of coefficients to which the INTT is applied, i.e., $\hat{\mathbf{a}} = \text{NTT}(\boldsymbol{a})$
- $\alpha \in \mathbb{Z}$ is the balancing parameter between parts 1 and 2

$\omega \leftarrow$ a primitive $2N$-th root of unity of p
$m = 2^\alpha$ ▷ Defines the radix-m INTT computation
▷ Part 1: $O\left(m(N/m)^2\right) = O\left(N^2/2^\alpha\right)$
for $k = 0$ to $m - 1$ **do**
  $\mathbf{g} \leftarrow (\hat{\mathbf{a}}[k], \hat{\mathbf{a}}[k+m], \ldots, \hat{\mathbf{a}}[N-2m], \hat{\mathbf{a}}[N-m])$
  $\mathbf{h}_k \leftarrow \mathbf{0} \in \mathbb{Z}_p^{N/m}$
  for $j = 0$ to $N/m - 1$ **do**
    $\mathbf{h}_k[j] \leftarrow \sum_{i=0}^{N/m-1} \mathbf{g}[i]\omega^{-2ij(N/m)} \mod p$
▷ Part 2: $O\left(m^2(N/m)\right) = O(N2^\alpha)$
for $k_1 = 0$ to $N/m - 1$ **do**
  for $k_2 = 0$ to $m - 1$ **do**
    $j \leftarrow k_1 + k_2(N/m)$
    $\mathbf{a}[j] \leftarrow \frac{1}{N} \sum_{i=0}^{m-1} \mathbf{h}_i[k_1]\omega^{-j-2i(k_1-k_2(N/m))} \mod p$
▷ Notice that the output is in natural, not bit-reversed, order
return $\mathbf{a} \in \mathbb{Z}_p^N$
end procedure

Algorithm 1. Two-part inverse NTT computation used by Guimarães et al. [GPVL23].

with the message extraction step [GPVL23, Algorithm 10 (see Remark 1)]. Moreover, to optimize performance, one may prefer to use a more efficient (but with higher noise growth) scalar multiplication or a slower (but more precise) multiplication in different parts of the code. This can also be seen in Guimarães et al.'s implementation, where either scalar multiplications with RGSW or RLWE registers are considered in each part of Algorithm 1.

Such practical considerations may result in some imbalance in the performance of the two parts of Algorithm 1, making the best choice of α diverge from the theoretical optimum. This leads to different ways of computing sums of products in Part 1 and Part 2 of Algorithm 1 and, to the best of our knowledge, explains why they use $\alpha = 6$ (i.e. $m = 64$) for $N = 1024$ in their code, whereas the theoretically best value of α would be $\alpha = 5$ if INTT was individually considered.

3.2 Secure and reliable instantiations

The security of Guimarães et al.'s algorithm [GPVL23] is based on the hardness of LWE under two different settings: one related to the LWE problem, used for encryption of the original messages, and another for the RingLWE problem, used to define the GSW-based accumulator. These are explicitly described as follows.

- LWE_1: For a prime modulus $p^\star$ and a power-of-two N, the first LWE setting is defined over the vector space $\mathbb{Z}_{p^\star}^N$. The secret key $\mathbf{s} \in \mathbb{Z}_{p^\star}^N$ and the error vector $\mathbf{e} \in \mathbb{Z}_{p^\star}^N$ are sampled according to distributions χ and χ', respectively, where χ is a balanced ternary distribution[2] of even Hamming weight w, and χ' is a discrete Gaussian with standard deviation of 1.

[2] A balanced ternary distribution of even Hamming weight w and length N samples vectors in $\{-1, 0, 1\}^N$ whose numbers of positive and negative coordinates are both equal to $w/2$.

- LWE_2: The second setting is a RingLWE defined over the circulant polynomial ring $\tilde{\mathcal{R}}_Q = \mathbb{Z}_Q[X]/(X^p - 1)$. The secret key $\boldsymbol{sk} \in \tilde{\mathcal{R}}_Q$ and the error polynomial $\boldsymbol{e} \in \tilde{\mathcal{R}}_Q$ are sampled according to discrete Gaussians of standard deviations of 3.2 and 1, respectively.

While parameters $(N, p^\star, w, p, Q)$ used above define the security of the scheme, practical values must ensure that the decryption failure rate (DFR) is sufficiently small. Naturally, the DFR also depends on the message space $\mathbb{Z}_{2^\kappa}$, which affects the scaling factor used when encoding coefficients of κ-bit messages into $\mathbb{Z}_{p^\star}$. Guimarães et al. [GPVL23] show that the DFR can be upper bounded by

$$\Pr\left(\text{Decryption of } \kappa\text{-bit message fails}\right) \leq 1 - \operatorname{erf}\left(\frac{p/2^{\kappa+1}}{\sqrt{2\varepsilon}}\right), \tag{1}$$

where erf is the Gaussian error function and $\varepsilon = \left(p + N^2/2\right)(\lg p^\star)(p/p^\star)^2 + w/12$.

Notice that Guimarães et al. [GPVL23] provide a generic formulation of the upper bound that takes into account different parameter formulations (mainly standard deviations for LWE noise that are later set to 1). Then, to select parameters achieving 128-bit security and low DFR, they rely on the LWE estimator [APS15] and use a manual search. Conversely, we choose the simpler and more concrete presentation above that both matches their results and is sufficient for our case. We report their parameter choices and performance in Section 5, providing comparisons with our own results in Table 2.

3.3 Acceleration using the Intel HEXL library

The code provided in [GPVL23] uses the Intel HEXL library [BKS+21] to accelerate some mathematical computations. Although HEXL comes with efficient vectorized implementations for functions related to number theory and multiplications in polynomial rings, they make some necessary adjustments in their code to be able to use HEXL for their purposes. For example, some of the most costly operations during bootstrapping take place inside the ring $\tilde{\mathcal{R}} = \mathbb{Z}[X]/(X^p - 1)$ for a prime p, but HEXL is optimized for fast NTT-based multiplication over rings of the form $\tilde{\mathcal{R}}' = \mathbb{Z}[X]/(X^{N'} - 1)$, where N' is a power of two.

In order to use HEXL for operations in $\tilde{\mathcal{R}}$, one may notice that a product $\boldsymbol{a} \cdot \boldsymbol{b}$ of polynomials $\boldsymbol{a}, \boldsymbol{b} \in \tilde{\mathcal{R}}$ has degree at most $2p - 2$. Hence, by considering N' as the first power of two larger than $2p - 2$, and representing the polynomials $\boldsymbol{a}$ and $\boldsymbol{b}$ as elements in $\tilde{\mathcal{R}}'$, the resulting product in $\tilde{\mathcal{R}}'$ can be efficiently computed in HEXL. At the same time, since N' is larger than the maximum degree of the product, the result is not affected by the modulo $(X^{N'} - 1)$. Therefore, by manually reducing the result by $(X^p - 1)$, one can obtain the desired product $\boldsymbol{a} \cdot \boldsymbol{b} \in \tilde{\mathcal{R}}$.

The drawback of this approach, however, is that memory usage and element-wise operations take roughly $2\times$ more when operating in $\tilde{\mathcal{R}}'$ instead of $\tilde{\mathcal{R}}$ directly. For this reason, [GPVL23] suggests choosing p as close to the next power of two as possible, which can be explained for two reasons. First, the bootstrapping performance is very similar for values of p with the same number of bits. Second, the LWE hardness and DFR are better when p is larger.

4 Our proposal: Amortized bootstrapping using the incomplete NTT

In this section, we describe how to use the incomplete NTT in the context of Guimarães et al.'s amortized bootstrapping algorithm. First, we show how to define a two-part incomplete NTT whose error growth is exactly the same as that obtained by Guimarães

et al. [GPVL23]. We then show how the base multiplication can also be integrated into the formulation without impacting the error growth. Finally, we present a surprising property of our construction: it incurs no performance penalty when using moderate levels of incompleteness, which we refer to as *incompleteness for free*. We then build upon the added flexibility provided by incomplete NTTs to obtain faster implementations of the amortized bootstrapping algorithm in Section 5.

