Revisiting the Slot-to-Coefficient Transformation for BGV and BFV

Robin Geelen

COSIC, KU Leuven, Leuven, Belgium

Abstract. Numerous applications in homomorphic encryption require an operation that moves the slots of a ciphertext to the coefficients of a different ciphertext. For the BGV and BFV schemes, the only efficient algorithms to implement this slot-to-coefficient transformation were proposed in the setting of non-power-of-two cyclotomic rings. In this paper, we devise an FFT-like method to decompose the slot-to-coefficient transformation (and its inverse) for power-of-two cyclotomic rings. The proposed method can handle both fully and sparsely packed slots. Our algorithm brings down the computational complexity of the slot-to-coefficient transformation from a linear to a logarithmic number of FHE operations, which is shown via a detailed complexity analysis.
The new procedures are implemented in Microsoft SEAL for BFV. The experiments report a speedup of up to 44× when packing 2^{12} elements from GF(8191^8). We also study a fully packed bootstrapping operation that refreshes 2^{15} elements from GF(65537) and obtain an amortized speedup of 12×.

Keywords: Homomorphic encryption · Linear transformations · Bootstrapping · BGV · BFV

1 Introduction

Several fully homomorphic encryption (FHE) schemes offer the capability of encoding multiple numbers in a ciphertext [SV14]. This functionality of "packing" numbers together is referred to as *batching*, and each entry in the packed vector is called a *plaintext slot*. For example, the closely related BGV [BGV14] and BFV [Bra12, FV12] schemes encode a vector of numbers defined modulo a power of a prime. Similarly, the CKKS [CKKS17] scheme encodes a vector of complex numbers approximated up to a limited precision. A recent attempt was made to introduce batching in third generation schemes as well [LW23a]. However, those techniques remain currently only of theoretical interest and will not be considered in the rest of this paper.

A very common operation in both BGV/BFV and CKKS is converting between slot and coefficient representation. Informally, the slot-to-coefficient transformation is defined as follows: given one or multiple ciphertexts encoding $m_0, m_1, \ldots, m_{N-1}$ in the plaintext slots, compute a ciphertext that encrypts the polynomial $m_0 + m_1 \cdot X + \ldots + m_{N-1} \cdot X^{N-1}$. The most important applications of the slot-to-coefficient transformation include (amortized) bootstrapping [GHS12a, AP13, HS21, CH18, GV23, GIKV23, OPP23, CHK$^+$18, LW23b], scheme conversion [BGGJ20, LHH$^+$21, BCK$^+$23] and transciphering [CHK$^+$21]. The inverse operation (coefficient-to-slot transformation) also appears in these applications.

The slot-to-coefficient transformation and its inverse are linear operations. Several methods exist to compute it homomorphically, each of which is useful in a particular setting. For the CKKS scheme, one typically takes the number of messages N as a power

E-mail: robin.geelen@esat.kuleuven.be (Robin Geelen)

of two, and efficient algorithms have been proposed to compute the slot-to-coefficient transformation in that setting [CCS19, HHC19]. For the BGV and BFV schemes, there are two common options: one can take the number of messages N different from a power of two, which is the direction taken by HElib. In that case, an efficient method exists to compute the slot-to-coefficient transformation [HS18, HS21]. Alternatively, one can take the number of messages as a power of two, in which case no efficient algorithm for the slot-to-coefficient transformation has been proposed to the best of our knowledge.

The main focus of this paper is studying the slot-to-coefficient transformation for BGV and BFV in the power-of-two setting. Similarly to the CKKS case, we propose an FFT-like algorithm to decompose the transformation in multiple stages, which scales well even for a large number of plaintext slots. The algorithm is implemented in the Magma bootstrapping library [GV23] and Microsoft SEAL [SEA23].

1.1 Related Work

Research in the slot-to-coefficient transformation can be divided into two categories: on the one hand, several works bring down the computational complexity of generic linear transformations. Many of the techniques on this front are shared between the BGV/BFV and CKKS case. On the other hand, there also exist algorithms to decompose the slot-to-coefficient map into smaller linear transformations, each of which can then be evaluated with the generic approach. Below we give an extensive overview of the existing literature for BGV/BFV and CKKS.

1.1.1 Linear Transformations in BGV and BFV

Generic linear transformations. HElib contains several algorithms to evaluate generic homomorphic linear transformations [HS18, CCLS19]. The following two techniques are used: a linear transformation, which is a sum of D weighted "rotations", can be rewritten as a double summation. By taking $O(\sqrt{D})$ terms in both the inner and outer sum, one can reduce the number of rotations to $O(\sqrt{D})$. This algorithm is called a *baby-step/giant-step* implementation. Moreover, the so-called *hoisting* technique can be used to simultaneously compute all inner-sum rotations, which is more efficient than computing each of them separately. Technical details about both techniques are given in Section 2.4.

FFT-like decomposition. As mentioned earlier, the slot-to-coefficient transformation in HElib (also called the *evaluation map*) employs non-power-of-two message packing [HS21]. More specifically, one chooses a cyclotomic index m and then packs $N = \varphi(m)$ messages where $\varphi(\cdot)$ denotes Euler's totient function. The parameter m is typically chosen as a product of smaller prime powers for two reasons: it gives reasonably high packing capacity of $\mathbb{Z}_{p^e}$-vectors and allows FFT-like decomposition of the slot-to-coefficient transformation. However, this parameter setting also has significant disadvantages compared to the power-of-two case, including less efficient implementations and a larger noise variance of the input ciphertext during bootstrapping (the exact noise variance depends on the number of distinct prime factors in m, following the heuristic analysis of Halevi and Shoup [HS21]).

From the theoretical side, it is known that the slot-to-coefficient transformation can be evaluated in quasilinear time [AP13]. Note that this early work employed a "ring switching" technique to convert between different values of the cyclotomic index m, which was necessary to reach an efficient decomposition of the linear transformation. However, subsequent implementations of bootstrapping (including the one in HElib) do not use ring switching anymore, because it is conjectured that ring switching would not give a substantial performance benefit in practice [HS21].

Finally, we note that no FFT-like algorithm has yet been proposed for the slot-to-coefficient transformation in the case of power-of-two cyclotomics. To the best of our knowledge, the only approach is the one from SEAL [CH18], but it does not use an efficient decomposition into smaller-dimensional matrices.

1.1.2 Linear Transformations in CKKS

Generic linear transformations. As mentioned earlier, the strategy to evaluate generic linear transformations in CKKS is very similar to BGV/BFV (including optimizations such as baby-step/giant-step implementations and hoisting). Moreover, a *double-hoisting* method was proposed to accelerate the computation of the inner-sum rotations even more [BMTH21]. Notably, the double-hoisting technique carries over to BGV/BFV when hybrid key switching [KPZ21] is used.

FFT-like decomposition. The CKKS scheme is only used in combination with a power-of-two cyclotomic index m, which implies that $N = \varphi(m) = m/2$. Very similar FFT-like algorithms to decompose the slot-to-coefficient transformation were proposed by Chen et al. [CCS19] and by Han et al. [HHC19]. Compared to the BGV/BFV case, the CKKS transformations resemble much more a classical FFT algorithm due to exclusive use of power-of-two packing. Technical details about those algorithms are discussed in Section 2.3, where we approach the problem from the same point of view as Han et al.

1.2 Contributions and Outline

This work makes the following contributions:

- Section 3 describes new properties of the automorphism group and plaintext packing for BGV/BFV. For a prime-power plaintext modulus p^e, we make a crucial difference between the situation $p = 1 \pmod 4$ (where the rotation group of the plaintext slots has rank two) and $p = 3 \pmod 4$ (where the rotation group of the plaintext slots has rank one). Sparsely packed slots are handled as a special case of fully packed slots by encoding messages in a subring.
- Section 4 describes a CKKS-like method to decompose the slot-to-coefficient transformation in BGV and BFV, which scales well even for a large number of slots. The complexity of the new method is analyzed and compared to related work in Section 5. The implementation and results are discussed in Section 6.[1] Specifically, we compare a single linear transformation with the state-of-the-art, and we also demonstrate the advantage of our method by running a fully packed bootstrapping application.

2 Preliminaries

2.1 Notations

We will use power-of-two cyclotomic indices m and m', where m' divides m. Their totient is written as $N = \varphi(m) = m/2$ and $N' = \varphi(m') = m'/2$ respectively. The involved homomorphic encryption schemes work over the ring $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ and its subring $\mathcal{R}' = \mathbb{Z}[X^{N/N'}]/(X^N + 1)$. For an integer $k \geq 2$, we write the quotient ring of $\mathcal{R}$ modulo k as $\mathcal{R}_k = \mathcal{R}/k\mathcal{R}$ and similarly for $\mathcal{R}'$. All ring elements are shown in bold lower case letters (e.g., $\boldsymbol{a} \in \mathcal{R}$) or explicitly as polynomials (e.g., $a(X) \in \mathcal{R}$). The infinity norm of $\boldsymbol{a} \in \mathcal{R}$ (i.e., its largest coefficient) is denoted by $||\boldsymbol{a}||_\infty$. The unit group of integers modulo m is written as $\mathbb{Z}_m^*$ (this is isomorphic to the automorphism group of $\mathcal{R}/\mathbb{Z}$). The subgroup generated by $g \in \mathbb{Z}_m^*$ is denoted by $\langle g \rangle$. Finally, we summarize commonly used symbols in Table 1 (some of these notations will be introduced in later sections).

[1] Code is available at https://github.com/KULeuven-COSIC/Bootstrapping_BGV_BFV/tree/traces.

Table 1: List of commonly used symbols and their meaning

Symbol	Meaning
m (or m')	Power-of-two cyclotomic index ($m \geq m'$)
N (or N')	Totient of cyclotomic index m (or m')
$\mathcal{R}$ (or $\mathcal{R}'$)	Cyclotomic ring of index m (or m')
E (or E')	Slot algebra for m (or m') and p^e
p^e	Plaintext modulus for BGV/BFV
q	Ciphertext modulus for BGV/BFV
d	Multiplicative order of p in $\mathbb{Z}_m^*$
ℓ	Number of slots (equals N/d)

2.2 BGV and BFV Encryption

BGV and BFV encrypt plaintexts from the ring $\mathcal{R}_{p^e}$ for some prime p and positive integer e. We will see the plaintext space $\mathcal{R}_{p^e}$ as a subset of $\mathcal{R}$ where polynomials have coefficients in $[-p^e/2, p^e/2) \cap \mathbb{Z}$. As mentioned above, we will only consider power-of-two-dimensional rings $\mathcal{R}$, although one can easily generalize encryption to arbitrary cyclotomic rings. A ciphertext is a pair of ring elements, i.e., it lives in $\mathcal{R}_q^2$. For correctness, the ciphertext modulus needs to be much larger than the plaintext modulus ($q \gg p^e$).

