A New Perspective on Key Switching for BGV-like Schemes

Johannes Mono ●
johannes.mono@rub.de
0000-0002-0839-058X

Tim Güneysu ●●
tim.gueneysu@rub.de
0000-0002-3293-4989

● Ruhr University Bochum, Bochum, Germany
● DFKI GmbH, Bremen, Germany

Fully homomorphic encryption is a promising solution for privacy-preserving computation, especially involving sensitive data. For BFV, BGV, and CKKS, three state-of-the-art encryption schemes, the most costly homomorphic primitive is the so-called key switching. While a decent amount of research has been devoted to optimize other aspects of these schemes, key switching has gone largely untouched. One exception has been a recent work by Kim et al. at CRYPTO 2023 [26] introducing a new double-decomposition technique for state-of-the-art key switching. While their contributions are interesting, the authors have a skewed perspective on the complexity of key switching which results in a flawed parameter analysis and incorrect conclusions about the effectiveness of their approach. In this work, we correct their analysis with a new perspective on key switching and provide the new asymptotic bound $\mathcal{O}(\omega\ell)$. More generally, we take a holistic look at key switching and parameter selection. We revisit an idea by Gentry, Halevi, and Smart [19] improving key switching performance by up to 63 % and explore novel possibilities for parameter optimization. We also reduce the number of multiplications in key switching using new constant folding techinques, which speed up execution times by up to 11.6 %. Overall, we provide an in-depth analysis of key switching, guidelines for optimal parameter selection, and novel ideas which speed up execution times significantly.

Cryptography originally had one goal in mind: encrypting messages and ensuring the confidentiality of the encrypted data. Since then, it passed many generations and branched out to a wide variety of other applicable areas. One such application was first envisioned in the late 1970s under the name privacy homomorphism, a hopeful possibility of arbitrary computations on encrypted data [29]. It is a rather simple idea: A user encrypts (sensitive) data and sends it to a powerful server, the server manipulates the ciphertext to compute on the encrypted data, and then the user decrypts the ciphertext recovering the result. As simple as the idea sounds, realizing it proved to be a tough challenge for many years. Some constructions supported multiplications on encrypted numbers, but no additions. Others supported unlimited additions, but only one multiplication or many additions, but only a few multiplications. For arbitrary computations, however, any number of additions *and* multiplications was needed.

In 2009, Gentry introduced a novel idea, bootstrapping, and gave birth to the first ever encryption scheme for privacy homomorphism [17]. Over the years, many more and more efficient schemes have been conceived. Today, they are known as fully homomorphic encryption (FHE) schemes and, at their heart, are still rooted in Gentry's ingenious idea of bootstrapping. Conceptually, a ciphertext in any modern FHE scheme has an associated error which grows for each addition or multiplication. Once this error reaches a certain threshold, no further operations are possible without destroying the encrypted numbers. Gentry noticed that, given an encryption of the secret key, we can bootstrap the ciphertext and essentially refresh the associated error. By interleaving operations and bootstrappings appropriately, a server can perform any amount of additions and multiplications on the underlying numbers.

Today's state-of-the-art schemes fall into two groups: Boolean-based schemes encrypt single bits or small bit groups, and word-based schemes encrypt large vectors of numbers. While the former enjoys relatively fast bootstrapping and high computational flexibility, the latter suffers from much slower bootstrapping and less flexibility. For highly parallelizable arithmetic, however, word-based schemes outshine their Boolean-based companions. Consider vector arithmetic: In word-based schemes, homomorphic addition and multiplication map to the component-wise addition and multiplication of the encrypted vectors of numbers. In Boolean-based schemes, each bit of a vector element would require its own ciphertext and a vast number of homomorphic operations for a vector addition or multiplication. Word-based schemes also support a third prim-

itive, somewhat increasing their computational flexibility: rotations of the encrypted vector. Using rotations, we can map unencrypted algorithms to the homomorphic realm even if vector elements at different positions need to interact with each other. An example is homomorphic matrix multiplication which requires only two ciphertexts for word-based schemes, one for each matrix operand [23]. In Boolean-based schemes, we would need a seperate ciphertext for every single bit of every matrix element and many homomorphic operations.