4.1 An incomplete formulation of the two-part inverse NTT

The main idea behind our proposal is to use a similar approach to the two-part inverse NTT used by Guimarães et al., but generalizing it to enable the use of the radix-2 incomplete NTT formulation. One challenge is that the twiddle factors are not easily defined by a closed expression in this case, because there is no guarantee that a primitive $2N$-th root of unity exists in $\mathbb{Z}_p$.

A simple way to compute the twiddle factors is by considering the factorization of the inverse NTT into matrices, analogously to how Gentleman and Sande [GS66] considered for the FFT. Let $\mathbf{M} \in \mathbb{Z}_p^{N \times N}$ be the matrix associated with the inverse NTT operation over bit-reversed vectors. Then, the well-known fast inverse incomplete NTT [LS19, Kan22, HYJ+25] yields a factorization of $\mathbf{M}$ into

$$\mathbf{M} = 2^{-(\lg N - \ell)} \mathbf{M}_1 \dots \mathbf{M}_{(\lg N - \ell)},$$

where ℓ is the incompleteness level, and each $\mathbf{M}_i$ is a sparse matrix often referred to as a butterfly matrix. Figure 1 shows the overall shape of the butterfly matrices for the particular case when $N = 16$ and $\ell = 1$.

Let α be an integer such that $0 \leq \alpha \leq \lg N - \ell$. We can factor $\mathbf{M}$ into two matrices as $\mathbf{M} = \mathbf{M}_\mathrm{B}\mathbf{M}_\mathrm{A}$, where $\mathbf{M}_\mathrm{B} = 2^{-(\lg N - \ell)} \mathbf{M}_1 \dots \mathbf{M}_\alpha$ and $\mathbf{M}_\mathrm{A} = \mathbf{M}_{\alpha+1} \dots \mathbf{M}_{(\lg N - \ell)}$. Since $\mathbf{M}_\mathrm{A}$ and $\mathbf{M}_\mathrm{B}$ are products of sparse butterfly matrices, they are somewhat sparse: each row of $\mathbf{M}_\mathrm{B}$ has 2^α nonzero coefficients and $\mathbf{M}_\mathrm{A}$ is a block diagonal matrix in which each block is an $(N/2^\alpha) \times (N/2^\alpha)$ matrix. We can also define an auxiliary block diagonal matrix $\mathbf{A}$ as

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & \cdots & \mathbf{0} \\ \vdots & \ddots & \vdots \\ \mathbf{0} & \cdots & \mathbf{A}_{(N/2^\ell)} \end{bmatrix} \in \mathbb{Z}_p^N.$$

Figure 2 shows the sparsity of $\mathbf{M}_\mathrm{A}$, $\mathbf{M}_\mathrm{B}$ and $\mathbf{A}$ for $N = 16$, $\ell = 1$ and $\alpha = 2$.

Algorithm 2 shows how the sparsity of the matrices $\mathbf{M}_\mathrm{A}$ and $\mathbf{M}_\mathrm{B}$ can be leveraged to compute the incomplete NTT in two steps: the multiplications required by both parts

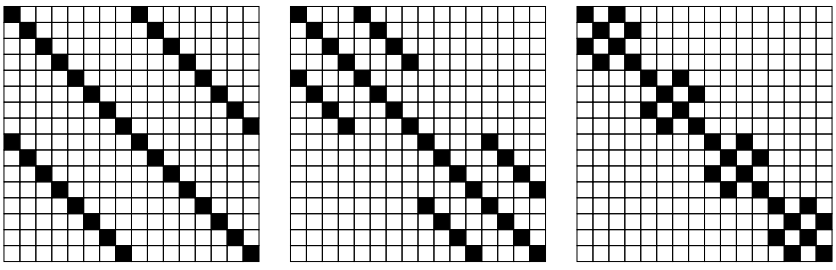

Figure 1. The shape of the butterfly matrices that decompose the inverse NTT for $N = 16$ and $\ell = 1$. White and black squares represent the positions of null and non-null values, respectively.

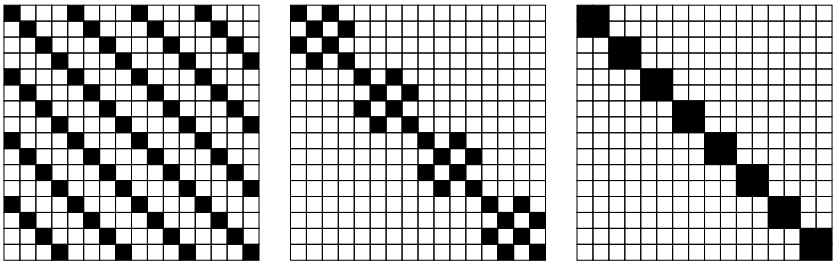

Figure 2. Shapes of the matrices $\mathbf{M}_\mathrm{A}$, $\mathbf{M}_\mathrm{B}$ and $\mathbf{A}$ for $N = 16$, $\ell = 1$ and $\alpha = 2$. White and black squares represent the positions of null and non-null values, respectively.

1 and part 2 are implemented by discarding the null entries of matrices $\mathbf{M}_\mathrm{A}$ and $\mathbf{M}_\mathrm{B}$, respectively. Notice that, since N and p are part of the scheme parameters, we can easily pre-compute matrices $\mathbf{M}_\mathrm{A}$ and $\mathbf{M}_\mathrm{B}$ for the targeted incompleteness level ℓ. With these precomputed matrices, Algorithm 2 uses $O\big(N2^\alpha + N^2/2^\alpha\big)$ operations and, hence, has the same complexity as Algorithm 1.

Generalizing to three-part inverse NTT or more. We remark that it is possible to split the inverse NTT into more than only two parts, simply by grouping more products of butterfly matrices that factor $\mathbf{M}$. This is analogous to using a larger number of radix-m

procedure `INTT_incomplete`($\hat{\mathbf{a}}$, ℓ, α)
- $\hat{\mathbf{a}} \in \mathbb{Z}_p^N$ is the vector of coefficients to which the INTT is applied, i.e., $\hat{\mathbf{a}} = \mathrm{NTT}(\boldsymbol{a})$
- $\ell \in \mathbb{Z}$ is the incompleteness level (compatible with N and p)
- $\alpha \in \mathbb{Z}$ is the balancing parameter between parts 1 and 2

▷ Build matrices $\mathbf{M}_\mathrm{A}$ and $\mathbf{M}_\mathrm{B}$ (can be precomputed for fixed (N, p, ℓ, α))
$\mathbf{M} \leftarrow$ Matrix associated with the ℓ-incomplete INTT of length N in $\mathbb{Z}_p$
$\mathbf{M}_1, \ldots, \mathbf{M}_{(\lg N-\ell)} \leftarrow$ Butterfly matrices that factor $\mathbf{M}$
$\mathbf{M}_\mathrm{A} \leftarrow \mathbf{M}_{\alpha+1} \ldots \mathbf{M}_{(\lg N-\ell)}$
$\mathbf{M}_\mathrm{B} \leftarrow 2^{-(\lg N-\ell)}\mathbf{M}_1 \ldots \mathbf{M}_\alpha$
$m \leftarrow 2^\alpha$
▷ Part 1: Compute $\mathbf{a}' = \mathbf{M}_\mathrm{A}\hat{\mathbf{a}}$ in $O\big(N^2/m\big) = O\big(N^2/2^\alpha\big)$
$\mathbf{a}' = \mathbf{0} \in \mathbb{Z}_p^N$
for $j = 0$ to $m - 1$ **do**
  for $k = 0$ to $N/m - 1$ **do**
    $\mathbf{a}'[jm + k] \leftarrow \sum_{i=0}^{N/m-1} \mathbf{M}_\mathrm{A}[jN/m + k][i] \cdot \hat{\mathbf{a}}[jN/m + i]$
▷ Part 2: Compute $\mathbf{a} = \mathbf{M}_\mathrm{B}\mathbf{a}'$ in $O(Nm) = O(N2^\alpha)$
$\mathbf{a} = \mathbf{0} \in \mathbb{Z}_p^N$
for $i = 0$ to $N - 1$ **do**
  for $k = 0$ to $m - 1$ **do**
    $j \leftarrow i + kN/m \bmod N$
    $\mathbf{a}[i] \leftarrow \mathbf{a}[i] + \mathbf{M}_\mathrm{B}[i][j] \cdot \mathbf{a}'[j]$
return $\mathbf{a} \in \mathbb{Z}_p^N$
end procedure

Algorithm 2. Two-part inverse ℓ-incomplete NTT.

splitting layers, which has been considered by Guimarães et al. [GPVL23]. Although this would make the inverse NTT computation more efficient[3], their conclusion also holds for our formulation: if we split into more than two parts, the error growth gets too high due to the increased homomorphic multiplication depth. Therefore, we only consider the two-part inverse NTT in our experiments.

