


Better GBFV Bootstrapping and Faster Encrypted Edit Distance Computation

Robin Geelen  and Frederik Vercauteren 

COSIC, ESAT, KU Leuven, Belgium
`firstname.lastname@esat.kuleuven.be`

Abstract. We propose a new iterative method to convert a ciphertext from the Generalized BFV (GBFV) to the regular BFV scheme. In particular, our conversion starts from an encrypted plaintext that lives in a large cyclotomic ring modulo a small-norm polynomial $t(x)$, and gradually changes the encoding to a smaller cyclotomic ring modulo a larger integer p . Previously, only a trivial conversion method was known, which did not change the underlying cyclotomic ring.

Using our improved conversion algorithm, we can bootstrap the GBFV scheme almost natively, in the sense that only a very small fraction of the operations is computed inside regular BFV. Specifically, we evaluate (an adapted version of) the slot-to-coefficient transformation entirely in the GBFV scheme, whereas the previous best method used the BFV scheme for that transformation. This insight allows us to bootstrap either with less noise growth, or much faster than the state-of-the-art.

We implement our new bootstrapping in Microsoft SEAL. Our experiments show that, for the same remaining noise budget, our bootstrapping runs in only 800 ms when working with ciphertexts containing 1024 slots over \mathbb{F}_p with $p = 2^{16} + 1$. This is $1.6\times$ faster than the state-of-the-art.

Finally, we use our improved GBFV bootstrapping in an application that computes an encrypted edit distance. Compared to the recent TFHE-based Leuvenstein algorithm, our GBFV version is almost two orders of magnitude faster in the amortized sense.

Keywords: Fully homomorphic encryption · GBFV · Bootstrapping · Edit distance.

1 Introduction

Applications of fully homomorphic encryption (FHE) often rely on parallel data processing. To accommodate this parallelism, many state-of-the-art FHE schemes such as BGV [5], BFV [4,15] and CKKS [12] have a built-in mechanism for “single instruction, multiple data” (SIMD) operations. In typical settings, these schemes can pack thousands of data elements in a SIMD vector. Basic arithmetic operations are then evaluated entry-wise on these huge vectors.

While the above SIMD schemes can pack a massive amount of data, certain applications (such as encrypted edit distance computation) only need a moderate

packing capacity. Of course, it is always possible to use a subset of the available SIMD slots, but this is quite wasteful in practice because it leaves a large portion of the message space unused. Recently, Geelen and Vercauteren [17] proposed a generalized version of the BFV scheme - called GBFV - which brings a better solution to this problem.¹ Loosely speaking, the GBFV scheme decreases the size of the SIMD vector by some *small* integer factor α , resulting in a scheme that can be up to α times as efficient as BFV. This means that either the latency can be reduced, or the computational capacity increased, almost by a factor of α . In other words, the GBFV scheme trades packing for efficiency.

Similarly to other FHE schemes, GBFV has a bootstrapping procedure to evaluate circuits of arbitrary size. In fact, the current bootstrapping algorithm can be understood as a hybrid solution: its last step (called *digit removal*) is evaluated using GBFV directly; its first step (called *noisy expansion*) is instantiated in the BFV domain; connecting both steps is achieved via a conversion routine between GBFV and regular BFV. However, a major drawback of this hybrid approach is that it hurts the performance. In particular, the BFV scheme uses much more computational budget than GBFV, so the cost of bootstrapping becomes completely dominated by the noisy expansion step. Therefore, further improvements of this step are required to reduce the computational cost.

1.1 Contributions

The main observation of this paper is that noisy expansion does not use the available plaintext space efficiently. Since this step is evaluated in the BFV domain, only a $1/\alpha$ -fraction of the slots contain relevant information and the others contain garbage. One solution to this problem is batch bootstrapping [17], but this is less interesting from a software engineering point of view, because multiple ciphertexts need to be collected (which may not even be available depending on the application) and then bootstrapped simultaneously. Therefore, we propose a bootstrapping algorithm for individual ciphertexts that makes better use of the available space by staying much longer in the GBFV domain. In particular, we contribute to the state-of-the-art as follows:

- We propose an improved noisy expansion step in which *almost all* operations are evaluated inside GBFV. As a result, we either retain more computational budget, or can bootstrap faster than previous work. The core of our algorithm is a new subroutine that homomorphically converts between two isomorphic rings \mathcal{R}'_t and \mathcal{R}''_p with different encoding, and where \mathcal{R}'' is a subring of \mathcal{R}' . This is an iterative procedure consisting of $\min(\log_2(n''), \log_2(n'/n''))$ steps, where n' and n'' are the vector dimensions of \mathcal{R}' and \mathcal{R}'' respectively. The previous best GBFV bootstrapping algorithm used a trivial conversion from \mathcal{R}'_t to \mathcal{R}''_p , instead of \mathcal{R}''_p . Our implementation in Microsoft SEAL shows that we can bootstrap a ciphertext of 4096 slots in only a second, while still having 6 remaining multiplicative levels.²

¹ A similar solution in leveled mode was proposed independently by Cha et al. [6].

² See <https://github.com/KULEuven-COSIC/Bootstrapping-BGV-BFV/tree/traces>.

- Using our improved bootstrapping algorithm, we present a GBFV-tailored algorithm to homomorphically compute the Levenshtein distance of two encrypted strings. The Levenshtein distance computes the minimal number of deletions, insertions and substitutions that turns the first string into the second. Measuring amortized performance, our GBFV version is approximately two orders of magnitude faster than the state-of-the-art solution, called Levenshtein [21], which takes a TFHE-based approach.

2 Preliminaries

Generalized BFV (GBFV) is a relatively new FHE scheme that was proposed independently by Geelen and Vercauteren [17], and by Cha et al. [6] for two different applications (respectively low-latency bootstrapping and the offline phase of SPDZ). This section recaps the aspects of GBFV that are relevant to us.

2.1 Notations

This paper uses standard power-of-two cyclotomics, which are more convenient to implement than the somewhat more convoluted non-power-of-two cyclotomics. All plaintext data lives in the $2n$ -th cyclotomic ring $\mathcal{R} = \mathbb{Z}[x]/(x^n + 1)$, and we will also use its subrings $\mathcal{R}' = \mathbb{Z}[x^{n/n'}]/(x^n + 1)$ and $\mathcal{R}'' = \mathbb{Z}[x^{n/n''}]/(x^n + 1)$. We call \mathcal{R} the *R-LWE ring* because it is used to generate samples from the ring learning with errors distribution (hence it determines the security level of our homomorphic encryption scheme). The $2n'$ -th cyclotomic ring \mathcal{R}' is called the *GBFV ring*, and the $2n''$ -th cyclotomic ring \mathcal{R}'' is called the *BFV ring*. Those subrings are embedded in \mathcal{R} , and they will encode GBFV and BFV plaintexts respectively. We can apply ring automorphisms $\sigma_i: x \mapsto x^i$ to elements of \mathcal{R} for any odd integer $i \in \mathbb{Z}_{2n}^\times$. It is well known that the automorphism group has rank two and can be decomposed as $\langle \sigma_{-1} \rangle \times \langle \sigma_5 \rangle$.

We consider a modulus $T = T(x) \in \mathcal{R}$ (not necessarily a scalar) and define the quotient ring $\mathcal{R}_T = \mathcal{R}/T\mathcal{R}$. Elements of \mathcal{R} or \mathcal{R}_T are written as $\mathbf{m} = m(x)$, and those will correspond to the plaintexts of our encryption scheme. We also define $p \in \mathbb{Z}$ as the smallest positive integer in $t\mathcal{R}$, with $t = t(y) \in \mathcal{R}'$ the initial plaintext modulus. Here we substituted the indeterminate $y = x^{n/n'}$ for ease of notation. The integer modulus p is constant during the linear transformations, but the modulus T gradually changes from $T = t(y)$ when working in the GBFV ring, up to $T = p$ when working in the BFV ring.

2.2 The GBFV Scheme

The GBFV scheme generalizes both BFV [4,15] and CLPX [8]. More specifically, the modulus T is a scalar value in BFV, a linear polynomial in CLPX and an arbitrary non-zero element in GBFV. Since GBFV is the most general notion, we omit the description of BFV and CLPX.

Homomorphic Operations. We now define all relevant GBFV subroutines, skipping key generation, encryption and decryption. Encryption is carried out in the R-LWE ring \mathcal{R} , though in practice we may encode our data in a cyclotomic subring. Each algorithm is assumed to know the public key, and each ciphertext implicitly includes its own plaintext modulus T .