Our work focuses on the word-based schemes BFV [6, 15], BGV [7], and CKKS [11]. They are also known as BGV-like due to their similar structure and base their security on the Learning with Errors over Rings (RLWE) assumption. For a ciphertext modulus q and a degree N, the ring $\mathcal{R}_q = \mathbb{Z}_q[X]/\left(X^N + 1\right)$ serves as the mathematical foundation of a ciphertext polynomial $c(s)$ with coefficients $c_i \in \mathcal{R}_q$. For decryption, we evaluate a ciphertext polynomial in the secret key s and recover the message m with an additional error te for a known constant t:

$$c(s) = c_0 + c_1 s = m + te.$$

Obviously, we only publish the coefficients of a ciphertext polynomial and keep the secret key our secret. Homomorphic addition and multiplication straightforwardly map to polynomial addition and multiplication of ciphertexts. Addition adds the messages as $c(s)+c'(s) = m+m'+te_{\text{add}}$ and multiplication multiplies them as $c(s) \cdot c'(s) = mm' + te_{\text{mul}}$. But, as straightforward as it may seem, two issues arise.

Especially for a multiplication, the error grows fast. To accomodate it, we require a large ciphertext modulus q sized several hundred bits and, as a consequence, require a large degree N to keep us secure in the RLWE setting. The large parameters result in expensive computations and subpar performance. To speed up computations, implementations employ two strategies: First, they decomposes the ciphertext modulus into ℓ coprime q_i with $q = \prod q_i$ enabling computations on the smaller moduli q_i; this is known as residue number system (RNS). Second, they use the forward and inverse number theoretic transform (NTT) for multiplication in $\mathcal{R}_q$. Although these two strategies ease the computational burden, they do not lift it and large parameters continue to be a major problem for performance.

The second issue is about the output ciphertext and how we decrypt it. The sum $(c+c')(s) = (c_0+c'_0)+(c_1+c'_1)s$ only requires s for decryption. The product

$$(c \cdot c')(s) = (c_0 c'_0) + (c_0 c'_1 + c_1 c'_0)s + (c_1 c'_1)s^2$$

on the other hand suddenly requires s^2 for decryption and further multiplications would worsen our problem exponentially. BGV-like schemes

avoid this ciphertext expansion with an internal housekeeping operation called key switching. Key switching transforms

$$(c_1 c_1')s^2 \quad \mapsto \quad \tilde{c}_0 + \tilde{c}_1 s + t\tilde{e},$$

holding the same information at the cost of an additional small error $\tilde{e}$. The modified homomorphic multiplication without ciphertext expansion outputs

$$(c \cdot c')(s) = (c_0 c_0' + \tilde{c}_0) + (c_0 c_1' + c_1 c_0' + \tilde{c}_1)s + t\tilde{e}$$

and only requires s for decryption. We also switch keys after our third primitive operation, namely rotations. To rotate an encrypted vector, we apply a permutation π on the ciphertext as $\pi(c(s)) = \pi(c_0) + \pi(c_1)\pi(s)$. As with a multiplication, we transform $\pi(c_1)\pi(s)$ to $\tilde{c}_0 + \tilde{c}_1 s$ at the cost of an added error. We control this error with two additional parameters: the key switching modulus P and the decomposition number ω.

1.1 Related Work

Although there exists a relatively large body of work exploring efficient parameter selection for the security level λ, polynomial degree N, and the ciphertext modulus q [2, 12, 1, 13, 28], the same cannot be said for the key switching parameters P and ω. This is even more surprising if we take into account that key switching is the most expensive primitive in BGV-like scheme. It is responsible for roughly 40 % of execution time during bootstrapping and $11\times$ slower compared to an expansionless ciphertext multiplication, the most expensive arithmetic operation[1]. Despite its significant impact on performance, related work on key switching is sparsely populated. In the extended version of their work, Kim, Polyakov, and Zucca [25] explore the current state-of-the-art on key switching. They describe two different methods, the BV method [8] and the GHS [18] method and their RNS variants. While they analyse computational and memory complexity, they only talk about correct parameter selection for key switching parameters and do not extend their analysis to optimal selection. In another work, Han and Ki [21] shortly discuss trade-offs for P on a high-level, but fail to provide a more detailed analysis such as asymptotic complexity or a method for selecting key switching parameters optimally.