4.2 Free incompleteness: embedding the base multiplication into Part 1

We know that the incomplete NTT provides some clear advantages compared to the complete formulation: it is faster to compute because of the reduced number of iterations, and enables the use of a strictly larger set of prime moduli. The only drawback is the increased complexity of the base multiplication, which involves higher-degree polynomials than the complete NTT. Although some lattice-based cryptographic schemes [ABD+21, Nat24] show that the incomplete NTT with $\ell = 1$ can be faster than the complete NTT, the optimal level of incompleteness can depend on the scheme's parameters and hardware characteristics [HYJ+25, Kan22]. Interestingly, when using the two-part INTT, the base multiplication can be embedded into the first part, i.e., the multiplication by $\mathbf{M}_{\mathrm{A}}$. As shown in what follows, this enables the use of the two-part INTT in a way that a small incompleteness level $\ell \geq 1$ can be adopted without any negative performance impact.

Embedding the base multiplication into Part 1. Suppose we want to compute the product $\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b}$ of polynomials $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{Z}_p[X]/\left(X^N + 1\right)$, and let us assume that p and N enable the use of an ℓ-incomplete NTT. Hence, the polynomial $\left(X^N + 1\right)$ factors as

$$\left(X^N + 1\right) = \prod_{i=1}^{N/2^\ell} \left(X^{2^\ell} - \psi_i\right) \mod p,$$

for a particular set of elements $\psi_1, \ldots, \psi_{(N/2^\ell)} \in \mathbb{Z}_p$. In the ℓ-incomplete NTT domain, each polynomial $\boldsymbol{a}$ and $\boldsymbol{b}$ is represented by a sequence $\{\hat{\boldsymbol{a}}_i\}$ and $\{\hat{\boldsymbol{b}}_i\}$, respectively, for $i = 1$ to $N/2^\ell$, where $\hat{\boldsymbol{a}}_i, \hat{\boldsymbol{b}}_i \in \mathbb{Z}_p[X]/\left(X^{2^\ell} - \psi_i\right)$. In this case, the base multiplications translate to the products $\hat{\boldsymbol{c}}_i = \hat{\boldsymbol{a}}_i \cdot \hat{\boldsymbol{b}}_i$ for all i. The product polynomial can then be computed using the inverse NTT as $\boldsymbol{c} = \mathrm{INTT}\left(\hat{\boldsymbol{c}}_1, \hat{\boldsymbol{c}}_2, \ldots, \hat{\boldsymbol{c}}_{N/2^\ell}\right)$.

Since we use the two-part inverse NTT based on matrix-vector multiplications in the amortized bootstrapping application, let us put the base multiplications in the same form. The base multiplication is typically done using a naive schoolbook multiplication algorithm, as it is often the best choice for small values of ℓ [HYJ+25, Kan22]. Notice that the schoolbook multiplication of polynomials $\hat{\boldsymbol{a}}_i, \hat{\boldsymbol{b}}_i \in \mathbb{Z}_p[X]/\left(X^{2^\ell} - \psi_i\right)$ can be easily represented in the matrix-vector form as follows. Let $\hat{\boldsymbol{c}}_i = \hat{\boldsymbol{a}}_i \cdot \hat{\boldsymbol{b}}_i$ and $\hat{\mathbf{a}}_i, \hat{\mathbf{b}}_i, \hat{\mathbf{c}}_i \in \mathbb{Z}_p^{2^\ell}$ be the vectors of coefficients of polynomials $\hat{\boldsymbol{a}}_i$, $\hat{\boldsymbol{b}}_i$, and $\hat{\boldsymbol{c}}_i$, respectively. Then, we have $\hat{\mathbf{c}}_i = \mathbf{A}_i \hat{\mathbf{b}}_i$, where

$$\mathbf{A}_i = \begin{bmatrix} \hat{\mathbf{a}}_i[0] & \hat{\mathbf{a}}_i[n-1]\psi_i & \cdots & \hat{\mathbf{a}}_i[1]\psi_i \\ \hat{\mathbf{a}}_i[1] & \hat{\mathbf{a}}_i[0] & \cdots & \hat{\mathbf{a}}_i[2]\psi_i \\ \vdots & \vdots & \ddots & \vdots \\ \hat{\mathbf{a}}_i[n-1] & \hat{\mathbf{a}}_i[n-2] & \cdots & \hat{\mathbf{a}}_i[0] \end{bmatrix} \in \mathbb{Z}_p^{2^\ell \times 2^\ell}.$$

If we choose parameter α in $0 \leq \alpha \leq \lg N - \ell$, we can define the 2-part INTT matrices $\mathbf{M}_{\mathrm{B}}$ and $\mathbf{M}_{\mathrm{A}}$ and use the auxiliary block diagonal matrix $\mathbf{A}$, described in Section 4.1 Specifically, if $\mathbf{c} \in \mathbb{Z}_p^N$ is the vector of coefficients of the product $\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b}$, we obtain the matrix description of the inverse NTT as $\mathbf{c} = \mathbf{M}_{\mathrm{B}} \mathbf{M}_{\mathrm{A}} \mathbf{A} \hat{\mathbf{b}}$.

[3] To understand why more splittings make the inverse NTT more efficient, remember that the fastest way to compute the INTT is to split it into $\lg N$ parts, each corresponding to one butterfly matrix.

Notice that $\mathbf{M}_{\mathrm{A}}\mathbf{A}$ is a product of two block diagonal matrices, where the square blocks in $\mathbf{M}_{\mathrm{A}}$ and $\mathbf{A}$ have size $(N/2^{\alpha})$ and 2^{ℓ}, respectively, as illustrated in Figure 2. Since N is a power of 2, both dimensions are also powers of 2, which means $\mathbf{M}_{\mathrm{A}}\mathbf{A}$ is also a block matrix with square blocks of size $\max\{N/2^{\alpha}, 2^{\ell}\}$. However, since $\alpha \leq \lg N - \ell$, then $\max\{N/2^{\alpha}, 2^{\ell}\} = N/2^{\alpha}$. Therefore, if $\mathbf{X} = \mathbf{M}_{\mathrm{A}}\mathbf{A}$, the computation of the matrix-vector product $\mathbf{X}y$ has exactly the same complexity as computing $\mathbf{M}_{\mathrm{A}}\mathbf{y}$, for an arbitrary vector $\mathbf{y}$. We then say that matrix $\mathbf{X} = \mathbf{M}_{\mathrm{A}}\mathbf{A}$ embeds the base multiplication into the first part of the two-part INTT.

Incompleteness for free. Suppose we use the complete NTT formulation for a prime p and a power-of-two N, and let us assume $\alpha^{\star}$ is the balancing factor that yields the best performance of the two-part INTT. Let p' be a prime for which we can use the ℓ-incomplete negacyclic NTT for polynomials in $\mathbb{Z}_{p'}[X]/\big(X^N + 1\big)$. Then, as long as $\ell \leq \lg N - \alpha^{\star}$, if we use the two-part INTT while embedding the base multiplication into Part 1, the operation count for these two procedures is the same:

1. Computing the pointwise product between two vectors in the NTT domain followed by the complete INTT for fast multiplication in $\mathbb{Z}_p[X]/\big(X^N + 1\big)$.
2. Computing the base multiplication of vectors in the ℓ-incomplete NTT domain followed by the incomplete INTT for fast multiplication in $\mathbb{Z}_{p'}[X]/\big(X^N + 1\big)$.