A BGV ciphertext $(\boldsymbol{c}_0, \boldsymbol{c}_1) \in \mathcal{R}_q^2$ is said to encrypt the plaintext $\boldsymbol{m} \in \mathcal{R}_{p^e}$ under secret key $\boldsymbol{s} \in \mathcal{R}$ if

$$\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} = \boldsymbol{m} + p^e \boldsymbol{e} \pmod{q}$$

for some noise term $\boldsymbol{e} \in \mathcal{R}$ that satisfies $||\boldsymbol{e}||_\infty < (q/p^e - 1)/2$. Similarly, a valid BFV ciphertext satisfies

$$\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} = \lfloor (q/p^e) \cdot \boldsymbol{m} \rceil + \boldsymbol{e} \pmod{q},$$

where the bound on $\boldsymbol{e}$ is the same.

2.2.1 Homomorphic Operations in BGV and BFV

BGV and BFV offer the same set of homomorphic operations over the plaintext space and have the same asymptotic performance. Each homomorphic operation causes a certain amount of noise growth (the term $\boldsymbol{e}$ will grow larger). The following operations are defined:

- Addition: given two ciphertexts that encrypt $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$, compute a new ciphertext that encrypts $\boldsymbol{m}_1 + \boldsymbol{m}_2$. The noise growth of addition is additive (the new noise is roughly equal to the sum of the noise terms). Addition is generally considered a cheap operation in terms of execution time.
- Multiplication: given two ciphertexts that encrypt $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$, compute a new ciphertext that encrypts $\boldsymbol{m}_1 \cdot \boldsymbol{m}_2$. Roughly, one can bound the new noise after multiplication by $c \cdot (||\boldsymbol{e}_1||_\infty + ||\boldsymbol{e}_2||_\infty)$, where c is a constant that grows linearly in p^e and quadratically in N (so larger parameters result in more noise). In terms of execution time, multiplication is much more expensive than addition, because it requires an expensive post-processing step called key switching.
- Automorphism: given a ciphertext that encrypts $a(X)$ and given a number $j \in \mathbb{Z}_m^*$, compute a new ciphertext that encrypts $a(X^j)$ (note that reduction modulo $X^N + 1$ is implicit here). The automorphism $a(X) \mapsto a(X^j)$ will be denoted by τ_j hereafter. The noise growth of automorphism is additive (the new noise is equal to the old noise plus some extra constant term). In terms of execution time, the cost of automorphism is similar to multiplication as it also requires key switching.

Next to the ciphertext-ciphertext addition and multiplication as defined above, one can also define addition and multiplication between a plaintext and a ciphertext. Plaintext-ciphertext multiplication is much cheaper than ciphertext-ciphertext multiplication, because no key switching is required. For more information about the practical performance of the homomorphic operations, we refer to Kim et al. [KPZ21].

2.2.2 Plaintext Slots in BGV and BFV

Based on the Chinese remainder theorem, Smart and Vercauteren [SV14] proposed a method to pack multiple numbers in a ciphertext, and perform "single instruction, multiple data" operations on these numbers. The idea relies on the following lemma.

Lemma 1. *Let p be an odd prime number and let e be a positive integer as above. Let $m \geq 2$ be a power-of-two cyclotomic index and $N = m/2$. Then the m-th cyclotomic polynomial factors modulo p^e as*

$$X^N + 1 = F_1(X) \cdot \ldots \cdot F_\ell(X) \pmod{p^e}. \tag{1}$$

Each factor is of degree d, which is the multiplicative order of p modulo m, and the number of factors is $\ell = N/d$.

As such, using the Chinese remainder theorem, the plaintext ring is isomorphic to

$$\mathcal{R}_{p^e} \cong \mathbb{Z}_{p^e}[X]/(F_1(X)) \times \ldots \times \mathbb{Z}_{p^e}[X]/(F_\ell(X)),$$

where addition and multiplication at the right-hand side correspond to component-wise addition and multiplication. Since all rings $\mathbb{Z}_{p^e}[X]/(F_i(X))$ are Galois rings of the same parameters, they are isomorphic to each other. Therefore, we can see the plaintext space as ℓ copies of $\mathbb{Z}_{p^e}[X]/(F_1(X))$, each of which is called a *plaintext slot.*

Slot permutations. Gentry et al. [GHS12b] noticed that one can homomorphically apply permutations to the plaintext slots using the automorphisms from above. This is most easily seen as follows: first, define the *slot algebra* as $E = \mathbb{Z}_{p^e}[\zeta_m]$ with ζ_m a formal root of $F_1(X)$, and let $S \subseteq \mathbb{Z}$ be a complete system of representatives for $\mathbb{Z}_m^*/\langle p \rangle$ (as done in HElib [HS18]). In practice, we construct the set S as

$$S = \{g_1^{e_1} \cdot \ldots \cdot g_t^{e_t} \mid 0 \leq e_i < \ell_i\}, \tag{2}$$

where $\ell = \ell_1 \cdot \ldots \cdot \ell_t$ is the number of slots. The integers g_i and e_i are called generators and orders respectively.[2] The plaintext ring is now isomorphic to E^ℓ, which can be explicitly computed as

$$\mathcal{R}_{p^e} \to E^\ell \colon a(X) \mapsto \left\{a(\zeta_m^h)\right\}_{h \in S}. \tag{3}$$

As such, each plaintext slot corresponds to one index $h \in S$ or one tuple $(e_1, \ldots, e_t)$. By associating each plaintext slot with such a tuple, the full plaintext space corresponds to a t-dimensional hypercube [HS20].

To enable permutations on a plaintext $\boldsymbol{m}$, we take $0 \leq v < \ell_i$ and compute

$$\rho_i^v(\boldsymbol{m}) = \boldsymbol{\mu} \cdot \tau_j(\boldsymbol{m}) + (1 - \boldsymbol{\mu}) \cdot \tau_k(\boldsymbol{m}), \tag{4}$$

where $j = g_i^{-v} \pmod m$ and $k = g_i^{\ell_i - v} \pmod m$, and $\boldsymbol{\mu}$ is the "mask" obtained by embedding '0' in the plaintext slots with $e_i < v$, and embedding '1' in the other slots. Letting $\boldsymbol{m}' = \rho_i^v(\boldsymbol{m})$, it is easy to see that the value of $\boldsymbol{m}'$ in slot $(e_1, \ldots, e_i', \ldots, e_t)$ is equal to the value of $\boldsymbol{m}$ in slot $(e_1, \ldots, e_i, \ldots, e_t)$ with $e_i' = e_i + v \pmod{\ell_i}$. Since the permutation ρ_i^v only acts on a single index in the tuple (by mapping each slot v positions forward), it is called a one-dimensional *rotation* [HS18].

In general, computing the rotation from Equation (4) requires two automorphisms. However, if $g_i^{\ell_i} = 1 \pmod m$, then the equation simplifies to $\rho_i^v(\boldsymbol{m}) = \tau_j(\boldsymbol{m})$, so we need only one automorphism. For $1 \leq i \leq t$, we call dimension i "good" if only one automorphism is required and "bad" otherwise [HS18].

[2] More information about the construction of S for power-of-two cyclotomics will be given in Section 3.

The Frobenius map. The rotations from Equation (4) use two automorphisms τ_j and τ_k, which represent the elements of S (and thereby the quotient group $\mathbb{Z}_m^*/\langle p\rangle$). However, the full automorphism group of $\mathcal{R}/\mathbb{Z}$ consists of τ_j with $j \in \mathbb{Z}_m^*$. The remaining automorphisms for $j \in \langle p\rangle$ induce automorphisms on E: they act on each plaintext slot individually as the map $a(\zeta_m) \mapsto a(\zeta_m^j)$ for arbitrary $a(X)$. This subgroup of automorphisms is generated by $\sigma = \tau_p$ (the so-called *Frobenius map*), which acts on the slots as $\sigma_E\colon a(\zeta_m) \mapsto a(\zeta_m^p)$.

2.3 The Slot-to-Coefficient Transformation in CKKS

The slot-to-coefficient transformation is achieved by evaluating the decoding function homomorphically, which is a linear transformation over the plaintext slots. Similarly, the coefficient-to-slot transformation evaluates the encoding function, which corresponds to the inverse linear transformation. In the CKKS scheme, this involves multiplication by an $m/4 \times m/4$-matrix defined as

$$S_{m/4} = \left(\zeta_m^{5^i \cdot \mathsf{rev}_{m/4}(j)}\right)_{0 \le i,j < m/4}, \tag{5}$$

where ζ_m is a primitive m-th root of unity, and $\mathsf{rev}_{m/4}$ denotes the standard bit-reversal permutation of $m/4$ items (the definition of $S_{m/4}$ above specifies entries in row i and column j). Although CKKS defines the matrix over the complex numbers, it trivially extends to rings that have a primitive m-th root of unity ζ_m (such as the slot algebra E).

In order to efficiently evaluate multiplication by $S_{m/4}$, Han et al. [HHC19] show an FFT-like method to decompose this matrix into a product of sparse matrices. More specifically, they prove the following lemma.

Lemma 2. *Let $S_{m/4}$ be as above, and correspondingly, let $S_{m/8}$ be the $m/8 \times m/8$-matrix defined with respect to $\zeta_{m/2} = \zeta_m^2$. Then for $m \ge 8$, we have*

$$S_{m/4} = \begin{bmatrix} I & W_{m/8} \\ I & -W_{m/8} \end{bmatrix} \cdot \begin{bmatrix} S_{m/8} & 0 \\ 0 & S_{m/8} \end{bmatrix},$$

where $W_{m/8} = \mathsf{diag}(\zeta_m^{5^i})_{0 \le i < m/8}$.

By applying the above lemma recursively on $S_{m/8}$, we can factor $S_{m/4}$ into a product of $\log_2(m/4)$ sparse matrices (each of which contains only $m/2$ non-zero elements). Finally, note that it can be useful to exploit an "incomplete" factorization of $S_{m/4}$ in which multiple factors are merged, as this leads to less noise growth in homomorphic encryption (additional details will be given later).