- $\text{GBFV.Add}(\text{ct}_1, \text{ct}_2) \rightarrow \text{ct}_{\text{add}}$: given ct_1 and ct_2 that encrypt $\mathbf{m}_1, \mathbf{m}_2 \in \mathcal{R}_T$ respectively, compute ct_{add} that encrypts $\mathbf{m}_{\text{add}} = \mathbf{m}_1 + \mathbf{m}_2$.
- $\text{GBFV.Mul}(\text{ct}_1, \text{ct}_2) \rightarrow \text{ct}_{\text{mul}}$: given ct_1 and ct_2 that encrypt $\mathbf{m}_1, \mathbf{m}_2 \in \mathcal{R}_T$ respectively, compute ct_{mul} that encrypts $\mathbf{m}_{\text{mul}} = \mathbf{m}_1 \cdot \mathbf{m}_2$.
- $\text{GBFV.Auto}(\text{ct}, \sigma_i) \rightarrow \text{ct}_{\text{auto}}$: given ct encrypting $\mathbf{m} \in \mathcal{R}_T$, and given σ_i such that $\sigma_i(T) = T$, compute ct_{auto} that encrypts $\mathbf{m}_{\text{auto}} = \sigma_i(\mathbf{m})$.
- $\text{GBFV.ModUp}(\text{ct}, T_2) \rightarrow \text{ct}_{\text{up}}$: given ct encrypting $\mathbf{m} \in \mathcal{R}_{T_1}$, and $T_2 \in T_1\mathcal{R}$, compute ct_{up} that encrypts $\mathbf{m}_{\text{up}} \in \mathcal{R}_{T_2}$ such that $\mathbf{m}_{\text{up}} = \mathbf{m} \pmod{T_1\mathcal{R}}$, and such that $\mathbf{m}_{\text{up}} = 0 \pmod{(T_2/T_1)\mathcal{R}}$.
- $\text{GBFV.ModDown}(\text{ct}, T_2) \rightarrow \text{ct}_{\text{down}}$: given ct encrypting $\mathbf{m} \in \mathcal{R}_{T_1}$, and T_2 such that $T_1 \in T_2\mathcal{R}$, compute ct_{down} that encrypts $\mathbf{m}_{\text{down}} = \mathbf{m} \pmod{T_2\mathcal{R}}$ in the plaintext ring \mathcal{R}_{T_2} .
- $\text{GBFV.Divide}(\text{ct}, T_2) \rightarrow \text{ct}_{\text{div}}$: given ct that encrypts $(T_1/T_2) \cdot \mathbf{m} \in \mathcal{R}_{T_1}$, and also T_2 such that $T_1 \in T_2\mathcal{R}$, compute ct_{div} that encrypts $\mathbf{m} \in \mathcal{R}_{T_2}$.
- $\text{GBFV.InProd}(\text{ct}) \rightarrow \text{ct}_{\text{fresh}}$: given ct that encrypts $\mathbf{m} \in \mathcal{R}_p$, compute ct_{fresh} that encrypts $\mathbf{m}_{\text{fresh}} = p \cdot \mathbf{m} + e \in \mathcal{R}_{p^2}$ for some small error polynomial e .

As usual, addition and multiplication also have a plaintext-ciphertext variant. Observe that BFV-to-GBFV conversion from Geelen and Vercauteren [17] is a special case of **ModDown** where $T_1 = p$. On the other hand, the **ModUp** method is a stronger notion than GBFV-to-BFV conversion [17], because we additionally impose that $\mathbf{m}_{\text{up}} = 0 \pmod{(T_2/T_1)\mathcal{R}}$.

Encoding in Plaintext Slots. In our improved bootstrapping, the plaintext space of the input (and output) ciphertext is

$$\mathcal{R}'_t = \mathbb{Z}[y]/(y^{n'} + 1, t(y)).$$

Intermediately, bootstrapping will also use \mathcal{R}''_p and other characteristic- p rings between those two. We additionally assume that p is a prime number congruent to 1 modulo $2n'$. This restriction gives us base field encoding in \mathbb{F}_p and is most commonly used in (G)BFV to maximize the SIMD parallelism. Bootstrapping for extension fields can still be handled with ring switching [18, 1].

Each ring in our method can pack \mathbb{F}_p -elements, so we need a mechanism to write those slots into a vector. To this end, consider some cyclotomic ring \mathcal{R}^i such that $\mathcal{R}'' \subseteq \mathcal{R}^i \subseteq \mathcal{R}'$ and a modulus $t_j \in \mathcal{R}^i$ such that $t \mid t_j \mid p$. We will consider the quotient ring

$$\mathcal{R}^i_{t_j} = \mathbb{Z}[y]/(y^{n_i} + 1, t_j(y)),$$

where we substituted $y = x^{n/n_i}$ for ease of notation. Since the characteristic of this ring is a prime number p , we can work in the principal ideal domain $\mathbb{F}_p[y]$, and it follows that $(y^{n_i} + 1, t_j(y)) = (\tau(y), p)$, where $\tau(y) = \gcd(y^{n_i} + 1, t_j(y))$.

Roots of $\tau(y)$ over \mathbb{F}_p are also roots of $y^{n_i} + 1$, so they are primitive $2n_i$ -th roots of unity. Let $\zeta \in \mathbb{F}_p$ be such a root, then the other ones are of the form

$$y = \zeta^{(-1)^\alpha \cdot 5^\beta}, \quad (1)$$

though not necessarily each element of this shape is a root of $\tau(y)$. Our analysis will enforce an order to these roots and the corresponding plaintext slots. More precisely, let S be the set of roots of $\tau(y)$, with order relation \leq such that

$$\zeta^{(-1)^\alpha \cdot 5^\beta} \leq \zeta^{(-1)^\gamma \cdot 5^\delta}$$

if either $\beta < \delta$, or $\beta = \delta$ and $\alpha \leq \gamma$. This definition assumes that $0 \leq \alpha, \gamma < 2$ and $0 \leq \beta, \delta < n_i/2$. Finally, we use the isomorphism

$$\begin{aligned} \mathcal{R}_{t_j}^i &= \mathbb{Z}[y]/(\tau(y), p) \rightarrow \mathbb{F}_p^{|S|} \\ m(y) &\mapsto \{m(s)\}_{s \in S}. \end{aligned}$$

Addition and multiplication act component-wise on the vector of plaintext slots, and rotations can be implemented with automorphisms.

Our slot ordering is quite non-standard: the more common definition in the BFV literature [16] switches the role of α and β . However, it is more natural for our purpose due to the following reasons:

- Noisy expansion maps slots to coefficients in a permuted order. In the standard definition, this permutation is a bit reversal in both hypercolumns. Our ordering results in one full bit reversal over the entire slot-vector, similarly to the plaintext NTT algorithm.
- Given a cyclotomic ring \mathcal{R}^i and two “adjacent” moduli such that $t_j \mid t_{j+1}$, we can interpret $\mathcal{R}_{t_j}^i$ as a subring of $\mathcal{R}_{t_{j+1}}^i$. In our ordering, the slots from the first ring are simply the even-numbered slots from the second one.
- Similarly, given two “adjacent” cyclotomic rings such that $\mathcal{R}^i \supsetneq \mathcal{R}^{i+1}$ and a modulus t_j , we can interpret $\mathcal{R}_{t_j}^{i+1}$ as a subring of $\mathcal{R}_{t_j}^i$. In our ordering, the slot-vector of the second ring is obtained simply by duplicating the slot-vector of the first one.
- Our conversion algorithm from \mathcal{R}_t' to \mathcal{R}_p'' applies an additional permutation to the plaintext slots. In our ordering, this is a simple circular bit shift of ℓ positions to the left, which we denote by $\text{shift}_{n'', \ell}$.

Additional Restriction. We impose one more restriction on the input ciphertext. Let $\tau(y) = \gcd(y^{n'} + 1, t(y))$ where $y = x^{n/n'}$, then we assume that $t(y)$ lives in the cyclotomic subring of degree n'/n'' , and that $\tau(y)$ has n'' roots, obtained by setting $\alpha = 0$ in Equation (1). Note that we set the number of GBFV slots to n'' such that $\mathcal{R}_t' \cong \mathcal{R}_p''$. This restriction supports the setting of binomial plaintext moduli [17], which is also the standard setting of TopGear 2.0 [6]. Moreover,

it covers the parameter sets that are obtained by “descending” via the relative field norm, such as the Goldilocks prime in power-of-two cyclotomics [17].

Observe that this restriction only involves the GBFV ring and the BFV ring. The R-LWE ring can still be an arbitrary power-of-two ring of dimension $n \geq n'$. The fact that $\alpha = 0$ implies that \mathcal{R}'_t has a cyclic rotation group, but it does not necessarily hold for the other moduli: internally, bootstrapping uses a series of rings between \mathcal{R}'_t and \mathcal{R}''_p , where the latter has a rank-2 rotation group.

In the GBFV scheme, the noise growth upon multiplication is directly proportional to $||t||$. With the above restriction and under constant p , this norm is typically inversely exponential in n'/n'' . As a direct consequence, both the number of plaintext slots and the multiplication noise decrease when n'/n'' increases. We refer to [17] for more details.

Hensel Lifting. The GBFV and BFV plaintext rings can be lifted to prime powers. Following the procedure of [17], lifting to p^2 is achieved in the rings \mathcal{R}'_{t^2} and \mathcal{R}''_{p^2} . Both these quotient rings encode n'' slots from \mathbb{Z}_{p^2} . It is also possible to construct lifted rings between \mathcal{R}' and \mathcal{R}'' .

Trace Function. To take an encrypted plaintext from an extension ring to a subring, we can homomorphically apply the trace function [1]. For \mathcal{R}/\mathcal{R}' , this function is defined as

$$\text{Tr}_{\mathcal{R}/\mathcal{R}'} : \sum_{i=0}^{n-1} m_i \cdot x^i \mapsto (n/n') \cdot \left(\sum_{i=0}^{n'-1} m_{i \cdot n/n'} \cdot y^i \right). \quad (2)$$

It is well known that the trace can be computed iteratively, by passing through all intermediate cyclotomic subrings, and the cost is dominated by $\log_2(n/n')$ homomorphic automorphisms.

Linear Transformations. The core of our new bootstrapping is an algorithm that evaluates a specific \mathbb{F}_p -linear transformation on the vector of plaintext slots. This linear map converts between coefficient and slot representation. Below we discuss the state-of-the-art for implementing such linear transformations.

Baby-Step/Giant-Step Method. We implement “sparse” linear transformations with the baby-step/giant-step algorithm. In particular, we use the generalized variant of Cheon et al. [10]. This method computes linear combinations of rotated ciphertexts by splitting the relevant index set into two components. More specifically, let $I = J \cdot K \subseteq \mathbb{Z}_{2n}^\times$. Then we can rewrite a linear map L over index set I as

$$L(\mathbf{m}) = \sum_{i \in I} \kappa_i \cdot \sigma_i(\mathbf{m}) = \sum_{k \in K} \sigma_k \left[\sum_{j \in J} \kappa'_{jk} \cdot \sigma_j(\mathbf{m}) \right],$$

where $\kappa'_{jk} = \sigma_k^{-1}(\kappa_{jk})$. The minimum number of automorphisms is reached when J and K have (approximately) the same cardinality. Our implementation uses a heuristic algorithm to determine J and K from I .