This observation is at the core of our contribution, since it enables us to leverage incompleteness to use smaller primes. The benefit is that homomorphic operations modulo smaller primes can be processed faster, with the peculiarity that the number of operations remains the same even when using those smaller primes.

4.3 Homomorphic polynomial multiplication using the incomplete NTT

We now give a full description of the homomorphic implementation of our multiplication algorithm. To do that, we use the homomorphic dot product presented in [GPVL23, Algorithm 7] as a building block. This function is denoted as `homomorphic_dot_product`. It uses the encryption $\mathbf{K}_{\boldsymbol{sk}}$ of the secret key $\boldsymbol{sk}$ (for the $\tilde{\mathcal{R}}_Q$GSW encryptions) and the set of key-switching keys $\{\mathbf{K}_v\}_{v=0}^{p-1}$, together with automorphisms $X \mapsto X^v$, to compute the dot product between a plaintext vector $\mathbf{u} \in \mathbb{Z}_p^k$ and an encrypted vector $\mathbf{m} \in \mathbb{Z}_p^k$. Concretely, if $\mathbf{m}, \mathbf{u} \in \mathbb{Z}_p^k$ and each $\mathbf{C}_i$ encrypts $X^{\mathbf{m}_i}$, then

$$\mathbf{C}' \leftarrow \texttt{homomorphic\_dot\_product}\Big(\{\mathbf{C}_i\}_{i=1}^{k}, \mathbf{u}, \mathbf{K}_{\mathbf{sk}}, \{\mathbf{K}_v\}_{v=0}^{p-1}\Big)$$

results in an encryption $\mathbf{C}'$ of $X^{\langle \mathbf{u}, \mathbf{m} \rangle}$.

Algorithm 3 shows the homomorphic implementation of the polynomial multiplication using the two-part ℓ-incomplete NTT. The algorithm homomorphically multiplies the polynomial $\boldsymbol{a} \in \hat{\mathcal{R}}_p$ by an encrypted polynomial $\boldsymbol{zk}$ as follows.

The core inputs of Algorithm 3 are the polynomial $\boldsymbol{a} \in \hat{\mathcal{R}}_p$ and the GSW encryptions $\mathbf{Z}_i$ of each coefficient $\hat{\mathbf{z}}_i$ of the NTT representation of the polynomial $\boldsymbol{zk} \in \hat{\mathcal{R}}_p$. It also receives the encryption $\mathbf{K}_{\boldsymbol{sk}}$ of the secret key $\boldsymbol{sk}$, and a set $\{\mathbf{K}_v\}$ of key-switching keys, used to perform the dot products. The last parameters are the incompleteness parameter ℓ and the balancing parameter α, which define the two-part incomplete NTT.

The algorithm begins by performing some non-homomorphic operations, which have negligible cost compared to the homomorphic ones. First, it computes the auxiliary matrices $\mathbf{M}_{\mathrm{A}}$ and $\mathbf{M}_{\mathrm{B}}$, as well as the base multiplication embedding matrix $\mathbf{X}$. Then, it performs two rounds of homomorphic dot products, the first for the multiplication by $\mathbf{X}$, and the second for the multiplication by $\mathbf{M}_{\mathrm{B}}$. Each element $\mathbf{C}_i$ from the output

procedure `homomorphic_polymul`$\Big(\boldsymbol{a}, \{\mathbf{Z}_i\}_{i=0}^{N-1}, \mathbf{K}_{\boldsymbol{sk}}, \{\mathbf{K}_v\}_{v=0}^{p-1}, \ell, \alpha\Big)$

- $\boldsymbol{a} \in \hat{\mathcal{R}}_p$
- $\{\mathbf{Z}_i\}_{i=0}^{N-1}$ are the bootstrapping keys, where each $\mathbf{Z}_i \in \tilde{\mathcal{R}}_Q\mathrm{GSW}_{\boldsymbol{sk}}\big(X^{-\hat{\mathbf{z}}[i]}\big)$
- $\mathbf{K}_{\boldsymbol{sk}} \in \tilde{\mathcal{R}}_Q\mathrm{GSW}_{\boldsymbol{sk}}(-\boldsymbol{sk})$
- Each $\mathbf{K}_v \in \tilde{\mathcal{R}}_Q\mathrm{KS}_{\boldsymbol{sk}}(\boldsymbol{sk}(X^v))$ is a key-switching key for the automorphism $X \mapsto X^v$
- $\ell \in \mathbb{Z}$ is the incompleteness level (compatible with N and p)
- $\alpha \in \mathbb{Z}$ is the balancing parameter between parts 1 and 2

▷ Build matrices $\mathbf{M}_\mathrm{A}$ and $\mathbf{M}_\mathrm{B}$ (can be precomputed for fixed (N, p, ℓ, α))
$\mathbf{M} \leftarrow$ Matrix associated with the ℓ-incomplete INTT of length N in $\mathbb{Z}_p$
$\mathbf{M}_1, \ldots, \mathbf{M}_{(\lg N-\ell)} \leftarrow$ Butterfly matrices that factor $\mathbf{M}$
$\mathbf{M}_\mathrm{A} \leftarrow \mathbf{M}_{\alpha+1} \ldots \mathbf{M}_{(\lg N-\ell)}$
$\mathbf{M}_\mathrm{B} \leftarrow 2^{-(\lg N-\ell)}\mathbf{M}_1 \ldots \mathbf{M}_\alpha$
▷ Build matrix $\mathbf{X}$ embedding the base multiplication by $\hat{\mathbf{a}}$ into $\mathbf{M}_\mathrm{A}$
$\hat{\mathbf{a}} \leftarrow \mathrm{NTT}(\mathbf{a}) \in \mathbb{Z}_p^N$
$\mathbf{A} \leftarrow$ block matrix representing the base multiplication by $\hat{\mathbf{a}}$
$\mathbf{X} \leftarrow \mathbf{M}_\mathrm{A}\mathbf{A}$
▷ No homomorphic operations yet: all code above has negligible performance impact
▷ Part 1: Homomorphic computation of $\mathbf{X}\hat{\mathbf{z}}$ in $O\big(N^2/2^\alpha\big)$
for $i = 0$ to $N-1$ **do**
  $\mathbf{C}'_i \leftarrow$ `homomorphic_dot_product`$\Big(\mathbf{Z}_i, \mathbf{X}[i], \mathbf{K}_{\mathbf{sk}}, \{\mathbf{K}_v\}_{v=0}^{p-1}\Big)$
▷ Part 2: Homomorphic computation of $\mathbf{M}_\mathrm{B}(\mathbf{X}\hat{\mathbf{z}})$ in $O(N2^\alpha)$
for $i = 0$ to $N-1$ **do**
  $\mathbf{C}_i \leftarrow$ `homomorphic_dot_product`$\Big(\{\mathbf{C}'_i\}_{i=0}^{N-1}, \mathbf{M}_\mathrm{B}[i], \mathbf{K}_{\mathbf{sk}}, \{\mathbf{K}_v\}_{v=0}^{p-1}\Big)$
return $\{\mathbf{C}_i\}_{i=0}^{N-1}$ ▷ Each $\mathbf{C}_i \in \tilde{\mathcal{R}}_Q\mathrm{GSW}_{sk}\big(X^{(a\cdot zk)[i]}\big)$
end procedure

Algorithm 3. Proposed homomorphic polynomial multiplication using the two-part incomplete INTT formulation. This function can be plugged into the partial decryption function defined in the GPVL algorithm [GPVL23, Algorithm 9].

$(\mathbf{C}_0, \ldots \mathbf{C}_{N-1})$ consists of the encryption of X^{c_i}, where c_i is the ith coefficient of the polynomial product $\boldsymbol{a} \cdot \boldsymbol{z}$.