2.4 Baby-Step/Giant-Step Algorithm

2.4.1 Linear Transformations on $\mathcal{R}_{p^e}$

To implement multiplication by the matrices from Section 2.3, one can use the baby-step/giant-step algorithm [HS18]. This algorithm multiplies the vector of plaintext slots (in one dimension of S) by a generic matrix. More specifically, one can express an E-linear transformation L on a plaintext $\boldsymbol{m} \in \mathcal{R}_{p^e}$ in dimension i as

$$L(\boldsymbol{m}) = \sum_{v=0}^{\ell_i - 1} \boldsymbol{\kappa}(v) \cdot \rho_i^v(\boldsymbol{m}),$$

where $\boldsymbol{\kappa}(v) \in \mathcal{R}_{p^e}$ are appropriate constants. Letting $g = \lceil \sqrt{\ell_i} \rceil$ and $h = \lceil \ell_i / g \rceil$, the idea is to rewrite the linear transformation as

$$\begin{aligned} L(\boldsymbol{m}) &= \sum_{v=0}^{\ell_i - 1} \boldsymbol{\kappa}(v) \cdot (\boldsymbol{\mu}(v) \cdot \tau^v(\boldsymbol{m}) + (1 - \boldsymbol{\mu}(v)) \cdot \tau^{v-\ell_i}(\boldsymbol{m})) \\ &= \sum_{k=0}^{h-1} \tau^{gk} \left[\sum_{j=0}^{g-1} (\boldsymbol{\kappa}'(j+gk) \cdot \tau^j(\boldsymbol{m}) + \boldsymbol{\kappa}''(j+gk) \cdot \tau^j(\tau^{-\ell_i}(\boldsymbol{m}))) \right], \end{aligned} \tag{6}$$

where $\tau = \tau_{g_i}^{-1}$ and

$$\begin{aligned} \boldsymbol{\kappa}'(j+gk) &= \tau^{-gk}(\boldsymbol{\mu}(j+gk) \cdot \boldsymbol{\kappa}(j+gk)), \\ \boldsymbol{\kappa}''(j+gk) &= \tau^{-gk}((1 - \boldsymbol{\mu}(j+gk)) \cdot \boldsymbol{\kappa}(j+gk)). \end{aligned}$$

It is clear that Equation (6) can be computed using $O(\sqrt{\ell_i})$ automorphisms and $O(\ell_i)$ plaintext-ciphertext multiplications. Moreover, if key switching keys are available for all j, then we can use well-known (double-)hoisting techniques [HS18, BMTH21] to speed up the computation of $\tau^j(\boldsymbol{m})$ and $\tau^j(\tau^{-\ell_i}(\boldsymbol{m}))$. This is possible because both are sequences of automorphisms on the same input (respectively $\boldsymbol{m}$ and $\tau^{-\ell_i}(\boldsymbol{m})$).

In a good dimension, Equation (6) collapses to

$$L(\boldsymbol{m}) = \sum_{k=0}^{h-1} \tau^{gk} \left[\sum_{j=0}^{g-1} \boldsymbol{\kappa}'(j+gk) \cdot \tau^j(\boldsymbol{m}) \right], \tag{7}$$

where $\boldsymbol{\kappa}'(j+gk) = \tau^{-gk}(\boldsymbol{\kappa}(j+gk))$. This saves approximately 50% of the automorphisms and plaintext-ciphertext multiplications in the inner-sum computation.

2.4.2 Linear Transformations on E

Similarly to above, one can express a $\mathbb{Z}_{p^e}$-linear transformation on $a \in E$ as a weighted sum of $\sigma_E^v(a)$. As such, for a plaintext $\boldsymbol{m} \in \mathcal{R}_{p^e}$, the map

$$L(\boldsymbol{m}) = \sum_{v=0}^{d-1} \boldsymbol{\kappa}(v) \cdot \sigma^v(\boldsymbol{m}) \tag{8}$$

acts on each slot individually as a $\mathbb{Z}_{p^e}$-linear transformation. This functionality can be implemented in the same manner as Equation (7) if we take $\tau = \sigma$. The cost is dominated by $O(\sqrt{d})$ automorphisms and $O(d)$ plaintext-ciphertext multiplications.

2.4.3 Multidimensional Baby-Step/Giant-Step Algorithm

The baby-step/giant-step algorithm (for E-linear as well as $\mathbb{Z}_{p^e}$-linear transformations) can be generalized to a multidimensional version [CCLS19]. More specifically, the goal is to compute weighted sums of automorphisms τ_i with $i \in I \subseteq \mathbb{Z}_m^*$. We first split the index set I in two components $J, K \subseteq \mathbb{Z}_m^*$ such that each $i \in I$ can be written as $i = jk$ with $j \in J$ and $k \in K$. For a plaintext $\boldsymbol{m} \in \mathcal{R}_{p^e}$, we can now rearrange expressions of the form

$$L(\boldsymbol{m}) = \sum_{i \in I} \boldsymbol{\kappa}(i) \cdot \tau_i(\boldsymbol{m}) = \sum_{k \in K} \tau_k \left[\sum_{j \in J} \boldsymbol{\kappa}'(jk) \cdot \tau_j(\boldsymbol{m}) \right], \tag{9}$$

where $\boldsymbol{\kappa}'(jk) = \tau_k^{-1}(\boldsymbol{\kappa}(jk))$. We note that arbitrary linear transformations can be expressed in the form of Equation (9), so we are neither limited to slot-wise nor one-dimensional linear transformations. Finally, when merging an E-linear and a $\mathbb{Z}_{p^e}$-linear map, one can sometimes evaluate a bad dimension with the same cost as a good dimension, using the strategy of reassigning incomplete rotations [CCLS19].

3 An Alternative View of Plaintext Encoding

The goal of this section is to introduce useful properties of power-of-two cyclotomics. We first discuss the algebraic structure of the automorphism group and construction of the representative set S which was defined earlier. Then we derive properties about the factorization of cyclotomic polynomials, which will naturally lead to a method for encoding plaintext vectors in a subring of $\mathcal{R}_{p^e}$.

3.1 Structure of the Automorphism Group and Plaintext Slots

For $m \geq 4$ a power of two, it is a well-known fact that $\mathbb{Z}_m^* = \langle 5 \rangle \times \langle -1 \rangle$, where 5 has order $m/4$ and -1 has order 2 [Gau86]. Consequently, this group has two generators and is thus not cyclic for $m \geq 8$. Another useful property is that for $1 \leq r \leq m/4$ with r a power of two, the subgroup $\langle 5^r \rangle$ coincides with the set consisting of all numbers x such that $x = 1 \pmod{4r}$. This is because both sets are subgroups of the cyclic group $\langle 5 \rangle$, and both have the same order $m/(4r)$.

Lemma 3. *Let $m \geq 4$ be a power of two and consider a prime p. If $p = 4r \cdot k + 1$ with r a power of two and k odd, then the number of slots is $\ell = \min(2r, m/2)$. If $p = 4r \cdot k - 1$ with r a power of two and k odd, then the number of slots is $\ell = \min(2r, m/4)$.*

Proof. First note that the proof is trivial if $r \geq m/4$, because $p = \pm 1 \pmod{m}$ in that case. Therefore, the order of p will be equal to 1 or 2, giving $m/2$ or $m/4$ slots respectively. We will therefore assume that $1 \leq r < m/4$ in the rest of the proof.

We now prove the case $p = 4r \cdot k + 1$. Using the property from above, we know that $p \in \langle 5^r \rangle$. On the other hand, we know that $p \notin \langle 5^{2r} \rangle$ since k is odd, so p is not in any strict subgroup of $\langle 5^r \rangle$. It follows that p generates $\langle 5^r \rangle$ and the order of p is equal to $d = m/(4r)$. The number of slots is $\ell = m/(2d) = 2r \leq m/2$.

We now prove the case $p = 4r \cdot k - 1$. Using a similar reasoning as above, we obtain that the order of $-p$ is equal to $m/(4r)$. Since this order is at least 2, it follows that the order of p is also $d = m/(4r)$ and the number of slots is $\ell = m/(2d) = 2r \leq m/4$. □

A noteworthy corollary is that the number of slots is at most $(p+1)/2$. Hence one can only have a good packing density if p is large. The maximum number of $N = m/2$ slots is reached when $p = 1 \pmod{m}$.

3.1.1 Construction of the Representative Set

The set S from Equation (2) forms a complete system of representatives for $\mathbb{Z}_m^*/\langle p \rangle$ and can therefore be constructed with one or two generators. If $p = 1 \pmod 4$, then $p \in \langle 5 \rangle$ and the group $\mathbb{Z}_m^*/\langle p \rangle$ is not cyclic in general. Therefore, we use generators $g_1 = 5$ (of order $\ell_1 = \ell/2$) and $g_2 = -1$ (of order $\ell_2 = 2$). On the other hand, if $p = 3 \pmod 4$, then $p \in -\langle 5 \rangle$ and $\mathbb{Z}_m^*/\langle p \rangle$ is cyclic. Therefore, we use generator $g_1 = 5$ (of order $\ell_1 = \ell$).

We can also reinterpret the set S into $\log_2(\ell)$ dimensions of size 2. This is done by considering S through the generators $5^n, 5^{n/2}, \ldots, 5^2$, where $n = \ell/4$ if $p = 1 \pmod 4$ and $n = \ell/2$ if $p = 3 \pmod 4$. This interpretation is useful for decomposing the slot-to-coefficient transformation in multiple stages. Intermediate interpretations with fewer dimensions of larger size are also possible.

3.2 Factorization of Cyclotomic Polynomials

The following standard result can be obtained by merging the proofs from Lyubashevsky and Seiler [LS18] and Okada et al. [OPP23]. We also provide a unified and simplified proof for comprehensiveness below.

Lemma 4. *Let $m \geq 4$ be a power of two, then each factor in Equation (1) is of the shape $F_i(X) = X^d + a_i \cdot X^{d/2} + b_i$, where $a_i = 0$ if $p = 1 \pmod 4$.*

Proof. We first prove the case $p = 1 \pmod 4$. Using the result from Lemma 3, we know that $p = 4r \cdot k + 1$ where ℓ divides $2r$. Let $m' = m/d$, then substituting $2r$ by ℓ in the previous equation gives $p = 2\ell \cdot k' + 1 = m' \cdot k' + 1$, so $p = 1 \pmod{m'}$. Using Lemma 1, the m'-th cyclotomic polynomial splits modulo p^e into linear factors:

$$X^{N'} + 1 = F'_1(X) \cdot \ldots \cdot F'_\ell(X) \pmod{p^e}.$$

Then we explicitly obtain the factors of $X^N + 1$ as $F_i(X) = F'_i(X^{N/N'}) = F'_i(X^d)$.