At CRYPTO 2023, Kim et al. at [26] propose an extension to the current state-of-the-art with a double-decomposition technique. Although their contributions are interesting, the authors fail to properly analyse asymptotic complexity and optimal parameter selection for the current

1 We generated these numbers using our benchmarking setup using the open-source libraries OpenFHE and fhelib (see also Section 5).

state-of-the-art on key switching. Due to this insufficient analysis, they draw incorrect conclusions in the process and overestimate how significant their technique is for implementations.

1.2 Contributions

In this work, we take an in-depth look at key switching and make the following contributions:

- We analyze parameter selection for the key switching parameters P and ω and, taking a new perspective, show how to choose these parameters optimally for current state-of-the-art on key switching. Our new perspective introduces the new asymptotic complexity $\mathcal{O}(\omega\ell)$ on the number of NTT operations, correcting previous analysis by Kim et al. [26].
- We explore possibilities to reduce the parameters ω and ℓ. For the former, we dispel the myth that increasing the degree N always degrades performance and propose a new approach using two secret keys to improve performance. For the latter, we revisit an idea by Gentry, Halevi, and Smart [19] which we integrate into key switching to speed up key switching by up to 63 %.
- We correct the analysis by Kim et al. published at CRYPTO 2023 [26] and fully explore the parameter space for their double-decomposition technique. We implement their approach using state-of-the-art techniques and show its practical insignificance for BGV-like schemes.
- We highlight new opportunities for constant folding in key switching, reducing the total number of constant multiplications and speeding up key switching by up to 11.6 %.

We introduce the necessary background to our work in Section 2. In Section 3 and Section 4, we describe our theoretical contributions for single- and double-decomposition key switching. In Section 5, we evaluate our theoretical contributions with a state-of-the-art implementation. Finally, we discuss additional aspects of our work in Section 6 and conclude our work in Section 7.

Preliminaries
Section 2

The following provides the necessary background to our work. We start by defining notation used throughout the remainder of this work. Then, we introduce a shared context for the BGV-like schemes BFV, BGV, and CKKS and follow with short definitions for scheme-specific operations.

Table 1: Frequently used notation in the remainder of this work. Sometimes, additional variations are used to distinguish between different instances, for example q and q' for two distinct ciphertext moduli or s and s' for two distinct secret keys.

λ	security level
N	polynomial degree
p	plaintext modulus
t	error scaling factor
$q_{1,\ell}$	ciphertext modulus
ℓ	number of ciphertext primes
$P_{1,k}$	key switching modulus
k	number of key switching primes
ω	key switching decomposition number
b, β	bit size of q_i and P_j, respectively
B	upper bound on b and β
c_i	ciphertext polynomial
s	secret key polynomial
e	error polynomial

Afterward, we describe the Double CRT (DCRT) representation for efficient implementations and end with a unified view of modulus switching and key switching.

2.1 Notation

Let $q = \prod_{i=1}^{\ell} q_i$ be a product of primes with each q_i co-prime. For a subset of primes q_i, we write $q_{a,b} = \prod_{i=a}^{b} q_i$ for $a \leq b$ and $a, b \in \{1, \ldots, \ell\}$. For a power-of-two degree N, a ciphertext consists of polynomials c_i in the ring $\mathcal{R}_q = \mathbb{Z}_q[X]/\left(X^N + 1\right)$. We denote the centered modular reduction to the interval $[-q/2, q/2)$ as $[\cdot]_q$ and, for polynomials, apply modular reduction to each coefficient. We sometimes omit explicit modular reduction if no ambiguity exists. We denote a generalized rounding over the integers by $\lceil a \rfloor_t$, that is, we round to the closest integer such that $[a]_t = 0$. If not explicitly noted, we assume $t = 1$. We sample $s, u \leftarrow \chi_s$ and $e \leftarrow \chi_e$ from the secret and error distributions, respectively, and, with slight abuse of notation, write $a \leftarrow \mathcal{R}_q$ for a uniformly random sample from $\mathcal{R}_q$. Table 1 summarizes frequently used notation in the remainder of this work.