4.4 The bootstrapping algorithm

Finally, we present our amortized bootstrapping algorithm in Algorithm 5. It closely follows the structure of [GPVL23, Algorithm 11], with the only difference occurring in the homomorphic decryption procedure. We refer the reader to [GPVL23, Algorithm 16] for `PackLWE` and [GPVL23, Algorithm 10] for `MsgExtract`, respectively, since, as mentioned in Section 3, these algorithms are somewhat standard and not the focus of either our work or [GPVL23]. The last procedure, `NTTDecModified`, is very similar to [GPVL23, Algorithm 9], but the homomorphic polynomial multiplication step is executed with our Algorithm 3. The pseudocode for it is given in Algorithm 4.

We now provide an analysis of the cost, error growth, and correctness of our algorithm. Naturally, is closely follows the results presented in [GPVL23].

Theorem 1 (Cost of the bootstrapping algorithm, from [GPVL23] Lemma 9)**.** *Let $Q = \prod_{i=1}^{u} q_i$ be the ciphertext modulus and d be the number of CRT digits used in the GSW ciphertexts. Given at most N LWE ciphertexts and the relevant bootstrapping keys,*

procedure `NTTDecModified`$\Big((\boldsymbol{a},\boldsymbol{b}),\{\mathbf{Z}_i\}_{i=0}^{N-1},\mathbf{K}_{\boldsymbol{sk}},\{\mathbf{K}_v\}_{v=0}^{p-1},\ell,\alpha\Big)$

- $(\boldsymbol{a},\boldsymbol{b})\in\hat{\mathcal{R}}_p^2$ is the packed ciphertext
- $\{\mathbf{Z}_i\}_{i=0}^{N-1}$ are the bootstrapping keys, where each $\mathbf{Z}_i\in\tilde{\mathcal{R}}_Q\mathrm{GSW}_{\boldsymbol{sk}}\big(X^{-\hat{\mathbf{z}}[i]}\big)$
- $\mathbf{K}_{\boldsymbol{sk}}\in\tilde{\mathcal{R}}_Q\mathrm{GSW}_{\boldsymbol{sk}}(-\boldsymbol{sk})$
- Each $\mathbf{K}_v\in\tilde{\mathcal{R}}_Q\mathrm{KS}_{\boldsymbol{sk}}(\boldsymbol{sk}(X^v))$ is a key-switching key for the automorphism $X\mapsto X^v$
- $\ell\in\mathbb{Z}$ is the incompleteness level (compatible with N and p)
- $\alpha\in\mathbb{Z}$ is the balancing parameter between parts 1 and 2

2: $\quad\{\mathbf{C}_i\}_{i=0}^{N-1}\leftarrow$ `homomorphic_polymul`$\Big(\boldsymbol{a},\{\mathbf{Z}_i\}_{i=0}^{N-1},\mathbf{K}_{\boldsymbol{sk}},\{\mathbf{K}_v\}_{v=0}^{p-1},\ell,\alpha\Big)$
3: $\quad$**for** $i=0$ to $N-1$ **do**
4: $\quad\quad\tilde{\mathbf{C}}_i\leftarrow X^{\boldsymbol{b}[i]}\cdot\mathbf{C}_i$
5: $\quad$**return** $\big(\tilde{\mathbf{C}}_0,\cdots,\tilde{\mathbf{C}}_{N-1}\big)$ ▷ Each $\tilde{\mathbf{C}}_i\in\tilde{\mathcal{R}}_Q\mathrm{GSW}_{sk}\big(X^{(a\cdot zk+b)[i]}\big)$
6: **end procedure**

Algorithm 4. Homomorphic partial decryption.

Algorithm 5 outputs LWE ciphertexts at the cost of $O(N^{1+1/\rho}\cdot\rho\cdot d^2\cdot u)$ NTTs and $O(N^{1+1/\rho}\cdot\rho\cdot d\cdot u^2\cdot p)$ multiplications in $\mathbb{Z}_{q_i}$ where $\rho=\lg N/\alpha$.

Proof. In [GPVL23], the radix m is set to $m=N^{1/\rho}$, where ρ is the recursive depth of the bootstrapping algorithm. In our work, we set $m=2^\alpha$ and, thus, $\rho=\lg N/\alpha$. The overall cost of Algorithm 5 is asymptotically dominated by the cost of `NTTDecModified` and, within this algorithm, of `homomorphic_polymul`. Both Algorithm 1 (the original two-part inverse NTT computation) and Algorithm 3 (`homomorphic_polymul`) have the same asymptotic cost, where Part 1 uses $O(N^2/2^\alpha)$ homomorphic operations and Part 2 uses $O(N2^\alpha)$ homomorphic operations. Thus `NTTDecModified` has the same complexity as `NTTDec`, and the overall complexity of the algorithm remains unchanged from [GPVL23]. □

Noise growth is another important factor of bootstrapping, as it directly influences the correctness of the algorithm. The only difference between Algorithm 5 and [GPVL23, Algorithm 11] is the `NTTDec` procedure or, more precisely, the homomorphic polynomial multiplication within it.

Theorem 2 (Noise growth of the boostrapping algorithm)**.** *Given at most N LWE ciphertexts and the relevant bootstrapping keys, Algorithm 5 outputs refreshed LWE ciphertexts with at most the same noise as given in [GPVL23, Lemma 9].*

Proof. Algo, given N input LWE ciphertexts with associated initial error, the resulting error of the packed ring ciphertext after applying `PackLWE` remains the same as in [GPVL23] as no modification is done to this initial step of the amortized bootstrapping algorithm (we refer to [GPVL23, Algorithm 16] for the exact expression of the associated error). Hence, to prove that a similar final error is expected, it suffices to show that the noise associated with the GSW ciphertexts $\mathbf{C}_i$ given by `NTTDec` is the same as the noise of the equivalent GSW ciphertexts provided by `NTTDecModified`. The final output noise after `MsgExtract` follows from [GPVL23, Lemma 11].

In Algorithm 4, `NTTDecModified`, the noise growth solely comes from the call to `homomorphic_polymul`, as the multiplication by $X^{\boldsymbol{b}[i]}$ modulo X^p-1 (line 4) only rotates the coefficients of the noise term.

We also know that both Algorithm 1 (the original two-part inverse NTT computation) and Algorithm 3 (`homomorphic_polymul`) follow a very similar structure, having the same number of homomorphic operations along with the same inputs. Thus, it is reasonable to assume that the noise growth resulting from Algorithm 1 and Algorithm 3 will be the same as given by [GPVL23, Lemma 7].

1: **procedure** $\texttt{Bootstrap}\Big((\mathbf{c}_0,\ldots,\mathbf{c}_N),(f_0,\ldots,f_N),\{\mathbf{Z}_i\}_{i=0}^{N-1},\mathbf{K}_{\boldsymbol{sk}},\{\mathbf{K}_v\}_{v=0}^{p-1},\ell,\alpha\Big)$

- Each $\mathbf{c}_i \in \mathrm{LWE}_s^{p^\star}(\Delta \cdot m_i) \in \mathbb{Z}_{p^\star}^N$ for $0 \le i < N$ and $\Delta = \lfloor p^\star/2^\kappa \rceil$
- Each $f_i : \mathbb{Z}_{2^\kappa} \to \mathbb{Z}_{2^\kappa}$ is the function that will be applied to the ith packed ciphertext
- $\{\mathbf{Z}_i\}_{i=0}^{N-1}$ are the bootstrapping keys, where each $\mathbf{Z}_i \in \tilde{\mathcal{R}}_Q\mathrm{GSW}_{\boldsymbol{sk}}\big(X^{-\hat{\mathbf{z}}[i]}\big)$
- $\mathbf{K}_{\boldsymbol{sk}} \in \tilde{\mathcal{R}}_Q\mathrm{GSW}_{\boldsymbol{sk}}(-\boldsymbol{sk})$
- Each $\mathbf{K}_v \in \tilde{\mathcal{R}}_Q\mathrm{KS}_{\boldsymbol{sk}}(\boldsymbol{sk}(X^v))$ is a key-switching key for the automorphism $X \mapsto X^v$
- $\ell \in \mathbb{Z}$ is the incompleteness level (compatible with N and p)
- $\alpha \in \mathbb{Z}$ is the balancing parameter between parts 1 and 2