We now prove the case $p = 3 \pmod 4$. Using the result from Lemma 3, we know that $p = 4r \cdot k - 1$ where ℓ divides $2r$. Let $m' = 2m/d$, then substituting $2r$ by ℓ in the previous equation gives $p = 2\ell \cdot k' - 1 = (m'/2) \cdot k' - 1$, so $p^2 = 1 \pmod{m'}$ while $p \neq 1 \pmod{m'}$. Using Lemma 1, the m'-th cyclotomic polynomial splits modulo p^e into quadratic factors:

$$X^{N'} + 1 = F'_1(X) \cdot \ldots \cdot F'_\ell(X) \pmod{p^e}.$$

Then we explicitly obtain the factors of $X^N + 1$ as $F_i(X) = F'_i(X^{N/N'}) = F'_i(X^{d/2})$. □

3.2.1 Encoding Plaintext Vectors in a Subring

One does not have to use the full packing capacity of $\mathcal{R}_{p^e}$, but can also encode plaintext vectors in a subring $\mathcal{R}'_{p^e}$. Since the algebraic structure of $\mathcal{R}'_{p^e}$ depends on m' instead of m, this will result in fewer plaintext slots (smaller ℓ) or a lower extension degree of the slot algebra (smaller d) or both. Packing in a subring is useful in applications where a small number of messages suffices.

To take a plaintext from $\mathcal{R}_{p^e}$ to $\mathcal{R}'_{p^e}$, we can homomorphically evaluate the trace of $\mathcal{R}/\mathcal{R}'$. This operation is commonly called coefficient selection [CH18] or subsum [CCS19] in bootstrapping. Following the definition of the trace, it results in

$$\sum_{i=0}^{N-1} m_i \cdot X^i \mapsto (N/N') \cdot \left(\sum_{i=0}^{N'-1} m_{i \cdot N/N'} \cdot X^{i \cdot N/N'} \right).$$

It can be implemented efficiently using only $\log_2(N/N')$ automorphisms by going through all intermediate cyclotomic rings between $\mathcal{R}$ and $\mathcal{R}'$, and then iteratively evaluating the trace for all subrings [AP13]. The factor N/N' can be removed by folding $(N/N')^{-1} \pmod{p^e}$ in any subsequent linear transformation. We omit further details.

Example 1. We say that a plaintext is *sparsely packed* if each slot contains only an element from $\mathbb{Z}_{p^e} \subseteq E$. According to the Chinese remainder theorem, such a plaintext is constructed as

$$\boldsymbol{m} = \sum_{i=1}^{\ell} m_i \cdot G_i(X), \quad \text{where} \quad G_i(X) = \frac{X^N + 1}{F_i(X)} \cdot \left[\left(\frac{X^N + 1}{F_i(X)} \right)^{-1} \pmod{F_i(X)} \right].$$

Here the polynomials $F_i(X)$ denote the factors from Equation (1) and $m_i \in \mathbb{Z}_{p^e}$. Reduction modulo p^e is implicit in the equation above. Following Lemma 4, it is easy to see that $F_i(X) \in \mathbb{Z}_{p^e}[X^c]$ and therefore $G_i(X) \in \mathbb{Z}_{p^e}[X^c]$, where $c = d$ if $p = 1 \pmod 4$ and $c = d/2$ if $p = 3 \pmod 4$. It follows directly that $\boldsymbol{m} \in \mathcal{R}'_{p^e}$ with respect to $m' = m/c$.

4 The New Transformation for BGV and BFV

This section describes the new version of the slot-to-coefficient and coefficient-to-slot transformations, covering fully packed and sparsely packed slots. Our analysis distinguishes the case $p = 1 \pmod 4$ (treated in Section 4.2) from $p = 3 \pmod 4$ (treated in Section 4.3). These require a fundamentally different linear transformation, because the first case has a non-cyclic and the second case has a cyclic permutation group.

4.1 Intuition Behind the Proposed Method

In both variants of our method, we start from a ciphertext that encodes the desired numbers with respect to the "power basis" of $\zeta_{m,i}$ in slot i. First, we apply a map M to make the encoding uniform (i.e., each slot must use the power basis of ζ_m). Then we evaluate homomorphic matrix-vector multiplication by U_ℓ, which is an FFT-like linear transformation that evaluates in the roots of unity. Finally, we apply the inverse map M^{-1} to change the encoding back to the power basis of $\zeta_{m,i}$ in slot i. Remark that the proposed method is equivalent to CKKS if $p = 3 \pmod 4$ and the slot width is $d = 2$.

In the case $p = 3 \pmod 4$, we additionally use the fourth root of unity ζ_4. It can be interpreted as the equivalent of the imaginary unit in the complex numbers. This fourth root of unity is invariant under the applied transformation, thereby making it E'-linear. This approach is not possible in the case $p = 1 \pmod 4$ because $\zeta_4 \in \mathbb{Z}_{p^e}$.

4.2 New Method for $p = 1 \pmod 4$

We will identify the slots of a plaintext with a vector in E^ℓ. This is done by "flattening" the representative set as $S = \{1, 5, \ldots, 5^{\ell_1 - 1}, -1, -5, \ldots, -5^{\ell_1 - 1}\}$ and filling the plaintext in Equation (3). We also use the notation $\zeta_{m,i} = \zeta_m^{h_i}$, where h_i is the i-th element of S.

4.2.1 Fully Packed Slots

In the slot-to-coefficient transformation, we start from a plaintext $\boldsymbol{m}$ that encodes

$$\vec{m} = \left(\sum_{j=0}^{d-1} m_{i,j} \cdot \zeta_{m,i}^j \right)_{0 \le i < \ell}$$

in the slots. Note that the encoding is done with respect to $\zeta_{m,i}^j$ instead of ζ_m^j so that we can use powers of X as the packing constants later. The goal is to map the elements $m_{i,j}$ to the coefficients of a new plaintext $\boldsymbol{m}'''$. This can be done in three steps:

1. Perform a slot-wise linear transformation M that maps $\zeta_{m,i}^j \mapsto \zeta_m^j$ for $0 \le j < d$ in slot i and extends by $\mathbb{Z}_{p^e}$-linearity. Note that this transformation depends on i and is thus different for each slot. After this step, the plaintext encodes the slot-vector

$$\vec{m}' = \left(\sum_{j=0}^{d-1} m_{i,j} \cdot \zeta_m^j \right)_{0 \le i < \ell} .$$

2. Multiply the slot-vector by (a column-permuted version of) the decoding matrix

$$U_\ell = \left(\zeta_{m,i}^{d \cdot j} \right)_{0 \le i,j < \ell},$$

where the matrix above is specified by the entries in its i-th row and j-th column. Observe that $\zeta_m^d \in \mathbb{Z}_{p^e}$, which implies $U_\ell \in \mathbb{Z}_{p^e}^{\ell \times \ell}$. After this step, the plaintext encodes the slot-vector

$$\vec{m}'' = \left(\sum_{k=0}^{\ell-1} \sum_{j=0}^{d-1} m_{k,j} \cdot \zeta_m^j \cdot \zeta_{m,i}^{d \cdot k} \right)_{0 \le i < \ell} .$$

A column-permuted version of U_ℓ will simply map the elements $m_{i,j}$ to a different order of the coefficients.

3. Perform the linear transformation M^{-1} that maps $\zeta_m^j \mapsto \zeta_{m,i}^j$ for $0 \leq j < d$ in slot i. After this step, the plaintext encodes the slot-vector

$$\vec{m}''' = \left(\sum_{k=0}^{\ell-1} \sum_{j=0}^{d-1} m_{k,j} \cdot \zeta_{m,i}^{j+d \cdot k} \right)_{0 \leq i < \ell},$$

which corresponds to the plaintext

$$\boldsymbol{m}''' = \sum_{k=0}^{\ell-1} \sum_{j=0}^{d-1} m_{k,j} \cdot X^{j+d \cdot k}.$$

Here we used the observation that M^{-1} does not act on the entries of U_ℓ (i.e., powers of ζ_m^d) due to $\mathbb{Z}_{p^e}$-linearity.

We emphasize that the slot-to-coefficient transformation is only $\mathbb{Z}_{p^e}$-linear and not E-linear in general, which is why step 1 and step 3 are required (step 2 alone is E-linear). The inverse map (coefficient-to-slot transformation) is given by $M^{-1} U_\ell^{-1} M$ instead of $M^{-1} U_\ell M$.

Matrix decomposition. The matrix from step 2 can be decomposed into a product of sparse matrices. To do this, we use the column-permuted version

$$U_\ell = \begin{bmatrix} S_{\ell/2} & \zeta_4 \cdot S_{\ell/2} \\ S'_{\ell/2} & -\zeta_4 \cdot S'_{\ell/2} \end{bmatrix} = \begin{bmatrix} S_{\ell/2} & 0 \\ 0 & S'_{\ell/2} \end{bmatrix} \cdot \begin{bmatrix} I & \zeta_4 \cdot I \\ I & -\zeta_4 \cdot I \end{bmatrix},$$

where $\zeta_4 = \zeta_m^{m/4}$. The submatrices $S_{\ell/2}$ and $S'_{\ell/2}$ are defined in Equation (5) and generated by the roots of unity ζ_m^d and ζ_m^{-d} respectively. Observe that $S_{\ell/2}$ and $S'_{\ell/2}$ can be further decomposed using Lemma 2. As a result, we can write U_ℓ as a product of $\log_2(\ell)$ sparse matrices, each of which acts on only one dimension of S (in the alternative interpretation where each dimension has size 2). The leftmost factor of this product will act on the first dimension and the rightmost factor on the last dimension.