Noisy Expansion. Bootstrapping requires linear transformations in the so-called noisy expansion step. Informally, noisy expansion takes a low-quality encryption of the SIMD vector $(m_0, \dots, m_{n'-1})$, and computes a high-quality encryption of $(p \cdot m_0 + e_0, \dots, p \cdot m_{n'-1} + e_{n'-1})$, where the entries of the input live in \mathbb{F}_p and the entries of the output in \mathbb{Z}_{p^2} . In other words, the plaintext space is expanded and contains additional “noise” terms e_i .

Noisy expansion consists of two major building blocks: the `SlotToCoeff` and the `CoeffToSlot` transformations. The first transformation converts an encryption of $(m_0, \dots, m_{n'-1})$ in BFV encoding to a ciphertext that encrypts the polynomial

$$m(x) = \sum_{i=0}^{n'-1} m_{\text{rev}_{n'}(i)} \cdot x^{(n/n') \cdot i} \in \mathcal{R}'_p,$$

where $\text{rev}_{n'}$ is the standard bit-reversal permutation of $\log_2(n')$ -bit elements. The `CoeffToSlot` transformation is simply the inverse map, but computed over \mathcal{R}'_{p^2} instead of \mathcal{R}'_p . Both transformations are \mathbb{F}_p -linear over the BFV plaintext space and can be further decomposed into a series of smaller stages, similarly to the number-theoretic transform (NTT).

The state-of-the-art `SlotToCoeff` transformation in power-of-two cyclotomics is the algorithm concurrently proposed by Geelen [16] and Ma et al. [23]. Both have identical complexity for the parameters in this paper. If $p = 1 \pmod{2n'}$, the algorithm boils down to homomorphically multiplying the vector of plaintext slots by an NTT-like matrix $U \in \mathbb{F}_p^{n' \times n'}$. Similarly to the NTT, this matrix can be decomposed as a product of up to $\log_2(n')$ sparse matrices, which we multiply to the vector of plaintext slots based on the baby-step/giant-step method.

3 Bootstrapping

We start this section with an informal description of our improved `SlotToCoeff` transformation and the resulting noisy expansion. Then we treat both building blocks formally, and we finally conclude by putting everything together in a new bootstrapping algorithm.

3.1 Informal Overview of the Proposed Method

A first idea to improve the `SlotToCoeff` transformation could be to evaluate its first couple of stages directly in the GBFV domain. More specifically, the first stage of the decomposition evaluates a “butterfly” operation on the two adjacent slots $m(\zeta^{5^i})$ and $m(\zeta^{-5^i})$ for all i in parallel. This is implemented as

$$\mathbf{m} \mapsto \kappa_0 \cdot \mathbf{m} + \kappa_1 \cdot \sigma_{-1}(\mathbf{m}),$$

for some constants κ_i . It is clear from the equation why this idea doesn't work: the automorphism σ_{-1} is invalid in GBFV, hence it must be evaluated in the BFV domain. More generally, for this strategy to work, we would need the stages of the NTT transformation to come in precisely the opposite order; that way, the automorphism σ_{-1} would come last in the decomposition, so we could postpone the GBFV-to-BFV conversion to that point.

Reverting the order of the NTT stages is not possible because they do not commute. Therefore, we need a different way to evaluate the transformation. Our idea is to rely on a subring encoding. First of all, note that we only consider the subset of GBFV slots, which is a fraction of the BFV slots. Therefore, we can map these GBFV slots to the coefficients of a BFV plaintext that lives in a cyclotomic subring of \mathcal{R}' . In summary, our method consists of two components, which are described in opposite order for didactic purposes:

- The second component is an algorithm that converts a GBFV ciphertext to a BFV ciphertext that lives in the smallest possible cyclotomic subring. This is basically a conversion from \mathcal{R}' to \mathcal{R}'' that also changes the slot-encoding. One side effect of the algorithm is that it applies an additional permutation on the plaintext slots, which can be understood as a circular bit shift.
- The first component is evaluating a negacyclic NTT, which is similar to the original transformation but defined in a smaller dimension (in dimension n'' rather than n'). To compensate for the circular bit shift, this transformation is applied to a slot-vector permuted with the inverse circular bit shift.

The computational cost of this algorithm is about the same as the state-of-the-art `SlotToCoeff` transformation (the only additional cost is a cheap trace operation). However, the new algorithm consumes substantially less computational budget, because the negacyclic NTT can be fully evaluated in the GBFV domain. Note that we do not achieve fully native GBFV bootstrapping (the second component converts to regular BFV), but in contrast to the previous work, only a very small fraction of the algorithm is evaluated in BFV.

We can exploit the reduced noise accumulation of our algorithm in two possible ways: either we use the same decomposition parameters for the `SlotToCoeff` transformation as the previous work, resulting in a ciphertext with more remaining computational budget. However, the previous work decomposed the transformation in only two stages - which is quite low in practice - making it the bottleneck in terms of execution time. Alternatively, since each stage consumes less computational budget, we can afford to decompose the transformation into more stages than the previous work. This second strategy is faster and uses fewer evaluation keys, but it slightly increases the accumulated noise.

3.2 GBFV to Subring BFV Conversion

This section describes our new homomorphic conversion routine from the GBFV plaintext ring \mathcal{R}'_t to the BFV plaintext ring \mathcal{R}''_p . Although both plaintext rings are isomorphic, they still have a different encoding, so that we cannot simply

reinterpret a plaintext from one ring as a plaintext of the other one. Instead, our algorithm evaluates a particular \mathbb{F}_p -linear transformation on the plaintext. This is done by iteratively passing through a series of intermediate rings, where each step “recodes” the underlying plaintext. More specifically, we use

$$\mathcal{R}' = \mathcal{R}^0 \supsetneq \mathcal{R}^1 \supsetneq \cdots \supsetneq \mathcal{R}^\ell \supseteq \mathcal{R}'',$$

where \mathcal{R}^i are consecutive power-of-two cyclotomic rings (i.e. the ratio of their dimensions is always 2). Since each step goes to a smaller cyclotomic ring, we need to compensate by increasing the underlying plaintext modulus. Therefore, we consider the chain of moduli

$$t = t_0 \mid t_1 \mid \cdots \mid t_\ell \mid p,$$

which are chosen in such a way that each $\mathcal{R}_{t_i}^i$ is isomorphic to $\mathbb{F}_p^{n''}$. In particular, our method constructs t_i by multiplying t_{i-1} with a shifted version of itself. More details follow later in this section.

Homomorphic Subring Conversion. We map a slot-permuted GBFV vector to the coefficients of a BFV ciphertext. This circular bit shift comes on top of the bit reversal permutation that was already present in the original `SlotToCoeff`. The circular bit shift permutation is a side effect of the subring conversion algorithm. If $n'' \geq n'/n''$, then subring conversion maps the slot with position i to the slot with position $\text{shift}_{n'',\ell}(i)$, where $\text{shift}_{n'',j}$ circularly shifts a $\log_2(n'')$ -bit number with j positions to the left. On the other hand, if $n'' \leq n'/n''$, no overall circular bit shift is applied. The next two paragraphs describe our algorithm in each case.

Many Plaintext Slots. If $n'' \geq n'/n''$, then we proceed as follows. Let there be $\ell = \log_2(n'/n'')$ iterations in our algorithm and consider

$$g_i = 5^{-2^{\ell-2}}, \dots, 5^{-2^1}, 5^{-2^0}, -1 \quad \text{for } i = 1, \dots, \ell. \quad (3)$$

Then we recursively define the next plaintext moduli as $t_i = t_{i-1} \cdot \sigma_{g_i}(t_{i-1})$ for each i .³ Iteration i of our subring conversion does the following:

- Compute `ModUp` to t_i , which gives an encrypted plaintext $\mu_i \in \mathcal{R}_{t_i}^{i-1}$. This plaintext has $2n''$ slots in total, the odd ones of which are identically zero.
- Next, we need to homomorphically map this plaintext to $\mathbf{m}_i \in \mathcal{R}_{t_i}^i$, which extracts the relevant n'' slots. To this end, we define the “mask” $\mathbf{a}_i \in \mathcal{R}_{t_i}^{i-1}$ that encodes ‘1’ in the first half and ‘0’ in the second half of the slot-vector. Then we homomorphically evaluate the function

$$\text{GBFV.SubPermute}_i: \mu_i \mapsto \mathbf{a}_i \cdot \mu_i + \sigma_{g_i}((1 - \mathbf{a}_i) \cdot \mu_i).$$

- Finally, we compute \mathbf{m}_i by taking the trace of $\mathcal{R}^{i-1}/\mathcal{R}^i$.

³ This operation corresponds to a field norm from the smallest cyclotomic ring \mathcal{T}^{i-1} that contains t_{i-1} , to its largest cyclotomic subring $\mathcal{T}^i \subsetneq \mathcal{T}^{i-1}$.

These steps are summarized in Algorithm 1.

Apart from mapping to a subring, each iteration also applies the permutation $\text{shift}_{n'',1}$ to the slot-vector, which accumulates to $\text{shift}_{n'',\ell}$ in total. This can be seen as follows:

- **ModUp** doubles the size of the slot-vector, and as a result, also doubles the index of each plaintext slot.
- **SubPermute** maps all slots in the second half from index j to index $j + 1$.
- The trace divides all slots in pairs and maps the slots in the second half from index j to $j - n''$.