2: $(\mathbf{a},\mathbf{b}) \leftarrow \texttt{PackLWE}(\mathbf{c}_0,\cdots,\mathbf{c}_{N-1})$
3: $(\mathbf{a}',\mathbf{b}') \leftarrow \texttt{ModSwitch}_{Q\to p}(\mathbf{a},\mathbf{b})$
4: $(\mathbf{C}_0,\cdots,\mathbf{C}_{N-1}) = \texttt{NTTDecModified}\Big((\mathbf{a}',\mathbf{b}'),\{\mathbf{Z}_i\}_{i=0}^{N-1},\mathbf{K}_{\boldsymbol{sk}},\{\mathbf{K}_v\}_{v=0}^{p-1},\ell,\alpha\Big)$
5: ▷ Each $\mathbf{C}_i \in \tilde{\mathcal{R}}_Q\mathrm{GSW}_{sk}\big(X^{e_i+\Delta\cdot m_i}\big)$ for small error e_i
6: **for** $i = 0$ to $N-1$ **do**
7: $\quad\mathbf{c}'_i \leftarrow \texttt{MsgExtract}(\mathbf{C}_i, f_i)$
8: **return** $(\mathbf{c}'_0,\cdots,\mathbf{c}'_{N-1})$
9: **end procedure**

Algorithm 5. Amortized bootstrapping algorithm for message space $\mathbb{Z}_{2^\kappa}$.

Finally, we note that `NTTDec` needs to account for noise coming from a key switching operation (line 6) after applying the GSW Galois automorphisms (line 5). Our algorithm `NTTDecModified` does not require this operation and, thus, does not need to account for the error growth resulting from it.[4]

□

Corollary 1 (From [GPVL23] Corollary 2)**.** *The bootstrapping algorithm presented in Algorithm 5 has noise overhead of $\tilde{O}(\lambda^{1.5+\rho})$, where λ is the security parameter and $\rho = \lg N/\alpha$.*

Proof. This follows from Theorem 2 and arguments given in [GPVL23, Corollary 2]. □

Theorem 3 (Correctness of bootstrapping, adapted from [GPVL23] Theorem 1)**.** *Let λ be a security parameter and $p^\star = \tilde{O}(\lambda^{2.5+\rho})$ for $\rho = \lg N/\alpha$. Consider the input ciphertexts $\boldsymbol{c}_i \in LWE_s^{p^\star}(\Delta \cdot m_i, E) \in \mathbb{Z}^{p+1}$ encrypting messages $m_i \in \mathbb{Z}_{2^\kappa}$ for $0 \le i < N$ with initial error satisfying $E = O(p^\star/\lambda)$ and $\Delta = \lfloor p^\star/2^\kappa \rceil$. Then, with probability $1 - 2^{-\lambda}$, the output of Algorithm 5 is correct, i.e., it outputs valid LWE encryptions of $f(m_i)$ for arbitrary functions $f : \mathbb{Z}_{2^\kappa} \to \mathbb{Z}_{2^\kappa}$.*

Proof. We refer to the proof of [GPVL23, Theorem 1] for the technical details. □

5 Experimental results

In this section, we present the experimental results of our work. We first recall the benefits of using an incomplete NTT and, in particular, illustrate the wide range of moduli that can be chosen as a function of the incompleteness degree ℓ. We then present our best timings and trade-offs between execution time and DFR, comparing our results with those given in [GPVL23] and with the state-of-the-art sequential algorithm TFHE-rs.

[4] We note that, in the original code, these extra key switching operations are avoided by multiplying the twiddle factors directly by the corresponding coefficients of $\hat{\mathbf{a}} = \mathrm{NTT}(\boldsymbol{a})$.

5.1 How incompleteness enables faster bootstrapping and lower DFR

We now discuss how incompleteness can help us obtain more efficient or reliable bootstrapping by reducing the constraints on p. Unfortunately, these two goals oppose each other: there are parameters that provide very low decryption failure rates but whose performance is impractical, and vice-versa.

Assume that, due to the requirements of some particular application, both the number N of LWE ciphertexts that are packed and the bit-length κ of the plaintext messages are fixed. In this scenario, the main parameter affecting the bootstrapping performance is the prime p, since most operations are done in the circulant ring $\tilde{\mathcal{R}} = \mathbb{Z}[X]/(X^p - 1)$. For higher performance, smaller values of p are better, as the reduced number of coefficients will entail less costly operations. Meanwhile, the most critical parameter for the DFR is also p, as shown in Equation 1 from Section 3.2.[5] In this case, however, reducing the DFR means that p should ideally be as large as possible.

Recall from Section 3.3 that, to accelerate operations in $\tilde{\mathcal{R}}$ using HEXL optimized implementations, [GPVL23] uses the auxiliary ring $\tilde{\mathcal{R}}' = \mathbb{Z}[X]/\left(X^{N'} - 1\right)$, where $N' = 2^{\lceil \lg p \rceil + 1}$. Therefore, to significantly improve the bootstrapping performance, it is not enough to simply reduce p. What is needed is to reduce its bit-length. This has an important consequence: when considering DFR and performance together, the optimal choices of p are close to, but below, powers of two. This ensures we are taking somewhat large values of p (to guarantee a reasonable DFR), without crossing over to the next power of two (which would cause a significant performance impact).

The benefit of the incomplete NTT for our amortized bootstrapping is that it allows us to select values that are close to powers of two. Figure 3 shows how a higher incompleteness level ℓ allows us to find parameters much closer to powers of two than the complete NTT ($\ell = 0$), considering the negacyclic NTT over $\mathbb{Z}_p[X]/\left(X^N + 1\right)$ for $N = 1024$. Our proposed technique leads to similar performance for all primes between the powers of two, represented by dotted vertical bars in Figure 3. Therefore, for any given bit-length, we have the flexibility to choose a prime that is the closest to the bars (i.e., allowing for a higher DFR) without impacting the performance. Concretely, for the range of interest between 13-bit and 15-bit primes, there are only two possible values of p when considering a complete NTT: 12289 (14-bit), which is used in [GPVL23], and 18433 (15-bit). In contrast, using the incomplete NTT with $\ell \leq 4$, significantly better parameters become available, such as 16001 and 32257, for the 14- and 15-bit ranges, respectively.

In the next section, we explore the possible tradeoffs between performance and DFR. In particular, we consider the different values of p made possible by the incomplete NTT and their interaction with other parameters, such as N, $p^\star$ or the Hamming weight of the secret key. This leads to significantly better results than those obtained in [GPVL23], which relies on the complete NTT.

5.2 Comparison with Guimarães et al.

We tested our algorithm with different sets of parameters: varying the ring $\hat{\mathcal{R}}$ size N, the prime moduli p and $p^\star$, and the Hamming weight w of the secret key. As incomplete NTT allows us to use a broader set of parameters, we begin by explaining our manual procedure to find reasonable parameter configurations.

Parameter selection. To ensure a parameter set offering 128-bit security, we use the Lattice Estimator from [APS15] considering the two independent LWE instances, defined

[5] We remark that $p^\star$ and w, appearing in the definition of ε used in Equation 1, also impact the DFR. Nonetheless, this impact is somewhat limited, since it is proportional to the square root of ε.