Unpacking the slots. After the coefficient-to-slot operation, bootstrapping needs to split the ciphertext that encrypts $\boldsymbol{m}$ in d sparsely packed ciphertexts. This can be done as follows: first, we homomorphically split the plaintext in two parts $\boldsymbol{m}_1 = \boldsymbol{m} + \sigma^{d/2}(\boldsymbol{m})$ and $\boldsymbol{m}_2 = X^{-1} \cdot (\boldsymbol{m} - \sigma^{d/2}(\boldsymbol{m}))$. The automorphism $\sigma^{d/2}$ will simply map $X \mapsto -X$ and thus acts on the slots as $\zeta_m \mapsto -\zeta_m$ (note that this is indeed an automorphism of E due to the special shape of $F_i(X)$ in Lemma 4). Therefore, the plaintexts will encode the slot-vectors

$$\vec{m}_1 = \left(2 \cdot \sum_{j=0}^{d/2-1} m_{i,2j} \cdot \zeta_{m,i}^{2j} \right)_{0 \leq i < \ell} \quad \text{and} \quad \vec{m}_2 = \left(2 \cdot \sum_{j=0}^{d/2-1} m_{i,2j+1} \cdot \zeta_{m,i}^{2j} \right)_{0 \leq i < \ell}.$$

We emphasize that multiplication by powers of X does not increase the noise as it just permutes coefficients negacyclicly. To obtain d sparse ciphertexts, we apply this procedure iteratively on the plaintexts $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$: in level $i = 1, \ldots, \log_2(d)$ of the iteration, we compute $\boldsymbol{m}_1 = \boldsymbol{m} + \sigma^{d/2^i}(\boldsymbol{m})$ and $\boldsymbol{m}_2 = X^{-2^{i-1}} \cdot (\boldsymbol{m} - \sigma^{d/2^i}(\boldsymbol{m}))$, where $\boldsymbol{m}$ loops over all outputs of the previous iteration. At the end of the unpacking procedure, the undesired factors of 2 in $\vec{m}_1$ and $\vec{m}_2$ will accumulate to a factor of d. It can be removed by folding a factor of $d^{-1} \pmod{p^e}$ in the preceding linear transformation.

Repacking the slots. Before the slot-to-coefficient transformation, bootstrapping needs to recombine d sparsely packed ciphertexts into one fully packed ciphertext. This is the inverse of unpacking and can be computed as $\boldsymbol{m} = \boldsymbol{m}_1/2 + X^{2^{i-1}} \cdot \boldsymbol{m}_2/2$, where we loop over i in reverse order than during unpacking. Note that the input of repacking is indeed given by $\boldsymbol{m}_1/2$ and $\boldsymbol{m}_2/2$ rather than $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$, assuming that the factor of d was removed appropriately before unpacking.

4.2.2 Sparsely Packed Slots

According to Example 1, sparsely packed plaintexts (where the slots contain an element from $\mathbb{Z}_{p^e}$) live in the subring $\mathcal{R}'_{p^e}$ with respect to $m' = m/d$. Conversely, it is easy to see that elements from the subring $\mathcal{R}'_{p^e}$ are sparsely packed. As such, both the slot-encoded plaintext $\boldsymbol{m}$ and the coefficient-encoded plaintext $\boldsymbol{m}'''$ are sparsely packed. Step 1 and step 3 of the slot-to-coefficient transformation can now be omitted, because they have no effect on $\mathbb{Z}_{p^e}$. Therefore, the slot-to-coefficient transformation is U_ℓ and the coefficient-to-slot transformation is U_ℓ^{-1}.

In thin bootstrapping [CH18], the input plaintext of the coefficient-to-slot transformation is an element of $\mathcal{R}_{p^e}$ rather than $\mathcal{R}'_{p^e}$. This is because it follows directly after the inner product, which creates an encryption of a noisy plaintext in the larger cyclotomic ring. Consequently, we first need to map the plaintext to $\mathcal{R}'_{p^e}$, which is done by removing redundant coefficients. This can be easily achieved with the trace method from Section 3.2.1 if we fold an additional factor of $d^{-1} \pmod{p^e}$ in U_ℓ^{-1}. In summary, the transformation has two steps: (1) evaluate the trace of $\mathcal{R}/\mathcal{R}'$ and (2) multiplication by $(d \cdot U_\ell)^{-1}$.

4.3 New Method for $p = 3 \pmod 4$

We will identify the slots of a plaintext with a vector in E^ℓ. This is done by "flattening" the representative set as $S = \{1, 5, \ldots, 5^{\ell_1 - 1}\}$ and filling the plaintext in Equation (3). Similarly to above, we use the notation $\zeta_{m,i} = \zeta_m^{h_i}$, where h_i is the i-th element of S. We also define $\zeta_4 = \zeta_m^{m/4}$ and $E' = \mathbb{Z}_{p^e}[\zeta_4]$.

4.3.1 Fully Packed Slots

In the slot-to-coefficient transformation, we start from a plaintext $\boldsymbol{m}$ that encodes

$$\vec{m} = \left(\sum_{j=0}^{d/2-1} (m_{i,j} + n_{i,j} \cdot \zeta_4) \cdot \zeta_{m,i}^j \right)_{0 \le i < \ell}$$

in the slots. Note that the encoding is done with respect to $\zeta_{m,i}^j$ instead of ζ_m^j so that we can use powers of X as the packing constants later. The goal is to map the elements $m_{i,j}$ and $n_{i,j}$ to the coefficients of a new plaintext $\boldsymbol{m}'''$. This can be done in three steps:

1. Perform a slot-wise linear transformation M that maps $\zeta_{m,i}^j \mapsto \zeta_m^j$ for $0 \le j < d/2$ in slot i and extends by E'-linearity. Note that this transformation depends on i and is thus different for each slot. After this step, the plaintext encodes the slot-vector

$$\vec{m}' = \left(\sum_{j=0}^{d/2-1} (m_{i,j} + n_{i,j} \cdot \zeta_4) \cdot \zeta_m^j \right)_{0 \le i < \ell} .$$

2. Multiply the slot-vector by (a column-permuted version of) the decoding matrix

$$U_\ell = \left(\zeta_{m,i}^{d \cdot j/2} \right)_{0 \le i,j < \ell} ,$$

where the matrix above is specified by the entries in its i-th row and j-th column. Observe that $\zeta_m^{d/2} \in E'$, which implies $U_\ell \in (E')^{\ell \times \ell}$. After this step, the plaintext encodes the slot-vector

$$\overrightarrow{m}'' = \left(\sum_{k=0}^{\ell-1} \sum_{j=0}^{d/2-1} (m_{k,j} + n_{k,j} \cdot \zeta_4) \cdot \zeta_m^j \cdot \zeta_{m,i}^{d \cdot k/2} \right)_{0 \le i < \ell} .$$

A column-permuted version of U_ℓ will simply map the elements $m_{i,j}$ and $n_{i,j}$ to a different order of the coefficients.

3. Perform the linear transformation M^{-1} that maps $\zeta_m^j \mapsto \zeta_{m,i}^j$ for $0 \le j < d/2$ in slot i. After this step, the plaintext encodes the slot-vector

$$\overrightarrow{m}''' = \left(\sum_{k=0}^{\ell-1} \sum_{j=0}^{d/2-1} (m_{k,j} + n_{k,j} \cdot \zeta_4) \cdot \zeta_{m,i}^{j+d \cdot k/2} \right)_{0 \le i < \ell} ,$$

which corresponds to the plaintext

$$\boldsymbol{m}''' = \sum_{k=0}^{\ell-1} \sum_{j=0}^{d/2-1} (m_{k,j} + n_{k,j} \cdot X^{m/4}) \cdot X^{j+d \cdot k/2}.$$

Here we used the observation that M^{-1} does not act on the entries of U_ℓ (i.e., powers of $\zeta_m^{d/2}$) due to E'-linearity.

We emphasize that the slot-to-coefficient transformation is only E'-linear and not E-linear in general, which is why step 1 and step 3 are required (step 2 alone is E-linear). The inverse map (coefficient-to-slot transformation) is given by $M^{-1} U_\ell^{-1} M$ instead of $M^{-1} U_\ell M$.

Matrix decomposition. The matrix from step 2 can be decomposed into a product of sparse matrices. To do this, we use the column-permuted version $U_\ell = S_\ell$, which is defined in Equation (5) and generated by the root of unity $\zeta_m^{d/2}$. Observe that S_ℓ can be further decomposed using Lemma 2. As a result, we can write it as a product of $\log_2(\ell)$ sparse matrices, each of which acts on only one dimension of S (in the alternative interpretation where each dimension has size 2). The leftmost factor of this product will act on the first dimension and the rightmost factor on the last dimension.

Unpacking the slots. The unpacking procedure is similar to the case $p = 1 \pmod 4$. First, we iteratively split the ciphertext in $d/2$ ciphertexts, each one encoding elements from E' in the slots. This is done by homomorphically computing $\boldsymbol{m}_1 = \boldsymbol{m} + \sigma^{d/2^i}(\boldsymbol{m})$ and $\boldsymbol{m}_2 = X^{-2^{i-1}} \cdot (\boldsymbol{m} - \sigma^{d/2^i}(\boldsymbol{m}))$ in level $i = 1, \dots, \log_2(d/2)$ of the iteration, where $\boldsymbol{m}$ loops over all outputs of the previous iteration. Note that multiplication by powers of X does not increase the noise.

After obtaining $d/2$ ciphertexts that encode elements from E', we split each one in two ciphertexts that encode elements from $\mathbb{Z}_{p^e}$. This is done by homomorphically computing $\boldsymbol{m}_1 = \boldsymbol{m} + \sigma(\boldsymbol{m})$ and $\boldsymbol{m}_2 = X^{-m/4} \cdot (\boldsymbol{m} - \sigma(\boldsymbol{m}))$, where $\boldsymbol{m}$ loops over all outputs of the previous step. Again, we can remove the undesired factor of d by merging $d^{-1} \pmod{p^e}$ in the preceding linear transformation.

Repacking the slots. This operation is the inverse of unpacking and can be computed iteratively as $\boldsymbol{m} = \boldsymbol{m}_1/2 + X^{2^{i-1}} \cdot \boldsymbol{m}_2/2$, where $i = \log_2(m/2), \log_2(d/2), \dots, 1$. Note that the input of repacking is indeed given by $\boldsymbol{m}_1/2$ and $\boldsymbol{m}_2/2$ rather than $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$, assuming that the factor of d was removed appropriately before unpacking.