These steps result in

$$\text{shift}_{n'',1}(j) = \begin{cases} 2j & \text{if } j < n''/2 \\ 2j - n'' + 1 & \text{if } j \geq n''/2. \end{cases}$$

This implements the aforementioned permutation, i.e. a circular bit shift to the left with one position.

We visualize the proposed transformation on the plaintext slots in Figure 1. The empty boxes are slots that are currently not in use. Each step permutes the order of the plaintext slots with a circular bit shift, and then maps to a smaller cyclotomic ring via the trace.

Few Plaintext Slots. If $n'' \leq n'/n''$, then the above method can be optimized by skipping some iterations. Intuitively, this works because the number of GBFV slots is relatively small, so we can map them to the BFV ring without passing through all intermediate cyclotomic rings. The only changes are the following:

- The number of iterations is $\ell = \log_2(n'')$. The other steps, including computation of the automorphism indices from Equation (3), are unchanged.
- Since $\mathcal{R}^\ell \neq \mathcal{R}''$ and $t_\ell \neq p$, we post-process by converting the modulus to p and taking the trace to \mathcal{R}'' .

These steps are again summarized in Algorithm 1, and we visualize the transformation on the plaintext slots in Figure 2. No overall circular bit shift is applied, because $\text{shift}_{n'',\ell}$ is the identity permutation. Notably, our conversion algorithm for few slots slightly violates the condition for an automorphism to be valid (i.e. we have that $\sigma_{g_i}(t_i) \neq t_i$). However, as shown in Figure 2, the result is still well-defined, because we ensure that the input plaintext is a multiple of $\sigma_{g_i}(t_{i-1})$. In the implementation, it suffices to compute the “invalid” automorphism σ_{g_i} by temporarily switching to the BFV domain.

Interestingly, the conversion algorithm is least efficient if $n'' = n'/n''$ (i.e. if the versions for many and few plaintext slots collide). In that case, the GBFV encoding and the BFV encoding are exactly incompatible, which results in a costly conversion algorithm.

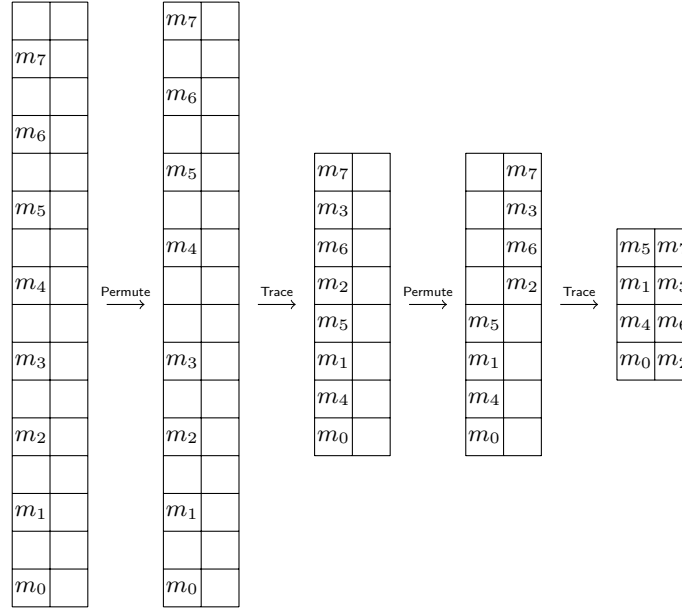


Fig. 1. Subring conversion for many plaintext slots (with $n' = 32$ and $n'' = 8$)

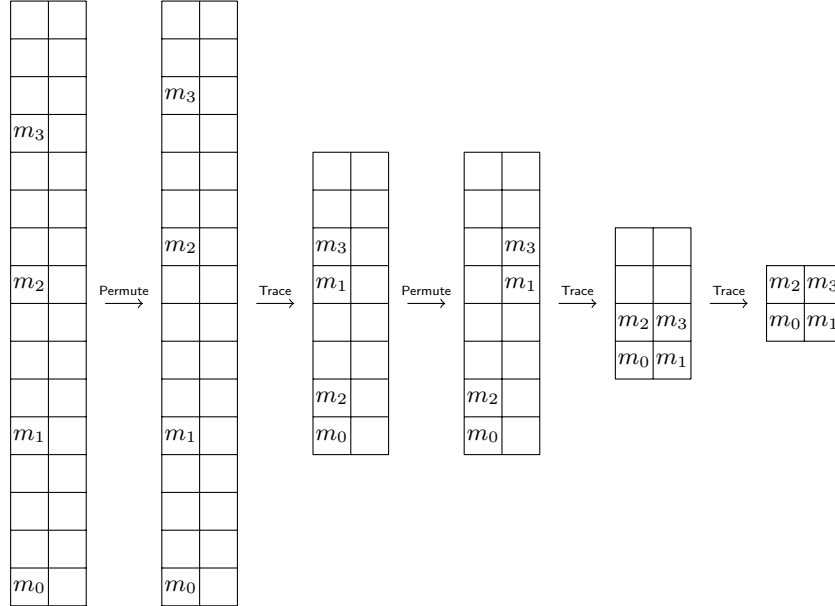


Fig. 2. Subring conversion for few plaintext slots (with $n' = 32$ and $n'' = 4$)

Optimizations. Algorithm 1 has multiplicative depth ℓ . One optimization that reduces the depth is merging multiple (or even all) iterations by skipping some intermediate rings of the form $\mathcal{R}_{t_i}^i$. To enable this optimization, the **SubPermute** operation must be implemented using a larger linear transformation, having 2^j terms when j levels are merged. This strategy is reminiscent of level-collapsing in homomorphic FFT evaluation [7].

Another improvement is merging the level of **ModUp** into **SubPermute**. More specifically, we can implement **ModUp** as a multiplication with a constant. This constant can be folded in the subsequent or previous linear transformation by adapting the definition of \mathbf{a}_i . This improvement is not possible if there is just a single slot (i.e. when we have $n'' = 1$), because there are zero iterations in that case. However, in the bootstrapping application, we can still fold our constant into the adapted digit removal polynomial, even if we have only a single slot.

Algorithm 1 Homomorphic subring conversion

Require: ct_0 that encrypts $\mathbf{m}_0 \in \mathcal{R}'_t$
Ensure: ct_ℓ that encrypts $\mathbf{m}_\ell \in \mathcal{R}''_p$, encoding the same slots as \mathbf{m}_0

- 1: **function** GBFV.SubConvert(ct_0)
- 2: $\ell \leftarrow \min(\log_2(n''), \log_2(n'/n''))$ ▷ Number of iterations
- 3: **for** $i \leftarrow 1$ to ℓ **do**
- 4: $\text{ct}_i \leftarrow \text{ModUp}(\text{ct}_{i-1}, t_i)$
- 5: $\text{ct}_i \leftarrow \text{SubPermute}_i(\text{ct}_i)$
- 6: $\text{ct}_i \leftarrow \text{Tr}_{\mathcal{R}^{i-1}/\mathcal{R}^i}(\text{ct}_i)$
- 7: **end for**
- 8: $\text{ct}_\ell \leftarrow \text{Tr}_{\mathcal{R}^\ell/\mathcal{R}''}(\text{ModUp}(\text{ct}_\ell, p))$ ▷ Void if $n'' \geq n'/n''$
- 9: **return** ct_ℓ
- 10: **end function**

Homomorphic Extension Ring Conversion. Our bootstrapping algorithm also requires the inverse operation of subring conversion: converting back to the original extension ring. Algorithm 2 specifies our extension ring conversion. A notable difference is that extension ring conversion is defined in characteristic p^2 . Specifically, it maps an element from \mathcal{R}''_{p^2} to the isomorphic ring \mathcal{R}'_{t^2} .

The basic idea of Algorithm 2 is to run all iterations of subring conversion in reverse order, using the inverse operations. The only exception is the trace: this function has no inverse, so instead we implicitly reinterpret an $\mathcal{R}^{\ell-i+1}$ -encryption as an $\mathcal{R}^{\ell-i}$ -encryption in iteration i . The inverse of **SubPermute_i** is

$$\text{GBFV.ExtPermute}_i : \mu_i \mapsto \mathbf{a}_i \cdot \mu_i + (1 - \mathbf{a}_i) \cdot \sigma_{g_i}^{-1}(\mu_i),$$

where we use the same \mathbf{a}_i as previously, but lifted to $\mathcal{R}_{t^2}^{i-1}$.

The previous optimizations also apply here. In particular, we can implement **ModDown** as a constant multiplication, similarly to **ModUp**. This multiplication can be merged with **ExtPermute**.

Algorithm 2 Homomorphic extension ring conversion

Require: ct_0 that encrypts $\mathbf{m}_0 \in \mathcal{R}_{p^2}''$
Ensure: ct_ℓ that encrypts $\mathbf{m}_\ell \in \mathcal{R}_{t^2}'$, encoding the same slots as \mathbf{m}_0

- 1: **function** GBFV.ExtConvert(ct_0)
- 2: $\ell \leftarrow \min(\log_2(n''), \log_2(n'/n''))$ ▷ Number of iterations
- 3: **for** $i \leftarrow 1$ to ℓ **do**
- 4: $\text{ct}_i \leftarrow \text{ExtPermute}_{\ell-i+1}(\text{ct}_{i-1})$
- 5: $\text{ct}_i \leftarrow \text{ModDown}(\text{ct}_i, t_{\ell-i}^2)$
- 6: **end for**
- 7: **return** ct_ℓ
- 8: **end function**

3.3 Improved Noisy Expansion

Algorithm 3 specifies our improved noisy expansion. In contrast to the original definition of noisy expansion [17], we explicitly incorporate the conversion from GBFV to BFV within the algorithm. It consists of the following steps:

- Multiply the slot-vector of the input ciphertext by an adapted NTT matrix. Let $U \in \mathbb{F}_p^{n'' \times n''}$ be the usual NTT matrix [16, 23], and furthermore consider the permutation matrix P that implements $\text{shift}_{n'', \ell}$. We define the adapted NTT matrix as $V = P^{-1}UP$ (i.e. a simple change of basis) and homomorphically multiply the slot-vector by V . We apply the same level-collapsing decomposition as the previous works [16, 23], but in the basis determined by P . Alternatively, this can be understood as evaluating the same transformation on a permuted slot-vector.
- Convert the resulting GBFV ciphertext to BFV with our new algorithm. A side effect of this step is multiplication by P , so the full transformation thus far is $PV = UP$.
- Refresh the ciphertext using **InProd** and take the trace to \mathcal{R}'' , which removes the redundant coefficients.
- Convert the resulting ciphertext back to the extension ring.
- Homomorphically multiply the slot-vector by $((n/n'') \cdot V)^{-1}$, which is defined over $\mathbb{Z}_{p^2}^{n'' \times n''}$. This step also compensates for the additional factor of (n/n'') in the trace (cf. Equation (2)).