Polynomial norms.

The infinity norm $\|a\|_\infty$ of a polynomial $a \in \mathcal{R}$ is the maximum absolute value of all its coefficients. The canonical embedding norm is defined as $\|a\|_{\mathsf{can}} = \max_{j \in \mathbb{Z}_{2N}^*} \|a(\xi^j)\|_\infty$ for all primitive $2N$-th roots of unity ξ^j with $\|a \cdot b\|_{\mathsf{can}} = \|a\|_{\mathsf{can}} \cdot \|b\|_{\mathsf{can}}$ for any $a, b \in \mathcal{R}$. For a power-of-two degree,

$\|a\|_\infty \leq \|a\|_{\mathsf{can}}$ [14]. The canonical norm is bounded by $\|a\|_{\mathsf{can}} \leq D\sqrt{NV_a}$ for some D and the variance V_a. For a random polynomial $a \in \mathcal{R}_q$, we have $V_a \approx q^2/12$. Note that $V_{a+b} = V_a + V_b$, $V_{ab} = NV_aV_b$, and, for a constant γ, $V_{\gamma a} = \gamma^2 V_a$. For D, a common choice is $D = 6$ [12].

2.2 BGV-like schemes

We define a unified public key using the scheme-specific error scaling factor t as

$$\mathsf{pk} = ([as + te]_q, [-a]_q)$$

for $a \leftarrow \mathcal{R}_q$, $s \leftarrow \chi_s$, and $e \leftarrow \chi_e$. Using this public key, the individual encryption routines of the BGV-like schemes BFV, BGV, and CKKS output a ciphertext $(c_0, c_1) \in \mathcal{R}_q^2$. A useful perspective on a BGV-like ciphertext is as polynomial $c(s)$, and evaluating this polynomial in the secret key results in

$$c(s) = c_0 + c_1 s = m + te,$$

where m is a (possibly encoded) message and te is an (initially small) error that grows during homomorphic computation. The process of removing the error and decoding the message depends on the combination of encoding and error scaling, and we describe it in more detail in the respective scheme-specific sections.

Arithmetic operations nicely map to our polynomial representation: Addition on the underlying plaintext data m with ciphertext $c = (c_0, c_1)$ and m' with ciphertext $c' = (c'_0, c'_1)$ requires a polynomial addition

$$c(s) + c'(s) = c_0 + c'_0 + (c_1 + c'_1)s = m + m' + te_{\text{add}},$$

constant multiplication with a constant scalar or polynomial γ results in

$$\gamma c(s) = \gamma c_0 + \gamma c_1 s = \gamma m + te_{\text{const}},$$

and multiplication outputs the quadratic polynomial

$$c(s) \cdot c'(s) = c_0c'_0 + (c_0c'_1 + c_1c'_0)s + c_1c'_1s^2 = mm' + te_{\text{mul}}.$$

Usually, BGV-like schemes operate on multiple integers simultaneously; depending on the specific parameters, we encode up to N integers or approximate numbers in one message polynomial m. Polynomial addition or multiplication corresponds to element-wise operations on the encoded plaintext data. In Section 1, we state for simplicity that we encrypt a vector of numbers. In reality, the encoded data has a hypercube structure and applying a parameter- and dimension-specific permutation π on each ciphertext coefficient c_i rotates the hypercube along the chosen dimension.

Key switching.

After a rotation, the ciphertext is encrypted under the permutation of the secret key $\pi(s)$, and we apply key switching to transform $c(s) = c_0 + c_1\pi(s)$ to $\tilde{c}(s) = c_0 + \tilde{c}_0 + \tilde{c}_1 s$, encrypting the same message for a known permutation π under the original secret key. For a multiplication, key switching enables transforming the output $c(s) = c_0 + c_1 s + c_2 s^2$ to a new polynomial $\tilde{c}(s) = c_0 + \tilde{c}_0 + (c_1 + \tilde{c}_1)s$, encrypting the same message. Under certain circumstances, we can employ lazy key switching to delay this expensive process. For example, when accumulating several ciphertext products, it is more efficient to only apply key switching on the resulting sum instead of applying it to each ciphertext term individually. However, we often require a key switching, especially for rotations on the underlying hypercube structure, as these are usually used to enable operations between elements at different positions in the hypercube. This requires different permutations and, hence, key switching to get back to a shared secret key before further operations are possible. In Subsection 2.8, we define a unified approach to key switching.