4.3.2 Sparsely Packed Slots

According to Example 1, sparsely packed plaintexts (where the slots contain an element from $\mathbb{Z}_{p^e}$) live in the subring $\mathcal{R}'_{p^e}$ with respect to $m' = 2m/d$. This result can even be extended to slots encoding values in E'. Conversely, it is easy to see that the subring $\mathcal{R}'_{p^e}$ packs elements from E'. As such, both the slot-encoded plaintext $\boldsymbol{m}$ and the coefficient-encoded plaintext $\boldsymbol{m}'''$ pack elements from E'. Step 1 and step 3 of the slot-to-coefficient transformation can now be omitted, because they have no effect on E'. Therefore, the slot-to-coefficient transformation is U_ℓ and the coefficient-to-slot transformation is U_ℓ^{-1}.

In thin bootstrapping [CH18], the input plaintext of the coefficient-to-slot transformation is an element of $\mathcal{R}_{p^e}$ rather than $\mathcal{R}'_{p^e}$. Consequently, we first need to map the plaintext to $\mathcal{R}'_{p^e}$, which is done by removing redundant coefficients. This can be easily achieved with the trace method from Section 3.2.1. However, using the trace is not sufficient in this case, because the slots can still encode elements from E' rather than $\mathbb{Z}_{p^e}$ after multiplication by U_ℓ^{-1} (the trace cannot remove all coefficients $n_{i,j}$). Therefore, we need to post-process the output of the coefficient-to-slot transformation by computing the trace of $E'/\mathbb{Z}_{p^e}$ in each slot (which is done as $\boldsymbol{m} \mapsto \boldsymbol{m} + \sigma(\boldsymbol{m})$). Again, we fold an additional factor of $d^{-1} \pmod{p^e}$ in U_ℓ^{-1} to remove undesired factors of 2, which were introduced during pre-processing and post-processing with the trace. In summary, the transformation has three steps: (1) evaluate the trace of $\mathcal{R}/\mathcal{R}'$, (2) multiplication by $(d \cdot U_\ell)^{-1}$ and (3) evaluate the trace of $E'/\mathbb{Z}_{p^e}$ slot-wise.

5 Complexity Analysis and Comparison to Related Work

This section performs an in-depth complexity analysis of the proposed algorithm, assuming a baby-step/giant-step implementation. Then we explain how the concurrent work due to Ma et al. [MHWW24b], and the prior works due to Halevi and Shoup [HS21] and Chen and Han [CH18] achieve the slot-to-coefficient transformation. Finally, we compare our asymptotic complexity to these related works.

5.1 Complexity Analysis

5.1.1 Fully Packed Slots

We first analyze the cost of evaluating the fully packed slot-to-coefficient transformation (the cost of the inverse map is identical). We break it down in two components:

- Baby-step/giant-step evaluations of size n in a good dimension. These need roughly $2\sqrt{n}$ automorphisms and n plaintext-ciphertext multiplications.
- Baby-step/giant-step evaluations of size n in a bad dimension. These need roughly $3\sqrt{n}$ automorphisms and $2n$ plaintext-ciphertext multiplications.

First, we consider the maps M and M^{-1}, which can be expressed as a sum of d terms using Equation (8). In the case $p = 3 \pmod 4$, both maps are E'-linear and we only need the automorphisms for even v (those correspond to the automorphism group of E/E'), so we have $d/2$ non-zero terms in Equation (8). To simplify notation, we define $c = d$ if $p = 1 \pmod 4$ and $c = d/2$ if $p = 3 \pmod 4$. The complexity corresponds to a baby-step/giant-step evaluation of size c in a good dimension, so it roughly requires $2\sqrt{c}$ automorphisms and c plaintext-ciphertext multiplications. In terms of noise growth, it uses one level of plaintext-ciphertext multiplications.

To implement homomorphic matrix-vector multiplication by U_ℓ, we decompose it as $U_\ell = U_{\ell,1} \cdot \ldots \cdot U_{\ell,T}$. If we denote the number of non-zero entries in every row/column of $U_{\ell,i}$ by L_i, then this decomposition is done under the constraint $\ell = L_1 \cdot \ldots \cdot L_T$. In other words, the sizes L_i of the matrices should multiply to the number of slots. Such a factorization of U_ℓ can be obtained by merging multiple factors of the decomposition – a technique referred to as level collapsing in CKKS [CCS19].

For small values of c, it is advantageous to merge the map M (resp. M^{-1}) with the last (resp. first) factor of U_ℓ to save multiplicative levels. In that case, the slot-to-coefficient transformation is given by

$$(M^{-1}U_{\ell,1}) \cdot U_{\ell,2} \cdot \ldots \cdot U_{\ell,T-1} \cdot (U_{\ell,T}M).$$

Each factor can be implemented with the multidimensional baby-step/giant-step algorithm: the rightmost factor $U_{\ell,T}M$ corresponds to a baby-step/giant-step evaluation of size cL_T in a bad dimension, so it roughly requires $3\sqrt{cL_T}$ automorphisms and $2cL_T$ plaintext-ciphertext multiplications; the intermediate factors $U_{\ell,i}$ correspond to a baby-step/giant-step evaluation of size L_i in a bad dimension, so they roughly require $3\sqrt{L_i}$ automorphisms and $2L_i$ plaintext-ciphertext multiplications; and the leftmost factor $M^{-1}U_{\ell,1}$ corresponds to a baby-step/giant-step evaluation of size $c \cdot L_1$ in a good dimension, so it roughly requires $2\sqrt{cL_1}$ automorphisms and cL_1 plaintext-ciphertext multiplications. In terms of noise growth, the algorithm uses T levels of plaintext-ciphertext multiplications. However, a larger number of factors T reduces the size L_i of the factors since the product is fixed. Therefore, there is a convenient trade-off between computational cost and multiplicative depth. We summarize the number of operations and levels in the top half of Table 2.

Unpacking and repacking. The cost of unpacking is dominated by $d-1$ automorphisms; it also requires $d-1$ plaintext-ciphertext multiplications and $2(d-1)$ additions. The cost of repacking is given by $d-1$ plaintext-ciphertext multiplications and equally many additions. Repacking requires no key switching and is therefore cheaper than unpacking. Unpacking and repacking use no multiplicative levels, because all constants are powers of X. We summarize the number of operations and levels in the top half of Table 2.

5.1.2 Sparsely Packed Slots

The maps M and M^{-1} can be omitted for sparsely packed slots, so we only need to evaluate $U_\ell = U_{\ell,1} \cdot \ldots \cdot U_{\ell,T}$. Each factor can again be implemented with the multidimensional baby-step/giant-step algorithm: the factors $U_{\ell,i}$ for $i \geq 2$ correspond to a baby-step/giant-step evaluation of size L_i in a bad dimension, so they roughly require $3\sqrt{L_i}$ automorphisms and $2L_i$ plaintext-ciphertext multiplications; and the leftmost factor $U_{\ell,1}$ corresponds to a baby-step/giant-step evaluation of size L_1 in a good dimension, so it roughly requires $2\sqrt{L_1}$ automorphisms and L_1 plaintext-ciphertext multiplications. In terms of noise growth, the algorithm uses T levels of plaintext-ciphertext multiplications, which brings the same trade-off between computational cost and multiplicative levels as previously. Evaluating the trace(s) for the coefficient-to-slot transformation can be done with $\log_2(d)$ automorphisms and additions, and no multiplicative levels. We summarize the number of operations and levels in the bottom half of Table 2.

5.1.3 Asymptotic Analysis

If we optimize the linear transformations for efficiency and decompose to the maximum number of stages, then it has $\log_2(\ell)$ stages of size 2. In the fully packed variant, there are 2 additional stages of size c. Therefore, the total number of FHE operations is asymptotically $O(c + \log_2(\ell))$ for fully packed slots and $O(\log_2(\ell))$ for sparsely packed slots.

Note that the number of FHE operations is asymptotically linear in c (and hence in the slot width d), which appears inherent to the problem. In any case, the unpacking procedure splits the ciphertext in d components; as such, there is not much to gain by further reducing the complexity of the linear transformations to a sublinear function of d. As a result of this limitation, one needs to consider parameter sets with small d when

Table 2: Algorithmic complexity of the proposed algorithms

	# multiplications			# automorphisms			# levels
SlotToCoeff	$M^{-1}U_{\ell,1}$	$U_{\ell,i}$	$U_{\ell,T}M$	$M^{-1}U_{\ell,1}$	$U_{\ell,i}$	$U_{\ell,T}M$	T
	cL_1	$2L_i$	$2cL_T$	$2\sqrt{cL_1}$	$3\sqrt{L_i}$	$3\sqrt{cL_T}$	
Unpacking	$d-1$			$d-1$			0
Repacking	$d-1$			0			0
SlotToCoeff	$U_{\ell,1}$	$U_{\ell,i}$		$U_{\ell,1}$	$U_{\ell,i}$		T
	L_1	$2L_i$		$2\sqrt{L_1}$	$3\sqrt{L_i}$		
Trace	0			$\log_2(d)$			0

Table 3: Algorithmic complexity of Ma et al. (using Bruun-style FFT if $p = 3 \pmod 4$)

	# multiplications		# automorphisms		# levels
SlotToCoeff	$M^{-1}U_{\ell,1}$	$U_{\ell,i}$	$M^{-1}U_{\ell,1}$	$U_{\ell,i}$	T
	dL_1	$2dL_i/c$	$2\sqrt{dL_1}$	$3\sqrt{dL_i/c}$	
Unpacking	d^2		$d-1$		1
Repacking	d		0		1
SlotToCoeff	$U_{\ell,1}$	$U_{\ell,i}$	$U_{\ell,1}$	$U_{\ell,i}$	T
	dL_1/c	$2dL_i/c$	$2\sqrt{dL_1/c}$	$3\sqrt{dL_i/c}$	
Trace	0		$\log_2(d)$		0

optimizing for fast execution time. This choice is often preferred at the application level as well, because it corresponds to a large number of small-domain plaintext slots.

5.2 Comparison to Concurrent Work

Independently from this work, Ma et al. [MHWW24b] developed a similar technique to evaluate the slot-to-coefficient transformation. Table 3 summarizes their complexity in a format comparable to Table 2. Our technique has a couple of advantages over theirs:

- In the slot representation, they use the normal element encoding from Halevi and Shoup. As such, their method cannot integrate the multiplicative level of unpacking and repacking in the linear transformations.
- For $p = 3 \pmod 4$, their stages are $\mathbb{Z}_{p^e}$-linear and not E'-linear, so the cost truly depends on d rather than $c = d/2$. There are two different algorithms for this case:
  1. Their Bruun-style decomposition uses a more complicated FFT butterfly, which is roughly twice the size of a standard FFT butterfly. Each stage is therefore two times more costly than our method.
  2. Their radix-2 decomposition uses a $\mathbb{Z}_{p^e}$-linear transformation in each stage. This results in an asymptotic cost of $O(d \cdot \log_2(\ell))$ FHE operations for fully packed slots, compared to $O(c + \log_2(\ell))$ for our routine.