Similarly to how we apply level-collapsing within multiplication by V , and within **SubConvert**, we can also apply level-collapsing by mixing these building blocks. Specifically, these building blocks boil down to linear transformations, and we can save multiplicative levels by composing multiple of them. For this optimization to work, we need to move the first occurrence of **ModUp** right before the composed linear transformation.

3.4 Our Bootstrapping Algorithm

We summarize our improved bootstrapping in Algorithm 4. First, it applies noisy expansion to the input ciphertext. Then it removes the additional noise terms

Algorithm 3 Noisy expansion

Require: ct that encrypts $\mathbf{m} \in \mathcal{R}'_t$
Ensure: ct_{exp} that encrypts $\mathbf{m}_{\text{exp}} \in \mathcal{R}'_{t^2}$, encoding a noisy version of the slots of \mathbf{m}

- 1: **function** GBFV.NoisyExpand(ct)
- 2: $\text{ct}_{\text{lt}} \leftarrow V \cdot \text{ct}$ ▷ Adapted SlotToCoeff
- 3: $\text{ct}_{\text{sub}} \leftarrow \text{SubConvert}(\text{ct}_{\text{lt}})$
- 4: $\text{ct}_{\text{fresh}} \leftarrow \text{Tr}_{\mathcal{R}/\mathcal{R}''}(\text{InProd}(\text{ct}_{\text{sub}}))$
- 5: $\text{ct}_{\text{ext}} \leftarrow \text{ExtConvert}(\text{ct}_{\text{fresh}})$
- 6: $\text{ct}_{\text{exp}} \leftarrow ((n/n'') \cdot V)^{-1} \cdot \text{ct}_{\text{ext}}$ ▷ Adapted CoeffToSlot
- 7: **return** ct_{exp}
- 8: **end function**

by applying a slot-wise digit removal polynomial. Finally, the result is converted back to modulus t with a homomorphic division.

Our bootstrapping uses the digit removal procedure from Ma et al. [22]. More specifically, we use their algorithm over \mathbb{Z}_{p^2} (not their alternative algorithm over the finite field \mathbb{F}_p because it is usually less efficient). We refer to the appendix of [16] for a simplified computation of the polynomial $H(x)$. Similarly to the previous work [17], we post-process by multiplying with $(p/t)^{-1} \pmod{t}$. In an implementation, this factor can be folded in the coefficients of $H(x)$.

Algorithm 4 Bootstrapping

Require: ct that encrypts $\mathbf{m} \in \mathcal{R}'_t$
Ensure: ct_{boot} that encrypts $\mathbf{m} \in \mathcal{R}'_t$ with less noise

- 1: **function** GBFV.Bootstrap(ct)
- 2: $\text{ct}_{\text{exp}} \leftarrow \text{NoisyExpand}(\text{ct})$
- 3: $\text{ct}_{\text{rem}} \leftarrow \text{Mul}(H(\text{ct}_{\text{exp}}), (p/t)^{-1})$ ▷ Adapted digit removal
- 4: $\text{ct}_{\text{boot}} \leftarrow \text{Divide}(\text{ct}_{\text{rem}}, t)$ ▷ Mere reinterpretation
- 5: **return** ct_{boot}
- 6: **end function**

3.5 Complexity Analysis

Neglecting the field trace (which can be evaluated in logarithmic time and without level consumption), noisy expansion consists of an equally expensive forward and inverse transformation. As such, it suffices to study the forward transformation (i.e. multiplication with the adapted SlotToCoeff matrix V in conjunction with SubConvert). In fully decomposed format, multiplication by V requires $\log_2(n'')$ iterations [16, 23]. Each iteration is a linear transformation that can be implemented using 2 (or sometimes 1) automorphisms and 3 (or sometimes 2) plaintext-ciphertext multiplications. Furthermore, our SubConvert algorithm has $\ell = \min(\log_2(n''), \log_2(n'/n''))$ iterations, each of which needs 1 automorphism and 2 plaintext-ciphertext multiplications.

As mentioned previously, we do not need to fully decompose the above transformation, and we can apply level-collapsing in each phase. In short, we choose decomposition parameters ℓ_1, \dots, ℓ_s such that $\ell_1 + \dots + \ell_s = \ell + \log_2(n'')$. Then we merge the first ℓ_1 iterations in a first stage, the next ℓ_2 iterations in a second stage, and so on. Each stage consumes one multiplicative level, and it requires $\mathcal{O}(2^{\ell_i/2})$ automorphisms and $\mathcal{O}(2^{\ell_i})$ plaintext-ciphertext multiplications using the baby-step/giant-step algorithm.

We believe that the most useful parameter sets in practice will be those with many slots (i.e. having $n'/n'' \geq n''$). In that case, the total number of iterations is $\log_2(n')$. This is the same as in the previous method [17], so our method does not improve the time complexity under the same decomposition parameters. Instead, the improvement is in terms of noise growth: the last stage typically covers the entire **SubConvert** algorithm, and **SlotToCoeff** is implemented in the other $s - 1$ stages. Therefore, the first $s - 1$ stages can be fully evaluated in GBFV, which incurs less noise growth than BFV. Because of the reduced noise growth, we can afford to decompose the linear transformation into more stages than the previous work. This does slightly increase the noise growth again, but it reduces the execution time. In particular, the previous work [17] decomposed in a very limited number of $s = 2$ stages for ring dimension $n = 2^{14}$, whereas we will also explore 3-stage and 4-stage decompositions.

In the end, the complexity of our algorithm (expressed as number of basic FHE operations), only depends on n' and n'' , but is independent of the R-LWE ring dimension n . In particular, for a constant number of slots n'' , the cheapest transformation is achieved when $n' = n''$. This special case coincides with the regular BFV scheme. When working in the GBFV scheme, the method needs more iterations because of **SubConvert**. It therefore becomes more expensive in execution time, but consumes less noise budget depending on the ratio n'/n'' .

4 Homomorphic Edit Distance

This section presents an algorithm to homomorphically compute the edit distance, more precisely the Levenshtein distance, leveraging our improved GBFV bootstrapping. Given two input strings \mathbf{a}, \mathbf{b} , the Levenshtein distance $\Delta(\mathbf{a}, \mathbf{b})$ is the minimal number of operations needed to transform the first input string into the second. The Levenshtein distance considers three operations: insertion, deletion and substitution. Each operation is considered to have unit cost.

4.1 Algorithmic Variants

Wagner-Fisher. As in previous works [13,21] on homomorphic edit distance, the starting point of our algorithm is the Wagner-Fisher [28] algorithm. Given two input strings

$$\mathbf{a} = a_1 \dots a_n \quad \text{and} \quad \mathbf{b} = b_1 \dots b_m,$$

this algorithm computes an $(n + 1) \times (m + 1)$ distance matrix D which equals $D[i, j] = \Delta(\mathbf{a}[1 : i], \mathbf{b}[1 : j])$ for $0 \leq i \leq n$ and $0 \leq j \leq m$, where $\mathbf{a}[1 : 0]$ and

$\mathbf{b}[1 : 0]$ are defined as the empty string. The first row thus satisfies $D[0, j] = j$ as well as $D[i, 0] = i$ for the first column. Since each operation has unit cost, we can compute $D[i, j]$ for $i, j > 0$ as

$$D[i, j] = \begin{cases} D[i-1, j-1] & \text{if } a_i = b_j \\ 1 + \min(D[i-1, j], D[i, j-1], D[i-1, j-1]) & \text{otherwise,} \end{cases} \quad (4)$$

where by definition $D[i, j] = 0$ for $i, j < 0$. The value $D[n, m]$ will then contain the actual Levenshtein distance. The definition of $D[i, j]$ already hints that an anti-diagonal approach is most natural, since the k -th anti-diagonal only depends on the $(k-1)$ and $(k-2)$ -th anti-diagonals.

To compute the above homomorphically, we thus need an efficient equality test and an efficient method to compute the minimum of three values. Note that the equality test is only computed on freshly encrypted ciphertexts, whereas the minimum function is not. Since the Levenshtein distance is limited to $n+m$, we know that the inputs to the minimum function are bounded by this value. However, computing such minimum is still very expensive homomorphically, which is why we switch to the Myers variant of the Wagner-Fischer algorithm.