Parameter selection.

Security for BGV-like schemes is based on the RLWE assumption. It depends on the distributions χ_s and χ_e, the polynomial degree N, and the product qP of the ciphertext modulus with the key switching modulus. A great tool to estimate security is the Lattice Estimator [2], which, given the parameters above, estimates the time and memory costs of the best-known lattice attacks. Common choices are a uniform ternary distribution χ_s and a centered Gaussian distribution χ_e with variance $\sigma = 3.19$ [1]. In this work, we sometimes refer to a simplified perspective on secure parameter selection, in which only the parameters N and qP determine security, implicitly assuming fixed distributions χ_s and χ_e.

Choosing the remaining parameters for a given use case is an extensive research area [2, 12, 1, 13, 28]. The following is a simplified (and slightly inaccurrate) description; however, it suffices for our purposes: An upper bound on the error is defined for each homomorphic operation for a given input bound. After multiplication, a scheme-specific error management technique slows down error growth to delay the necessity for bootstrapping. We retrieve an upper bound on q by composing the upper bounds of each operation according to a circuit model of a target use case. We then choose the smallest power-of-two degree N such that the security estimate for N and q is greater than λ. In most cases, the security estimate will exceed λ; we use this security margin for the key switching modulus P, choosing ω accordingly, such that a security estimate on N and qP is still larger than λ. The above essentially delivers a power-of-two degree N and ciphertext modulus q for a given use case and

security level, selecting key switching parameters afterward. Choosing optimal parameters can significantly reduce execution time and memory requirements.

2.3 The BFV scheme

The BFV scheme [6, 15, 25] is a state-of-the-art FHE scheme for integer arithmetic. Given a plaintext modulus p, the message polynomial $m \in \mathcal{R}_p$ is stored in the most significant bits of the ciphertext modulus encoded as $\lceil \frac{q}{p} m \rfloor$, while the error is stored in the least significant bits of the ciphertext modulus, that is, $t = 1$.

Encryption and decryption.

Using the unified public key pk, we encrypt a message as

$$(c_0, c_1) = \left(\left[\mathsf{pk}_0 u + e_0 + \left\lceil \frac{q}{p} m \right\rfloor \right]_q, [\mathsf{pk}_1 u + e_1]_q \right)$$

with $u \leftarrow \chi_s$, $e_0, e_1 \leftarrow \chi_e$, and decrypt with

$$m = \left[\left\lceil \frac{p}{q} [c_0 + c_1 s]_q \right\rfloor \right]_p$$

which is correct as long as the error is small enough. For a thorough analysis of error growth and security, we refer to the relevant works on the BFV scheme [6, 15, 25].

Error management.

In BFV, the error management technique requires a slight modification to the unified perspective on multiplication in Subsection 2.2. Instead of operating on c and c' modulo q, we switch the modulus of the latter to another co-prime modulus q' of approximately the same size. Then, after multiplication over the integers, a scaling and rounding $\left[\lceil \frac{p}{q'} cc' \rfloor \right]_q$ corrects the encoding factor [25]. In contrast to BGV and CKKS, we usually use the entire ciphertext modulus q throughout the complete circuit (temporarily operating modulo q' or qP); hence, BFV is also called scale-invariant, and the specific size of each q_i has negligible impact on error growth.

2.4 The BGV scheme

The BGV scheme [7, 25] is another state-of-the-art scheme for integer arithmetic. Contrary to BFV, the message polynomial $m \in \mathcal{R}_p$ is stored in the least significant bits of the ciphertext modulus, and the error sits above the message, that is, $t = p$. BFV and BGV are, in fact, so similar that we can convert between ciphertexts with a simple scalar multiplication [3].