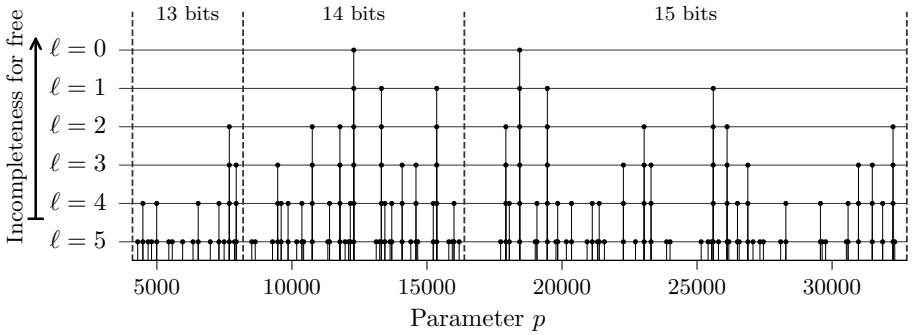

Figure 3. Possible prime values of $p \equiv 1 \bmod 2N/2^\ell$ that support an ℓ-incomplete NTT for fast multiplication in $\mathbb{Z}_p[X]/(X^N + 1)$, for $N = 1024$ and different ℓ. For $N = 1024$, the optimal balance parameter is $\alpha^\star = 6$, which means that incompleteness levels $\ell \leq \lg N - \alpha^\star = 4$ come for free in this case.

as LWE_1 and LWE_2 in Section 3.2.[6] For the modulo Q fixed as the product of three 49-bit primes, we found that $p \geq 7681$ is sufficient to ensure more than 128 bits of security considering LWE_2. However, parameters $p^\star$, N, w, and p appearing in LWE_1 also affect the DFR (see Equation 1), so it not trivial to find optimal values for them.

To find good parameter sets for a given incompleteness level ℓ, we use the following greedy manual approach, considering $N = 1024$ or 2048.

1. Choose the smallest $p^\star$ such that ε in the DFR expression (1) is sufficiently close to $w/12$, when considering the original parameters $(p, w) = (12289, 256)$. Intuitively, increasing $p^\star$ provides diminishing returns, so we want it to be just large enough to get the first term in ε to be much smaller than the second term.
2. Find the smallest w such that LWE_1 provides at least 128 bits of security.
3. Pick a value $p \geq 7681$ allowing ℓ-incompleteness for N, such that p is slightly smaller than some power-of-two.

Experimental setup and results. We ran all our experiments on the same machine, equipped with an Intel Xeon Platinum 8260 CPU at 2.4 GHz and 250 GB of RAM, running Ubuntu 20.04.6 LTS. We report results for single-threaded executions to ensure a fair comparison. However, our code is parallelized using OpenMP, so it can run roughly 13× faster using 24 cores. All the parameter sets considered achieve at least a 128-bit security level based on the LWE estimator from [APS15]. Our results are shown in Table 2, where the amortized execution time (in milliseconds) is the median observed over 10 experiments, ensuring a standard deviation of at most 1.3%. The DFR for κ-bit messages is computed using Equation 1. Notice that, according to the DFR equation, one can always reduce the DFR by choosing smaller values of κ,[7] but this may limit some applications that require messages of larger size. The first row of Table 2 lists the parameters and results obtained using [GPVL23]'s code with no modifications. The table also shows the smallest values of ℓ for which the prime p is compatible with N, i.e., $p \equiv 1 \bmod 2N/2^\ell$.

With the flexibility provided by NTT incompleteness, we are able to lower the modulus p to 7937 and improve the amortized time to 584 ms for $N = 1024$. This corresponds to

[6] Remember that LWE_1 and LWE_2 are related to the input LWE ciphertexts (before packing), and the resulting RLWE ciphertext (after packing), respectively.

[7] When a smaller number κ of bits is used for the message space and the LWE modulus $p^\star$ is fixed, the scaling factor $\Delta \approx p^\star/2^\kappa$ is larger. This means that a bigger portion of values modulo $p^\star$ is associated with each κ-bit possible message, thus making the final message recovery tolerate larger errors.

Table 2. Parameter sets satisfying a 128-bit security level and their corresponding performance-DFR tradeoffs. All sets use $\alpha = 6$ (the optimal value for the complete NTT), meaning that every incompleteness level $\ell \leq 4$ comes for free with $N = 1024$ and 2048.

Algorithm	N	p	$\lg(p^\star)$	w	ℓ	DFR for κ-bit messages			Amortized time (ms)	Speedup
						$\kappa = 7$	$\kappa = 8$	$\kappa = 9$		
[GPVL23]	1024	12289	24	256	0	$2^{-62.4}$	$2^{-17.3}$	$2^{-5.4}$	1,237.9	–
Ours	1024	7681	24	256	2	$2^{-30.0}$	$2^{-8.8}$	$2^{-3.0}$	585.6	2.11×
	1024	7937	24	256	3	$\mathbf{2^{-31.7}}$	$2^{-9.3}$	$2^{-3.1}$	**584.7**	**2.12**×
	1024	16001	24	256	4	$2^{-88.8}$	$2^{-24.1}$	$2^{-7.2}$	1,311.1	0.94×
	2048	7681	27	52	3	$2^{-147.8}$	$\mathbf{2^{-39.1}}$	$2^{-11.2}$	**879.8**	**1.41**×
	2048	7937	27	52	4	$2^{-157.2}$	$2^{-41.5}$	$2^{-11.8}$	882.7	1.40×
	2048	12289	27	52	0	$2^{-350.3}$	$2^{-90.2}$	$2^{-24.4}$	1,872.5	0.66×
	2048	15361	27	52	2	$2^{-516.3}$	$2^{-131.9}$	$\mathbf{2^{-35.0}}$	1,931.5	0.64×

a 2.12× speedup compared to [GPVL23]. However, in this case, our DFR is worse than [GPVL23] and as high as 2^{-9} for 8-bit messages. When considering 7-bit messages, though, the DFR goes down to $2^{-31.7}$. In order to improve upon the DFR for 8-bit messages while still boosting performance, our experiments also include a higher ring size of $N = 2048$. In this case, we are able to decrease the DFR significantly, to $2^{-39.1}$, while still improving on the amortized time for $p = 7681$. Specifically, we obtain an amortized time of 879 ms, which translates to a 1.41× speedup compared to [GPVL23]. We do not show the memory usage in Table 2 due to space limitations, but lower values of p also reduce RAM requirements.[8] For example, our parameter set with $(N, p) = (2048, 7681)$ requires about 50 GB of RAM, while their original parameters $(1024, 12289)$ uses 87 GB. Furthermore, although the parameter set for $N = 2048$ and $\ell = 0$ does not appear in [GPVL23], it was used with their original code for a fair comparison.

Overall, Table 2 showcases the aforementioned tradeoffs between the execution time of our algorithm and the observed DFRs: a smaller prime p leads to significantly better performance and higher DFR, while a lower p has the opposite effect. We identify the optimal values in bold. For both $N = 1024$ and $N = 2048$, choosing p larger than 12289, enables us to improve the DFR observed in [GPVL23] by 2^{-7} and 2^{-41}, respectively, with only a small impact on performance. Namely, by picking $p = 16001$ and $p = 15361$ for $N = 1024$ and $N = 2048$, respectively,the DFR values are 2^{-24} and 2^{-131}, while [GPVL23] obtains 2^{-17} for 8-bit messages; the performance, in turn, drops 0.94× and 0.64×, respectively. In the case of $N = 2048$ and $p = 15361$, we even obtain a better DFR than the 2^{-128} failure probability guaranteed by TFHE-rs v1.0 (see [BJSW24] for technical details). We also note that, for 9-bit messages, we achieve a DFR of at most 2^{-35} by choosing a larger prime $p = 15361$ and $N = 2048$.

5.3 Comparison with state-of-the-art

In addition to the improvement over [GPVL23], reported in Table 2, we compare our algorithm with its sequential counterpart for programmable bootstrapping, namely TFHE-rs [Zam22]. For that purpose, we use its latest version v1.0, from February 2025. In Table 3, we report the amortized cost in milliseconds. As before, the experiments were all run on the same machine with single-threaded executions.

For our algorithm, we highlight two main results. The first is our fastest execution using $N = 1024$, which incurs a high DFR (2^{-9}, as reported in Table 2 for 8-bit messages). The

[8] In performance-oriented implementations, like the original one by Guimarães et al. [GPVL23] and also ours, RAM usage is dominated by automorphism keys, which scale linearly with p. Guimarães et al. note that compression techniques can be used, although these come with performance impacts.