Myers. The Myers [25] variant differs from Wagner-Fischer in that it computes an $(n+1) \times m$ matrix H and an $n \times (m+1)$ matrix V containing the differences between horizontal and vertical values in D , so in particular

$$H[i, j] = D[i, j] - D[i, j-1] \quad \text{and} \quad V[i, j] = D[i, j] - D[i-1, j].$$

The first row of H and the first column of V contain all 1's. By substituting how the values of $D[i, j]$ are defined in Equation (4), we can compute the difference matrices H and V for $0 < i \leq n$ and $0 < j \leq m$ as

$$\begin{aligned} H[i, j] &= 1 - V[i, j-1] + \min(-\delta_{i,j}, V[i, j-1], H[i-1, j]) \\ V[i, j] &= 1 - H[i-1, j] + \min(-\delta_{i,j}, V[i, j-1], H[i-1, j]), \end{aligned}$$

where $\delta_{i,j} = 1$ if $a_i = b_j$ and $\delta_{i,j} = 0$ otherwise. The Levenshtein distance can then be recovered by computing the sum of the entries in H and V that lie on any path from index $(0, 0)$ to index $(n-1, m-1)$, e.g. one could first compute the sum of the first row of H and then add the sum of the last column of V . Furthermore, since the first row consists of all 1's, the Levenshtein distance can be recovered as

$$\Delta(\mathbf{a}, \mathbf{b}) = m + \sum_{i=1}^n V[i, m].$$

Although the Myers algorithm computes two matrices instead of only one in Wagner-Fischer, the non-linear part, i.e. $\min(-\delta_{i,j}, V[i, j-1], H[i-1, j])$, is common for both approaches. As such, the number of non-linear functions to compute is identical to Wagner-Fischer. Furthermore, the main advantage of the above approach compared to Wagner-Fischer is that we only need to evaluate the minimum function on the restricted domain $S = \{-1, 0\} \times \{-1, 0, 1\}^2$.

Figure 3 shows an edit distance computation between the acronyms “FHE” and “HFE”, similar to the visualization of Legiest et al. [21]. Our figure shows both the absolute distance matrix D (used in the Wagner-Fischer algorithm), and the horizontal and vertical difference matrices H and V (used in the Myers variant). The edit distance $\Delta(\text{FHE}, \text{HFE}) = 2$ is obtained in the bottom right. That is, one word can be transformed into the other via two substitutions, or via one insertion and one deletion.

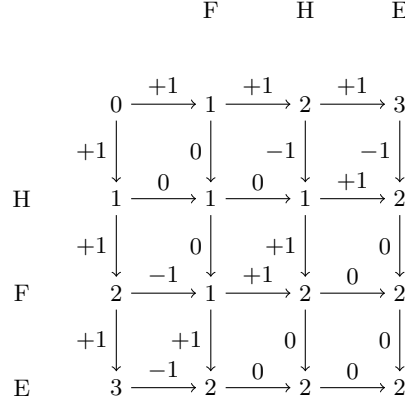


Fig. 3. Example edit distance computation between FHE and HFE

Homomorphic edit distance computation. Since we work with a finite field of odd characteristic, we can represent the minimum function $\min(x, y, z)$ on S by the following polynomial expression: set $h(t) = (1 - t) \cdot t$, then define

$$g(x, y, z) = x + ((1 + x)/4) \cdot (2h(y) + 2h(z) + h(y) \cdot h(z)) \quad \text{for } x, y, z \in S.$$

One can easily check that $g(x, y, z) = \min(x, y, z)$ on the domain S . Moreover, it is clear that g can be computed with a depth-3 circuit.

Assuming the characters belong to an alphabet \mathcal{A} , we will simply encode these as the set $\{0, \dots, |\mathcal{A}| - 1\}$. Of course this requires that the plaintext space can represent this set. In our application, we use $p = 2^{16} + 1$, which allows us to even represent 2-byte encodings, e.g. UTF-16. The complexity of the equality test directly depends on the size of \mathcal{A} . That is, using our encoding we can compute $\delta_{i,j}$ by first computing $z_{i,j} = (a_i - b_j)^2$, which can only take $|\mathcal{A}|$ values, namely $k^2 \bmod p$ for $k = 0, \dots, |\mathcal{A}| - 1$. Let $e(x)$ denote the polynomial that maps all these values to 0, except 0 which is mapped to 1, then it is clear that $\delta_{i,j} = e(z_{i,j})$. The total degree is $2(|\mathcal{A}| - 1)$, so the equality test has depth $1 + \lceil \log_2(|\mathcal{A}|) \rceil$. For the standard ASCII encoding which uses 7 bits, we obtain depth 8. If we use the full domain \mathbb{F}_p , the equality test simply becomes $1 - (a_i - b_j)^{p-1}$ because of Fermat’s little theorem.

Using the minimum and equality subroutines, we can formulate the homomorphic edit distance algorithm. For simplicity, we assume that $m = n$ and that our GBFV instance can encode up to d slots. In practice, we take $p = 2^{16} + 1$ and $d = 4096$ for a ring dimension of $n = 2^{14}$.

The main use case of edit distance computation is that one encrypted input string $\mathbf{b} = b_1 \dots b_m$ needs to be compared with a (typically large) number N of encrypted input strings $\mathbf{a}^{(i)}$. Since the GBFV instance can manipulate vectors of length d , and the intermediate values in the computation consist of m values per edit distance, we will be able to process d/m edit distance computations in parallel, where m is a power of two. By repeating this routine $\lceil Nm/d \rceil$ times, we can process N strings. To simplify notation, we will specify the algorithm for a single distance computation between \mathbf{a} and \mathbf{b} . The parallel version is similar, where the only difference is that d/m vectors of length m are packed in a single ciphertext. Specifically, our implementation uses an “interleaved” packing where multiple vectors are merged, so that their circular rotations do not interfere.

From the definition of the matrices H and V , we see that (like the Wagner-Fischer algorithm), it is possible to compute them in an anti-diagonal manner. In the k -th step we will therefore compute the k -th anti-diagonal of the matrices H and V . These anti-diagonals are packed in a single ciphertext ct_H (resp. ct_V) which in the k -th step (for $k \leq m$) of the algorithm contains the length- m vector

$$[H[0, k], \dots, H[k-1, 1], 0, \dots, 0],$$

and similarly for ct_V .

For $k > m$, the anti-diagonals shrink in size and the ciphertext ct_H contains the length- m vector

$$[H[k-m, m], \dots, H[m, k-m], 0, \dots, 0].$$

The actual algorithm does not explicitly put the last components of the above vector to zero (they may contain garbage values), but the first components will match. This saves unnecessary masking operations.

The only data that are still missing are the $\delta_{i,j}$. Since these are also needed in the same anti-diagonal order, and we want to compute them in the slots, we first duplicate the entries of \mathbf{a} to obtain $M = \lceil m^2/d \rceil$ ciphertexts (or up to $M = m$ ciphertexts in the parallel version) containing the m^2 values

$$a_1 \mid a_1, a_2 \mid a_1, a_2, a_3 \mid \dots \mid a_{m-1}, a_m \mid a_m,$$

where \mid separates subblocks (this is just for sake of clarity). We use a similar encoding for \mathbf{b} , but with subblocks in reverse order:

$$b_1 \mid b_2, b_1 \mid b_3, b_2, b_1 \mid \dots \mid b_m, b_{m-1} \mid b_m.$$

This can be derived using $(2m-1)$ maskings and $(2m-2)$ shifts for the vector \mathbf{a} if the result fits in a single ciphertext (and somewhat more masking operations otherwise). For \mathbf{b} , we first need to reverse the order of the characters, which can

be done using m maskings and m shifts, and then we proceed similar to **a**. Since the expansion operation for **b** is slightly more costly, it makes sense to use the fixed input ciphertext as the second argument in all edit distance computations, because we then only have to compute the above vector once.

By computing the difference of these M ciphertext pairs, squaring and evaluating the polynomial $e(x)$, we end up with M ciphertexts containing the $\delta_{i,j}$ in the order in which they are needed. This function is called `PrecompDeltas(cta, ctb)`. Finally, we can now formulate the whole homomorphic edit distance computation in Algorithm 5. This function uses the following GBFV subroutines:

- `Enc`: encryption.
- \oplus, \ominus and \otimes : shorthand for `GBFV.Add`, `GBFV.Sub` and `GBFV.Mul`.
- `ShiftRight` and `ShiftLeft`: circular shift to the right/left by 1 position, which requires a single automorphism.
- `Eval(g, [ct1, ..., ctk])`: evaluates multivariate polynomial g on $[ct_1, \dots, ct_k]$.
- `ExtractMinusDelta(C, k)`: extracts $-\delta_{i,j}$ with $i + j = k$ from the precomputed values in C . This requires one masking and one shift when these are contained in a single ciphertext, otherwise two maskings and two shifts.

The algorithm consists of two main loops: one for the top-left and one for the bottom-right differences. We use a temporary variable ct_T that holds the partial result for ct_H in the first loop and for ct_V in the second loop. This is in order not to overwrite the previous value of ct_H (resp. ct_V), which is still used on the subsequent line.

4.2 Complexity Analysis

The complexity and number of levels now follow from the description given in Algorithm 5. Precomputing and extracting $\delta_{i,j}$ requires depth $\lceil \log_2(|\mathcal{A}|) \rceil + 3$. Each evaluation of the polynomial g requires depth 3, so the total depth for all evaluations of g is $6m - 3$. The final masking has depth 1, but note that it can also be omitted (in that case, the other slots just contain garbage). Algorithm 5 on input strings of length m from an alphabet \mathcal{A} thus requires depth

$$\lceil \log_2(|\mathcal{A}|) \rceil + 6m + 1.$$

The evaluation of g is the only subroutine in our algorithm that accumulates multiplicative noise growth. As a consequence, it suffices to bootstrap ct_{\min} every iteration, or to bootstrap ct_H and ct_V once in L iterations, where L is defined as the ratio of the remaining noise budget after bootstrapping, to the consumed noise budget in one evaluation of g .

In the first and last quarter of the iterations, less than half of the plaintext vector contains useful data. Therefore, if we take the strategy of bootstrapping both ct_H and ct_V , we can combine them in a single ciphertext such that only one bootstrapping per iteration is required. This combination requires extra masking operations, but those can be folded for free in the computation of g . The time complexity for all operations is summarized in Table 1.