5.3 Comparison to Prior Work

5.3.1 Method of Halevi and Shoup

Halevi and Shoup [HS21] implement the slot-to-coefficient transformation for non-power-of-two cyclotomic rings. Their procedure takes a cyclotomic index m that splits into pairwise coprime factors as $m = m_1 \cdot \ldots \cdot m_t$. Then they represent the collection of plaintext slots as a t-dimensional hypercube by algebraically decomposing the quotient group as $\mathbb{Z}_m^*/\langle p \rangle = \mathbb{Z}_{m_1}^*/\langle p \rangle \times \mathbb{Z}_{m_2} \times \ldots \times \mathbb{Z}_{m_t}$. They show that the transformation can be implemented by evaluating one subtransformation in each of the t dimensions. The sizes of these transformations correspond to the values of $\varphi(m_i)$ (note that m_1 induces a $\mathbb{Z}_{p^e}$-linear transformation and the other factors induce an E-linear transformation). Thus the cost is minimized by splitting m into many smaller factors.

There are a few noteworthy differences between Halevi/Shoup and our method:

- Their method is not applicable to power-of-two cyclotomics for two reasons: (1) the factorization of m needs to be pairwise coprime, which means that there can only be one factor if m is a power of two; (2) they require that each group in the composition is cyclic, which is not guaranteed if m is a power of two.
- In the coefficient representation, they encode their numbers in the so-called "powerful basis" of the cyclotomic ring [LPR13]. This basis naturally arises from tensoring cyclotomic rings of pairwise coprime indices. For power-of-two cyclotomics, the powerful basis coincides with the standard "power basis" used in our method, so we can ignore this extra level of complexity.
- In the slot representation, they encode their numbers in the so-called "normal basis" generated by a normal element $\theta \in E$. This is done so that they only need to precompute and store d constants for the unpacking procedure instead of d^2. In our method, the unpacking constants are given by powers of X, so they do not need to be precomputed at all. Moreover, the complexity analysis below shows that the performance of our unpacking method is superior to Halevi and Shoup.

Unpacking and repacking. The unpacking routine of Halevi/Shoup computes $d - 1$ automorphisms of the input and then takes d weighted linear combinations of the results. It therefore requires d automorphisms and d^2 plaintext-ciphertext multiplications. Our method needs only $d - 1$ automorphisms and $d - 1$ plaintext-ciphertext multiplications. Moreover, we do not use any multiplicative levels, whereas Halevi and Shoup use one level.

Our cheaper version of the slot unpacking procedure does not come completely for free. It necessitates the extra map M in the slot-to-coefficient transformation, which is an original contribution of this work. However, this can be implemented with d plaintext-ciphertext multiplications, which is cheaper than d^2 plaintext-ciphertext multiplications in Halevi and Shoup's procedure. Moreover, our method allows more freedom in parameter selection since we can integrate the maps in U_ℓ and U_ℓ^{-1} to save multiplicative levels.

5.3.2 Method of Chen and Han

Chen and Han [CH18] implement the slot-to-coefficient transformation for power-of-two cyclotomic rings. However, they use the trivial method which does not decompose the transformation in smaller-dimensional stages. As a result, it involves homomorphic multiplication by a full $N \times N$-matrix (in the fully packed case) or an $\ell \times \ell$-matrix (in the sparsely packed case). The asymptotic number of FHE operations is linear in the lattice dimension N or the number of slots ℓ.

6 Implementation and Results

This section discusses the implementation and experimental results based on BFV. Our implementation is built on the framework provided by Okada et al. [OPP23]. All experiments were run with Microsoft SEAL version 4.1 [SEA23] on an Intel® Xeon® E5-2630 v2 CPU with 128 GB memory and Ubuntu 18.04.6 LTS, in a single thread. For all parameter sets, we took lattice dimension $N = 2^{15}$ and ciphertext modulus $q \approx 2^{1080}$. This gives slightly more than 100 bits of security according to the lattice estimator [APS15].

Section 6.1 gives results for the individual linear transformations and studies the trade-off between execution time and multiplicative depth. Section 6.2 studies the performance of a fully packed bootstrapping operation for 2^{15} elements of $\mathrm{GF}(2^{16}+1)$. In all experiments, the leftmost column of the table is considered the baseline, because no decomposition is applied. The baseline numbers were estimated by carefully timing the individual building blocks and multiplying by the number of operations. This is done because an enormous amount of precomputed constants is required, which do not even fit in the 128 GB physical memory of our machine. Computing those constants on the fly or loading them from disk comes with a performance penalty, and would therefore make the comparison unfair.

6.1 Performance of the Linear Transformations

Table 4 and Table 5 show experimental results for fully packed and sparsely packed slots respectively. Both tables use $N = 2^{15}$, $p = 2^{13} - 1$ and $e = 1$, resulting in $d = 8$. The leftmost column in both tables represents the baseline algorithm (it has the same complexity as Chen/Han). This means that no decomposition is applied, and unpacking and repacking are done with respect to the normal element encoding. Recall that the first stage and last stage of the fully packed transformation are c times larger than the regular ones. As such, the decomposition sizes were chosen differently in Table 4 and Table 5 in order to balance the execution time of the stages.

As predicted by the theoretical analysis, the tables indicate a trade-off between runtime and multiplicative depth: decomposing in more stages is faster, but also increases the consumed noise budget. For example, the SlotToCoeff row in the last column of Table 4 is roughly 44× faster than the baseline, but has 3× more noise growth. In practice, the optimal number of stages is application-dependent and can best be found via experimentation.

Observe that the first column and the second column of Table 4 use equally many levels: the first column uses one level for SlotToCoeff and one level for unpacking/repacking; the second column uses two levels for SlotToCoeff. Yet the second column reports 16× faster runtime than the first one. This is because we integrated the multiplicative level of unpacking/repacking into SlotToCoeff and CoeffToSlot, which gives us more freedom to choose optimal decomposition parameters.

Table 4: Results for fully packed slots using $N = 2^{15}$, $p = 2^{13} - 1$ and $e = 1$

	$L_1 \cdot \ldots \cdot L_T$	Baseline	$2^6 \cdot 2^6$	$2^4 \cdot 2^5 \cdot 2^3$	$2^2 \cdot 2^4 \cdot 2^4 \cdot 2^2$
Noise (bits)	SlotToCoeff	25	43	60	76
	Unpacking	20	2	2	2
	Repacking	20	1	1	1
Execution time (sec)	SlotToCoeff	730	45.1	22.8	16.6
	Unpacking	3.0	2.2	2.2	2.2
	Repacking	0.1	0.1	0.1	0.1

Table 5: Results for sparsely packed slots using $N = 2^{15}$, $p = 2^{13} - 1$ and $e = 1$

	$L_1 \cdot \ldots \cdot L_T$	Baseline	$2^6 \cdot 2^6$	$2^4 \cdot 2^4 \cdot 2^4$	$2^3 \cdot 2^3 \cdot 2^3 \cdot 2^3$
Noise (bits)	SlotToCoeff	24	42	58	74
	Trace	2	2	2	2
Execution time (sec)	SlotToCoeff	123	14.5	9.2	8.6
	Trace	1.2	1.2	1.2	1.2

6.2 Fully Packed Bootstrapping Application

The goal of this section is to demonstrate the benefits of our linear transformations by means of a fully packed bootstrapping application. Our implementation uses the standard workflow of BFV bootstrapping (see for example Fig. 1 by Chen and Han [CH18] or Fig. 3 by Geelen and Vercauteren [GV23]). Next to the linear transformations, bootstrapping also involves a digit extraction procedure. In our implementation, digit extraction is done with the improved method for large values of p due to Ma et al. [MHWW24a]. Furthermore, we describe an efficient way to construct the digit removal polynomials defined by Ma et al. in Appendix A. The noise cut-off parameter from the appendix is set to $B = 255$.

The large-p digit extraction procedure from Ma et al. requires that the input ciphertext has some remaining noise budget before the next bootstrapping is applied. Since SEAL does not cope with this, we manually subtracted 15 bits from the initial and remaining noise budget in Table 6. As such, the input noise of bootstrapping is at most 1 bit, and the performance of bootstrapping is dominated by modulus switching noise.

Our method can be further optimized with the sparse secret encapsulation technique from Bossuat et al. [BTH22]. However, the current version of SEAL only supports ternary uniform secret keys. As such, it is impossible to implement sparse secret encapsulation.

Table 6 shows experimental results for $N = 2^{15}$ and $p = 2^{16} + 1$. This parameter set corresponds to $d = 1$, so we pack 2^{15} elements from $\mathrm{GF}(2^{16} + 1)$. Note that the fully and sparsely packed transformations coincide for $d = 1$, which implies that unpacking and repacking are not required for this benchmark. The leftmost column is again the baseline and has the same complexity as Chen/Han. Finally, we note that the coefficient-to-slot transformation uses $e = 2$, whereas the slot-to-coefficient transformation uses $e = 1$.

The amortized improvements (i.e., taking into account both remaining noise budget and execution time) are equal to

$$\frac{1325.7 \cdot 347}{105 \cdot 400} \approx 11\times \qquad \text{and} \qquad \frac{1325.7 \cdot 294}{82.9 \cdot 400} \approx 12\times$$

for the second and third column respectively. This can be fully contributed to the faster coefficient-to-slot and slot-to-coefficient transformations. Note that the remaining noise budget of our procedure is slightly less than the baseline. This is because we consume one additional multiplicative level per transformation for the second column, and two additional levels for the third column.

Table 6: Results for fully packed bootstrapping using $N = 2^{15}$ and $p = 2^{16} + 1$

	$L_1 \cdot \ldots \cdot L_T$	Baseline	$2^8 \cdot 2^7$	$2^5 \cdot 2^5 \cdot 2^5$
Noise (bits)	Initial	942	942	942
	CoeffToSlot	44	79	114
	Digit extract	468	468	468
	SlotToCoeff	30	48	66
	Remaining	400	347	294
Execution time (sec)	Inner product	0.2	0.2	0.2
	CoeffToSlot	637	26.2	15.1
	Digit extract	52.5	52.5	52.5
	SlotToCoeff	636	26.1	15.1
	Total	1325.7	105	82.9

7 Conclusion

This paper derived explicit FFT-based formulas to implement the slot-to-coefficient and coefficient-to-slot transformations in BGV and BFV. The major application of the linear transformations is inarguably the bootstrapping procedure, for which we obtained 12× faster results than prior work. Interesting future work would be to apply the proposed techniques to other applications as well.