Table 3. Comparison with state-of-the-art implementations of sequential and amortized bootstrapping algorithms. For TFHE-rs, we run `pbs128-bench`.

Bootstrapping type	Algorithm	Source	LWE secret key type	Time per LWE ciphertext (ms)	Number of ciphertexts per bootstrapping
Sequential	TFHE PBS	TFHE-rs	Binary	660.68	1
Amortized	[GPVL23]	Code	Ternary	1,255.5	1024
	Ours	Code	Ternary	584.7	1024
	Ours	Code	Ternary	879.8	2048

second is our best running time for $N = 2048$, where the increased ring size allows us to lower the DFR significantly (to 2^{-39}, as reported in Table 2 for 8-bit messages) while still achieving better performance than [GPVL23]. For the sequential algorithm benchmarks, the only implementation for programmable bootstrapping is TFHE-rs, which provides a running time of 660ms. This is only 1.33× faster than our results with $N = 2048$ Our best performance, on the other hand, is 1.12× faster than TFHE-rs, as long as we can ignore the high DFR for 8-bit messages. For 7-bit messages, the speedup remains 1.12× faster than TFHE-rs and, in this case, the DFR stays at an acceptable level (namely, $2^{-31.7}$). We leave for future work the exploration of techniques to lower the DFR while preserving our best timings.

6 Conclusion and future work

In this work, we present an improved amortized bootstrapping algorithm for FHEW/TFHE ciphertexts, building on the algorithm introduced by Guimarães et al. [GPVL23]. While the asymptotic complexity of our algorithm remains the same as in [GPVL23] and [DMKMS24], we introduced algorithmic refinements that yield practical speedups. In particular, we build upon the flexibility provided by an incomplete NTT to expand the set of usable moduli, and then explore trade-offs between amortized runtime and decryption failure rates (DFR). Our alternative two-part incomplete NTT formulation embeds the base multiplication into the first part, allowing the use of incomplete NTT with no additional performance cost. As a result, we observe significant speedups compared to [GPVL23], as well as to the sequential TFHE-rs implementation, in different settings. We observe, however, that our best timings are obtained at the cost of a higher DFR. Reducing the DFR while maintaining comparable performance remains an open question, and presents a promising direction for future research.

Finally, we note that [GPVL23] relies on the HEXL library for optimization, which supports only power-of-two cyclotomic rings. Future improvements to amortized bootstrapping may benefit from extending support to arbitrary cyclotomics in widely used FHE libraries such as OpenFHE. Our work, aligned with recent advancements, further highlights the potential of amortized bootstrapping algorithms to outperform sequential counterparts or, at the very least, serves as a viable alternative in advancing the practicality of FHE.

Acknowledgements

We thank the authors of [GPVL23] for addressing our questions regarding their work, and in particular, the first author, Antonio Guimarães, for clarifying specific details in both the paper and the accompanying code.

References

[ABC19] Erdem Alkım, Yusuf Bilgin, and Murat Cenk. Compact and simple RLWE based key encapsulation mechanism. In *Progress in Cryptology–LATINCRYPT 2019*, pages 237–256. Springer, 2019.

[ABD+19] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber algorithm specifications and supporting documentation. *NIST PQC Round*, 2(4):1–43, 2019.

[ABD+21] Roberto Avanzi, Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: Algorithm specifications and supporting documentation (version 3.02), 2021. https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[BDF18] Guillaume Bonnoron, Léo Ducas, and Max Fillinger. Large FHE gates from tensored homomorphic accumulator. In *Progress in Cryptology–AFRICACRYPT 2018: 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7–9, 2018, Proceedings 10*, pages 217–251. Springer, 2018.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[BJSW24] Olivier Bernard, Marc Joye, Nigel P Smart, and Michael Walter. Drifting towards better error probabilities in fully homomorphic encryption schemes. *Cryptology ePrint Archive*, 2024.

[BKS+21] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D. M. de Souza, and Vinodh Gopal. Intel HEXL: Accelerating homomorphic encryption with Intel AVX512-IFMA52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 57–62, 2021.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Annual cryptology conference*, pages 868–886. Springer, 2012.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[CHK+21] Chi Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng Shih, and Bo Yang. NTT multiplication for NTT-unfriendly rings: New speed records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Trans. on Cryptographic Hardware and Embedded Systems*, pages 159–188, 2021.

[CP05] Richard Crandall and Carl Pomerance. *Prime numbers: a computational perspective*, volume 2. Springer, 2005.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 617–640. Springer, 2015.

[DMKMS24] Gabrielle De Micheli, Duhyeong Kim, Daniele Micciancio, and Adam Suhl. Faster amortized FHEW bootstrapping using ring automorphisms. In *IACR International Conference on Public-Key Cryptography*, pages 322–353. Springer, 2024.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[GP25] Antonio Guimarães and Hilder V. L. Pereira. Fast amortized bootstrapping with small keys and polynomial noise overhead. Cryptology ePrint Archive, Paper 2025/686, 2025.

[GPVL23] Antonio Guimarães, Hilder V. L. Pereira, and Barry Van Leeuwen. Amortized bootstrapping revisited: Simpler, asymptotically-faster, implemented. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 3–35. Springer, 2023.

[GS66] William Morven Gentleman and Gordon Sande. Fast Fourier Transforms: For fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 563–578, 1966.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 75–92. Springer, 2013.

[HYJ+25] Syed Mahbub Hafiz, Bahattin Yildiz, Marcos Simplicio Jr, Thales B. Paiva, Henrique S. Ogawa, Gabrielle De Micheli, and Eduardo L. Cominetti. Incompleteness in Number-Theoretic Transforms: New tradeoffs and faster lattice-based cryptographic applications. In *10th IEEE European Symposium on Security and Privacy, EuroS&P 2025, Venice, Italy, July 1-3, 2025*. IEEE, 2025.

[Kan22] Matthias J. Kannwischer. *Polynomial multiplication for post-quantum cryptography*. PhD thesis, Radboud University, 2022. https://kannwischer.eu/thesis/phd-thesis-2022-05-09.pdf.

[Lip21] Babette A. M. Lips. The efficiency of polynomial multiplication methods for ring-based PQC algorithms of Round 3 of the NIST PQC competition. Master's thesis, Eindhoven University of Technology, 2021.

[LMK+23] Yongwoo Lee, Daniele Micciancio, Andrey Kim, Rakyong Choi, Maxim Deryabin, Jieun Eom, and Donghoon Yoo. Efficient FHEW bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 227–256. Springer, 2023.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*, pages 1–23. Springer, 2010.

[LS19] Vadim Lyubashevsky and Gregor Seiler. NTTRU: Truly fast NTRU using NTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):180–201, May 2019.

[LW23a] Feng-Hao Liu and Han Wang. Batch bootstrapping I: A new framework for SIMD bootstrapping in polynomial modulus. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 321–352. Springer, 2023.

[LW23b] Feng-Hao Liu and Han Wang. Batch bootstrapping II: Bootstrapping in polynomial modulus only requires $\tilde{O}(1)$ FHE multiplications in amortization. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 353–384. Springer, 2023.

[LW23c] Zeyu Liu and Yunhao Wang. Amortized functional bootstrapping in less than 7 ms, with $\tilde{O}(1)$ polynomial multiplications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 101–132. Springer, 2023.

[Moe76] Robert Moenck. Practical fast polynomial multiplication. In *Proc. of the 3rd ACM symposium on Symbolic and algebraic computation*, pages 136–148. ACM, 1976.

[MS18] Daniele Micciancio and Jessica Sorrell. Ring Packing and Amortized FHEW Bootstrapping. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 100:1–100:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Nat24] National Institute of Standards and Technology. FIPS203: Module-lattice-based key-encapsulation mechanism standard. Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Aug 2024. https://doi.org/10.6028/NIST.FIPS.203.

[Nus80] Henri Nussbaumer. Fast polynomial transform algorithms for digital convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(2):205–215, 1980.

[SSTX09] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 617–635. Springer, 2009.

[Zam22] Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. https://github.com/zama-ai/tfhe-rs.