Algorithm 5 Homomorphic edit distance**Require:** ct_a, ct_b that encrypt input strings \mathbf{a} and \mathbf{b} of length m **Ensure:** ct_{acc} that encrypts Levenshtein edit distance in its first slot

```

1: function EditDistance( $\text{ct}_a, \text{ct}_b$ )
2:    $\text{ct}_H, \text{ct}_V \leftarrow \text{Enc}([1, 0, \dots, 0])$  ▷ Length- $m$  vectors
3:    $\text{ct}_{\text{acc}} \leftarrow \text{Enc}([m, 0, \dots, 0])$  ▷ Accumulator for result
4:    $C = (\text{ct}_1, \dots, \text{ct}_M) \leftarrow \text{PrecompDeltas}(\text{ct}_a, \text{ct}_b)$ 
5:   for  $k \leftarrow 2$  to  $m$  do
6:      $\text{ct}_{-\delta} \leftarrow \text{ExtractMinusDelta}(C, k)$ 
7:      $\text{ct}_{\min} \leftarrow \text{Eval}(g, [\text{ct}_{-\delta}, \text{ct}_H, \text{ct}_V])$  ▷ Minimum using polynomial  $g$ 
8:      $\text{ct}_T \leftarrow [1, 1, \dots, 0] \ominus \text{ct}_V \oplus \text{ct}_{\min}$  ▷ First  $k-1$  entries are 1's
9:      $\text{ct}_V \leftarrow [1, 1, \dots, 0] \ominus \text{ct}_H \oplus \text{ct}_{\min}$  ▷ First  $k$  entries are 1's
10:     $\text{ct}_H \leftarrow \text{ShiftRight}(\text{ct}_T) \oplus [1, 0, \dots, 0]$  ▷ Set  $H[1] = 1$ 
11:  end for
12:  for  $k \leftarrow m+1$  to  $2m$  do
13:     $\text{ct}_{-\delta} \leftarrow \text{ExtractMinusDelta}(C, k)$ 
14:     $\text{ct}_{\min} \leftarrow \text{Eval}(g, [\text{ct}_{-\delta}, \text{ct}_H, \text{ct}_V])$  ▷ Minimum using polynomial  $g$ 
15:     $\text{ct}_T \leftarrow [1, 1, \dots, 0] \ominus \text{ct}_H \oplus \text{ct}_{\min}$  ▷ First  $2m-k+1$  entries are 1's
16:     $\text{ct}_H \leftarrow [1, 1, \dots, 0] \ominus \text{ct}_V \oplus \text{ct}_{\min}$  ▷ First  $2m-k$  entries are 1's
17:     $\text{ct}_V \leftarrow \text{ShiftLeft}(\text{ct}_T)$ 
18:     $\text{ct}_{\text{acc}} \leftarrow \text{ct}_{\text{acc}} \oplus \text{ct}_T$ 
19:  end for
20:  return  $\text{ct}_{\text{acc}} \otimes [1, 0, \dots, 0]$  ▷ First entry contains actual distance
21: end function

```

5 Evaluation

5.1 Implementation

We implemented our new bootstrapping and the edit distance application in the SEAL FHE library [27]. All experiments below were performed on a 14-core M4 Pro machine with 64 GB RAM, and in a single thread. We take the same bootstrapping parameters as the previous work [17]: ring dimension $n = 2^{14}$, ciphertext modulus $q \approx 2^{420}$, noise cut-off parameter $B = 15$ for digit removal, a ternary secret key distribution with Hamming weight $h = 256$, and $\tilde{h} = 32$ for sparse secret encapsulation. This results in a security level of 128 bits. The baselines were re-evaluated because we use a different CPU than [17].

Table 1. Complexity of homomorphic edit distance computation for input length m , alphabet \mathcal{A} , duplication size M , and L evaluations of g between bootstrappings

PT \times CT	$9m + M \mathcal{A} $
CT \times CT	$8m + 2M\sqrt{ \mathcal{A} }$
Automorphisms	$9m$
Bootstrappings	$\min(3/L, 2) \cdot (m-1)$

Table 2. Comparison to the baseline [17] with $n = n' = 2^{14}$, $n'' = 2^{10}$ and $p = 2^{16} + 1$

Bootstrapping algorithm Number of stages s		Baseline 2	Ours 2	Ours 3	Ours 4
Noise (bits)	Initial	317	317	317	317
	Noisy expansion	111	81	96	110
	Digit removal	82	82	82	82
	Remaining	124	154	139	125
Execution time (sec)	Noisy expansion	0.95	1.16	0.61	0.47
	Digit removal	0.34	0.34	0.34	0.34
	Total	1.29	1.50	0.95	0.81

Table 3. Comparison to the baseline [17] with $n = n' = 2^{14}$, $n'' = 2^{12}$ and $p = 2^{16} + 1$

Bootstrapping algorithm Number of stages s		Baseline 2	Ours 2	Ours 3	Ours 4
Noise (bits)	Initial	317	317	317	317
	Noisy expansion	114	88	111	134
	Digit removal	113	113	113	113
	Remaining	90	116	93	70
Execution time (sec)	Noisy expansion	0.95	1.16	0.66	0.57
	Digit removal	0.35	0.35	0.35	0.35
	Total	1.30	1.51	1.01	0.92

Bootstrapping. Table 2 and Table 3 compare our new GBFV bootstrapping with the results from [17]. These GBFV parameter sets have 1024 and 4096 slots respectively, corresponding to 11 and 14 bits of multiplication noise. Due to the high noise growth of noisy expansion, the baseline algorithm decomposed the SlotToCoeff and CoeffToSlot transformations into a very limited number of only two stages. In contrast, we can decompose in more stages, while still having a comparable noise growth. For example, the four-stage decomposition of Table 2 has approximately the same remaining noise budget as the baseline, but the execution time is only 0.81 seconds instead of 1.29 seconds. On the other hand, if we decompose in two stages, our algorithm is 0.21 seconds slower than the baseline, but we have 30 bits extra remaining noise budget.

Table 4 shows experimental results for our new bootstrapping under a varying number of slots and stages. When there is only a single slot, we can bootstrap extremely fast and noisy expansion consumes almost no levels. In applications with a larger number of slots (such as encrypted edit distance), bootstrapping is slower and consumes more noise. However, amortizing the execution time and remaining noise capacity over all slots, the best performance is reached in the last column with 4096 slots.

Encrypted Edit Distance. Table 5 contains timings for the homomorphic edit distance application. We use the same parameter set as the third column of Table 3, which results in the best amortized performance. In particular, this

Table 4. Results for varying number of slots with $n = 2^{14}$, $n'/n'' = 2^2$ and $p = 2^{16} + 1$

Number of slots n''		1	16	256	4096
Number of stages s		0	1	2	3
Noise (bits)	Initial	317	317	317	317
	Noisy expansion	38	58	78	111
	Digit removal	109	109	112	113
	Remaining	170	150	127	93
Execution time (sec)	Noisy expansion	0.14	0.35	0.53	0.66
	Digit removal	0.35	0.35	0.35	0.35
	Total	0.49	0.70	0.88	1.01

parameter set has 93 remaining bits, which allows $L = 2$ evaluations of g between two bootstrappings. We use the variant of our algorithm that bootstraps both ct_H and ct_V every L iterations, which is the best strategy for $L \geq 2$. This table also compares to the TFHE-based Leuvenstein [21] method for homomorphic edit distance computation. The numbers of Leuvenstein are copied from their paper, where they used a dual AMD EPYC 9174F 16-core CPU. To allow a fair comparison, we use the same values of m , except for the second test since we are restricted to powers of two. We also use the same 7-bit ASCII alphabet.

The table indicates results for the parallel version of our algorithm. For all benchmarks, we compute the edit distance for a full batch of d/m strings. This setting utilizes the packing capability of GBFV to its maximum extent. Moreover, it results in $M = m$, meaning that the precomputation of $\delta_{i,j}$ can be stored in m intermediate ciphertexts.

Our amortized performance is up to $79\times$ better than Leuvenstein. This is a result of the parallel nature of GBFV, which allows us to process multiple edit distance computations in the same ciphertext. In terms of latency, Leuvenstein is faster than our method for a small number of $m = 8$ characters. However, we are still relatively close to their performance (the slowdown is a factor of 6) thanks to the very small bootstrapping latency of GBFV. For a larger number of $m = 128$ or even $m = 256$ characters, we do beat their latency because each ciphertext holds longer (but fewer) input strings. Most of the computation time is used by bootstrapping in our algorithm. The other operations only account for roughly 37% of the total computation time.

5.2 Comparison to Other Works

Bootstrapping GBFV with CKKS. A recent work from Kim [20] investigates alternative methods to bootstrap GBFV, but in a different setting. Specifically, this method can bootstrap a wider range of parameters (notably including the CLPX scheme), but this comes with two disadvantages: the ring dimension is 4 times higher, which leads to 20 times slower results, and the denoising factor is tightly coupled to the precision of CKKS bootstrapping. Therefore, Kim’s method is most useful in settings of extremely high-precision arithmetic such as the CLPX scheme. That scheme cannot be bootstrapped using our method,

Table 5. Timings (sec) for edit distance and comparison to Leuvenstein (LVS)

Algorithm	LVS	LVS	LVS	Ours	Ours	Ours
String size m	8	100	256	8	128	256
Batch size d/m	—	—	—	512	32	16
Equality tests	—	—	—	5.10	82	167
Bootstrappings	—	—	—	11.1	193	387
Other operations	—	—	—	1.50	26	57
Total latency	2.83	439	2903	17.7	301	611
Amortized time	2.83	439	2903	0.036	9.41	38.2
Speedup	—	—	—	$79\times$	—	$76\times$

because we convert to regular BFV. However, for moderately sized moduli such as the Fermat prime $p = 2^{16} + 1$, our method is preferred.