Acknowledgements

This work was supported by CyberSecurity Research Flanders with reference number VR20192203. In addition, this work is supported in part by the European Commission through the Horizon 2020 research and innovation program Belfort ERC Advanced Grant 101020005 and by the Defence Advanced Research Projects Agency (DARPA) under contract No. HR001120S0032 DARPA DPRIVE BASALISC. Robin Geelen is funded by Research Foundation – Flanders (FWO) under a PhD Fellowship fundamental research (project number 1162123N). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the ERC, DARPA, the U.S. Government, the European Union, CyberSecurity Research Flanders or the FWO. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein. The author would also like to thank Jiayi Kang and Frederik Vercauteren for useful discussions.

References

[AP13] Jacob Alperin-Sheriff and Chris Peikert. Practical bootstrapping in quasilinear time. In *CRYPTO (1)*, volume 8042 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2013.

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

[BCK$^+$23] Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, Jai Hyun Park, and Damien Stehlé. HERMES: efficient ring packing using MLWE ciphertexts and application to transciphering. In *CRYPTO (4)*, volume 14084 of *Lecture Notes in Computer Science*, pages 37–69. Springer, 2023.

[BGGJ20] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. CHIMERA: combining ring-lwe-based fully homomorphic encryption schemes. *J. Math. Cryptol.*, 14(1):316–338, 2020.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3):13:1–13:36, 2014.

[BLZ23] Matvey Borodin, Ethan Liu, and Justin Zhang. Results on vanishing polynomials and polynomial root counting with relevant technological applications. In *2023 IEEE MIT Undergraduate Research Technology Conference (URTC)*, pages 1–5. IEEE, 2023.

[BMTH21] Jean-Philippe Bossuat, Christian Mouchet, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In *EUROCRYPT (1)*, volume 12696 of *Lecture Notes in Computer Science*, pages 587–617. Springer, 2021.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012.

[BTH22] Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation. In *ACNS*, volume 13269 of *Lecture Notes in Computer Science*, pages 521–541. Springer, 2022.

[CCLS19] Jung Hee Cheon, Hyeongmin Choe, Donghwan Lee, and Yongha Son. Faster linear transformations in HElib, revisited. *IEEE Access*, 7:50595–50604, 2019.

[CCS19] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. In *EUROCRYPT (2)*, volume 11477 of *Lecture Notes in Computer Science*, pages 34–54. Springer, 2019.

[CH18] Hao Chen and Kyoohyung Han. Homomorphic lower digits removal and improved FHE bootstrapping. In *EUROCRYPT (1)*, volume 10820 of *Lecture Notes in Computer Science*, pages 315–337. Springer, 2018.

[CHK$^+$18] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *EUROCRYPT (1)*, volume 10820 of *Lecture Notes in Computer Science*, pages 360–384. Springer, 2018.

[CHK$^+$21] Jihoon Cho, Jincheol Ha, Seongkwang Kim, ByeongHak Lee, Joohee Lee, Jooyoung Lee, Dukjae Moon, and Hyojin Yoon. Transciphering framework for approximate homomorphic encryption. In *ASIACRYPT (3)*, volume 13092 of *Lecture Notes in Computer Science*, pages 640–669. Springer, 2021.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT (1)*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012. URL: https://eprint.iacr.org/2012/144.

[Gau86] Carl Friedrich. Gauss. *Disquisitiones arithmeticae*. Springer, Berlin, 1986.

[GHS12a] Craig Gentry, Shai Halevi, and Nigel P. Smart. Better bootstrapping in fully homomorphic encryption. In *Public Key Cryptography*, volume 7293 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2012.

[GHS12b] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012.

[GIKV23] Robin Geelen, Ilia Iliashenko, Jiayi Kang, and Frederik Vercauteren. On polynomial functions modulo p^e and faster bootstrapping for homomorphic encryption. In *EUROCRYPT (3)*, volume 14006 of *Lecture Notes in Computer Science*, pages 257–286. Springer, 2023.

[GV23] Robin Geelen and Frederik Vercauteren. Bootstrapping for BGV and BFV revisited. *J. Cryptol.*, 36(2):12, 2023.

[HHC19] Kyoohyung Han, Minki Hhan, and Jung Hee Cheon. Improved homomorphic discrete fourier transforms and FHE bootstrapping. *IEEE Access*, 7:57361–57370, 2019.

[HS18] Shai Halevi and Victor Shoup. Faster homomorphic linear transformations in helib. In *CRYPTO (1)*, volume 10991 of *Lecture Notes in Computer Science*, pages 93–120. Springer, 2018.

[HS20] Shai Halevi and Victor Shoup. Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive, Paper 2020/1481, 2020. URL: https://eprint.iacr.org/2020/1481.

[HS21] Shai Halevi and Victor Shoup. Bootstrapping for helib. *J. Cryptol.*, 34(1):7, 2021.

[KPZ21] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. Revisiting homomorphic encryption schemes for finite fields. In *ASIACRYPT (3)*, volume 13092 of *Lecture Notes in Computer Science*, pages 608–639. Springer, 2021.

[LHH$^+$21] Wen-jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. PEGASUS: bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *SP*, pages 1057–1073. IEEE, 2021.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-lwe cryptography. In *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 35–54. Springer, 2013.

[LS18] Vadim Lyubashevsky and Gregor Seiler. Short, invertible elements in partially splitting cyclotomic rings and applications to lattice-based zero-knowledge proofs. In *EUROCRYPT (1)*, volume 10820 of *Lecture Notes in Computer Science*, pages 204–224. Springer, 2018.

[LW23a] Feng-Hao Liu and Han Wang. Batch bootstrapping I: - A new framework for SIMD bootstrapping in polynomial modulus. In *EUROCRYPT (3)*, volume 14006 of *Lecture Notes in Computer Science*, pages 321–352. Springer, 2023.

[LW23b] Zeyu Liu and Yunhao Wang. Amortized functional bootstrapping in less than 7 ms, with $\tilde{O}(1)$ polynomial multiplications. In *ASIACRYPT (6)*, volume 14443 of *Lecture Notes in Computer Science*, pages 101–132. Springer, 2023.

[MHWW24a] Shihe Ma, Tairong Huang, Anyu Wang, and Xiaoyun Wang. Accelerating BGV bootstrapping for large p using null polynomials over $\mathbb{Z}_{p^e}$. In *EUROCRYPT (2)*, volume 14652 of *Lecture Notes in Computer Science*, pages 403–432. Springer, 2024.

[MHWW24b] Shihe Ma, Tairong Huang, Anyu Wang, and Xiaoyun Wang. Faster BGV bootstrapping for power-of-two cyclotomics through homomorphic NTT. Cryptology ePrint Archive, Paper 2024/164, 2024. URL: https://eprint.iacr.org/2024/164.

[OPP23] Hiroki Okada, Rachel Player, and Simon Pohmann. Homomorphic polynomial evaluation using galois structure and applications to BFV bootstrapping. In *ASIACRYPT (6)*, volume 14443 of *Lecture Notes in Computer Science*, pages 69–100. Springer, 2023.

[SEA23] Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, January 2023. Microsoft Research, Redmond, WA.

[SV14] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *Des. Codes Cryptogr.*, 71(1):57–81, 2014.

A Efficient Computation of the Digit Removal Polynomial

This appendix describes a conceptually simpler and more efficient way to construct the polynomials defined by Ma et al. [MHWW24a]. In the bootstrapping application – where p is an odd prime and $e = 2$ – digit extraction boils down to a single evaluation of the *lowest digit removal polynomial* $H(X)$. This polynomial satisfies

$$H(a) = p \cdot \left\lfloor \frac{a}{p} \right\rceil \pmod{p^2} \tag{10}$$

for all $a \in S$ where

$$S = \{\, c \cdot p + b \mid b, c \in \mathbb{Z} \text{ and } -B \le b \le B \,\}.$$

The original construction starts by computing a polynomial that satisfies Equation (10) for all $a \in \mathbb{Z}$, and then it performs reduction modulo a *null polynomial.* However, the computational cost of obtaining $H(X)$ in this method scales poorly for larger p since the intermediate polynomial has degree p.

In our implementation, we precompute the lowest digit removal polynomial in a different way. We first define

$$P(X) = \prod_{i=-B}^{B} (X - i).$$

Then we consider the ideal $\mathcal{I} = (P(X)^2, pP(X), p^2)$. Note that each $O(X) \in \mathcal{I}$ satisfies $O(a) = 0 \pmod{p^2}$ for all $a \in S$. In the literature, such polynomials have been called null polynomials over S [GIKV23] or local null polynomials [MHWW24a]. Since we are only interested in the set S, we can compute $H(X)$ over $\mathbb{Z}[X]/\mathcal{I}$ (in fact, this ring is isomorphic to the ring of polyfunctions modulo p^2 over S [GIKV23, BLZ23]).

We now observe that the so-called *digit extraction function*

$$g \colon \mathbb{Z}_{p^2} \to \mathbb{Z}_{p^2} \colon a \mapsto a - p \cdot \left\lfloor \frac{a}{p} \right\rceil \tag{11}$$

has period p. Following Equation (13) by Geelen et al. [GIKV23], a representation of this function is obtained analytically as

$$G(X) = \sum_{i=-r}^{r} i \cdot (1 - (X - i)^{p \cdot (p-1)}),$$

where $r = (p - 1)/2$. Since we are only interested in the set S, the polynomial $H(X)$ is computed by subtracting $G(X)$ from X and cutting off the summation at B instead of r:

$$H(X) = X - \sum_{i=-B}^{B} i \cdot (1 - (X - i)^{p \cdot (p-1)}).$$

This expression is computed over the ring $\mathbb{Z}[X]/\mathcal{I}$. As such, raising elements to the power of $p \cdot (p - 1)$ can be done efficiently with square-and-multiply in that ring. Finally, we note that the function from Equation (11) is odd. Therefore, the obtained polynomial has only odd-exponent terms after reduction modulo $\mathcal{I}$ [GIKV23].