RMFE-Based BGV Bootstrapping. A recent work [2] improved BGV bootstrapping in small characteristic using a reverse multiplication-friendly embedding (RMFE). Compared to our bootstrapping, this is still an order or magnitude slower because of the larger ring dimension, and the number of slots in their method is quite restrictive (typically not more than a hundred). On the other hand, they do natively support small-characteristic fields. However, large-characteristic fields may suffice for many FHE applications (such as encrypted edit distance calculation) because any desired functionality can be interpolated over a given finite field.

Multiple Precision CKKS. Cheon et al. [9] proposed a Mult^2 algorithm that multiplies two CKKS ciphertexts with asymptotically twice the cost but half the modulus consumption of a normal multiplication. This is enabled via a new pair representation by breaking each ciphertext in two components. A quantitative comparison is difficult because of the approximate nature of CKKS, so instead we qualitatively compare GBFV bootstrapping to their Mult^2 algorithm.

Our improved GBFV bootstrapping is much closer to the Mult^2 method than the original GBFV bootstrapping. Specifically, all our bootstrapping components (except for InProd) can be evaluated in the GBFV scheme, similarly to how all CKKS bootstrapping components (except for ModRaise) can be evaluated in pair representation of ciphertexts. Similarly, both techniques reduce the modulus consumption by exploiting the underlying high precision, but they cannot reduce the contribution inherent to the ring dimension and secret key distribution.

We argue that we can reduce the asymptotic latency of GBFV bootstrapping even more than CKKS bootstrapping in pair representation. If $n' = 2n''$, then the modulus consumption of GBFV is roughly half that of regular BFV. In that case, we can scale down both the ring dimension and bit-width of the ciphertext modulus by a factor of 2, resulting in 4 times lower latency. The same trick is applicable to CKKS bootstrapping, but in the end its latency can only be halved,

because Mult^2 is roughly twice as expensive as a regular Mult . Neither work can asymptotically improve the throughput because of the reduced number of slots.

Finally, we note that CKKS tuple multiplication (i.e. Mult^t instead of Mult^2) involves a quadratic number of $\mathcal{O}(t^2)$ tensorings. For large t , we therefore expect tensoring to dominate in execution time over relinearization, which will saturate their latency reduction at a certain point. This is not the case in our algorithm.

Low-Latency CKKS Bootstrapping. Cheon et al. [11] recently proposed a low-latency CKKS bootstrapping algorithm called SHIP, which is designed for parallel computing environments. SHIP can bootstrap in ring dimension $n = 2^{13}$ (half that of our experiments), which results in a latency of around 0.2 seconds for 4096 slots. However, this parameter set has only one remaining multiplicative level, supports limited precision (1 to 5 bits) and uses more computing resources (a 32-core machine) compared to GBFV. Concurrently, Coron and Köstler [14] showed how to reduce the latency of CKKS bootstrapping by using complex roots of unity, but they focus on settings with a limited number of slots. A fully fair comparison to these works is not possible because they use CKKS encoding.

6 Conclusion

We improved the bootstrapping performance of GBFV to only a single second. We also showed its advantage over TFHE in homomorphic edit distance computation - an application that requires moderate packing density. Interesting future work is to integrate our bootstrapping in all state-of-the-art libraries [26,24,3,19] and to apply it in more FHE applications.

Acknowledgements. This work was supported in part by CyberSecurity Research Flanders with reference number VR20192203 and by the Research Council KU Leuven grant C14/24/099. In addition, this work is also supported in part by the European Commission through the Horizon 2020 research and innovation program Belfort ERC Advanced Grant 101020005 and through the Horizon 2020 research and innovation program under grant agreement ISOCRYPT ERC Advanced Grant 101020788. Robin Geelen is funded by Research Foundation – Flanders (FWO) under a PhD Fellowship fundamental research (project number 1162125N). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ERC, the European Union, CyberSecurity Research Flanders or the FWO. The authors thank Wouter Legiest for discussions about homomorphic edit distance computation.

References

1. Alperin-Sheriff, J., Peikert, C.: Practical bootstrapping in quasilinear time. In: CRYPTO (1). Lecture Notes in Computer Science, vol. 8042, pp. 1–20. Springer (2013)

2. Aung, K.M.M., Lim, E., Sim, J.J., Tan, B.H.M., Wang, H.: Bootstrapping with rmfe for fully homomorphic encryption. In: Jager, T., Pan, J. (eds.) *Public-Key Cryptography – PKC 2025*. pp. 71–101. Springer Nature Switzerland (2025)
3. Badawi, A.A., Bates, J., Bergamaschi, F., Cousins, D.B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., Kim, A., Lee, Y., Liu, Z., Micciancio, D., Quah, I., Polyakov, Y., Saraswathy, R.V., Rohloff, K., Saylor, J., Suponitsky, D., Triplett, M., Vaikuntanathan, V., Zucca, V.: Openfhe: Open-source fully homomorphic encryption library. In: *WAHC@CCS*. pp. 53–63. ACM (2022)
4. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical gapsvp. In: *CRYPTO*. *Lecture Notes in Computer Science*, vol. 7417, pp. 868–886. Springer (2012)
5. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory* **6**(3), 13:1–13:36 (2014)
6. Cha, H., Hwang, I., Min, S., Seo, J., Song, Y.: TopGear 2.0: Accelerated authenticated matrix triple generation with scalable prime fields via optimized HE packing. *Cryptology ePrint Archive*, Paper 2024/1502 (2024), <https://eprint.iacr.org/2024/1502>
7. Chen, H., Chillotti, I., Song, Y.: Improved bootstrapping for approximate homomorphic encryption. In: *EUROCRYPT (2)*. *Lecture Notes in Computer Science*, vol. 11477, pp. 34–54. Springer (2019)
8. Chen, H., Laine, K., Player, R., Xia, Y.: High-precision arithmetic in homomorphic encryption. In: *CT-RSA*. *Lecture Notes in Computer Science*, vol. 10808, pp. 116–136. Springer (2018)
9. Cheon, J.H., Cho, W., Kim, J., Stehlé, D.: Homomorphic multiple precision multiplication for CKKS and reduced modulus consumption. In: *CCS*. pp. 696–710. ACM (2023)
10. Cheon, J.H., Choe, H., Lee, D., Son, Y.: Faster linear transformations in HELib, revisited. *IEEE Access* **7**, 50595–50604 (2019)
11. Cheon, J.H., Hanrot, G., Kim, J., Stehlé, D.: Ship: A shallow and highly parallelizable ckks bootstrapping algorithm. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. pp. 398–428. Springer (2025)
12. Cheon, J.H., Kim, A., Kim, M., Song, Y.S.: Homomorphic encryption for arithmetic of approximate numbers. In: *ASIACRYPT (1)*. *Lecture Notes in Computer Science*, vol. 10624, pp. 409–437. Springer (2017)
13. Cheon, J.H., Kim, M., Lauter, K.E.: Homomorphic computation of edit distance. In: *Financial Cryptography Workshops*. *Lecture Notes in Computer Science*, vol. 8976, pp. 194–212. Springer (2015)
14. Coron, J.S., Köstler, R.: Low-latency bootstrapping for ckks using roots of unity. *Cryptology ePrint Archive* (2025)
15. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, Paper 2012/144 (2012), <https://eprint.iacr.org/2012/144>
16. Geelen, R.: Revisiting the slot-to-coefficient transformation for BGV and BFV. *IACR Commun. Cryptol.* **1**(3), 37 (2024)
17. Geelen, R., Vercauteren, F.: Fully homomorphic encryption for cyclotomic prime moduli. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. pp. 366–397. Springer (2025)
18. Gentry, C., Halevi, S., Peikert, C., Smart, N.P.: Ring switching in bgv-style homomorphic encryption. In: *SCN*. *Lecture Notes in Computer Science*, vol. 7485, pp. 19–37. Springer (2012)

19. Halevi, S., Shoup, V.: Design and implementation of HELib: a homomorphic encryption library. Cryptology ePrint Archive, Paper 2020/1481 (2020), <https://eprint.iacr.org/2020/1481>
20. Kim, J.: Bootstrapping GBFV with CKKS. Cryptology ePrint Archive, Paper 2025/888 (2025), <https://eprint.iacr.org/2025/888>
21. Legiest, W., D’Anvers, J.P., Spasic, B., Tran, N.L., Verbauwhede, I.: Leuven-shtein: Efficient FHE-based edit distance computation with single bootstrap per cell. Cryptology ePrint Archive, Paper 2025/012 (2025), <https://eprint.iacr.org/2025/012>
22. Ma, S., Huang, T., Wang, A., Wang, X.: Accelerating BGV bootstrapping for large p using null polynomials over \mathbb{Z}_{p^e} . In: EUROCRYPT (2). Lecture Notes in Computer Science, vol. 14652, pp. 403–432. Springer (2024)
23. Ma, S., Huang, T., Wang, A., Wang, X.: Faster BGV bootstrapping for power-of-two cyclotomics through homomorphic NTT. In: ASIACRYPT (1). Lecture Notes in Computer Science, vol. 15484, pp. 143–175. Springer (2024)
24. Mouchet, C.V., Bossuat, J.P., Troncoso-Pastoriza, J.R., Hubaux, J.P.: Lattigo: A multiparty homomorphic encryption library in go. In: Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography. pp. 64–70 (2020)
25. Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic programming. J. ACM **46**(3), 395–415 (1999)
26. Okada, H., Player, R., Pohmann, S.: Fheanor: a new, modular FHE library for designing and optimising schemes. Cryptology ePrint Archive, Paper 2025/864 (2025), <https://eprint.iacr.org/2025/864>
27. Microsoft SEAL (release 4.1). <https://github.com/Microsoft/SEAL> (Jan 2023), microsoft Research, Redmond, WA.
28. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM **21**(1), 168–173 (1974)