Encryption and decryption.

We encrypt a message in BGV as

$$(c_0, c_1) = \left([\mathsf{pk}_0 u + pe_0 + m]_q , [\mathsf{pk}_1 u + pe_1]_q \right)$$

with random samples $u \leftarrow \chi_s$, $e_0, e_1 \leftarrow \chi_e$, and decryption computes as

$$m = \left[[c_0 + c_1 s]_q \right]_p .$$

As in BFV, the error needs to be small enough for correctness, and we refer to the relevant literature on the BGV scheme for details on error growth and security [7, 25, 28].

Error management.

In BGV, the error after multiplication is reduced by scaling the ciphertext with a part of the ciphertext modulus itself, consuming RNS primes q_i in the process. This process, modulus switching, scales the ciphertext, including the associated error, by q_i, while keeping the message the same. For example, consuming the ciphertext modulus prime q_ℓ scales the error by $1/q_\ell$, and afterward, ciphertexts live in the space $\mathcal{R}_{q_{1,\ell-1}}$. Therefore, the size of the individual q_i significantly impacts error growth during homomorphic evaluation and is highly relevant during parameter selection.

2.5 The CKKS scheme

Although a CKKS message $m \in \mathcal{R}$ consists of integers, we assume that an approximate result is good enough, such as with fixed-point arithmetic. Hence, we can consider the homomorphic error as part of the message itself, and both are stored in the least significant bits of the ciphertext modulus ($t = 1$). We refer to the relevant works on CKKS investigating security, error growth, and approximation error [11, 10, 24].

Encryption and decryption.

A CKKS encryption is the tuple

$$(c_0, c_1) = \left([\mathsf{pk}_0 u + e_0 + m]_q , [\mathsf{pk}_1 u + e_1]_q \right)$$

with $u \leftarrow \chi_s$, $e_0, e_1 \leftarrow \chi_e$, and, for decryption, we compute

$$m' = [c_0 + c_1 s]_q$$

recovering the approximate message m'.

Error management.

In CKKS, we use a conceptually similar approach as in BGV, serving a different purpose on the underlying plaintext data. As in BGV, we scale

by the individual q_i, operating on ciphertext polynomials in $\mathcal{R}_{q_{1,i-1}}$ afterward. There are two major differences: First, a small approximation error is acceptable as we are only interested in an approximate decryption. Second, the encrypted message moves to the more significant bits of the ciphertext modulus during multiplication, and instead of scaling the error, we move the message back to the least significant bits. As in BGV, choosing appropriate q_i is crucial to ensure correctness and a small approximation error.

2.6 The DCRT representation

The most common approach for implementing BGV-like schemes is using the Double CRT (DCRT) representation. We apply two Chinese Remainder Theorem (CRT) decompositions on a polynomial: the RNS for word-sized arithmetic modulo q and the NTT for fast polynomial multiplication.

Residue number system.

The RNS decomposes a number in $\mathbb{Z}_q$ towards each prime q_i. Instead of using one polynomial in $\mathcal{R}_q$, we use ℓ polynomials with $a_i = [a]_{q_i} \in \mathcal{R}_{q_i}$ for $a \in \mathcal{R}_q$. We also use the RNS for $P_{1,k}$. The CRT then reconstructs the original polynomial a based on the individual a_i. In the RNS, we can perform additions and (constant) multiplications as before. However, a limitation is that division and the modulo operator do not natively map to the RNS space, requiring other approaches. For division, there is a notable exception: If a constant $\gamma \in \mathbb{Z}$ divides each coefficient of $a \in \mathcal{R}_q$, division by γ over $\mathcal{R}$ corresponds to multiplication with the inverse $\gamma^{-1} \bmod q$.

We can convert a polynomial $a_i \in \mathcal{R}_{E_i}$ between two arbitrary RNS bases $E_{1,n}$ and $E'_{1,n'}$ with a fast base extension, only using word-sized arithmetic. The fast base extension is defined as

$$\mathsf{BaseExt}(a, E, E') = \left(\sum_{i=0}^{n} \left[a \frac{E_i}{E_{1,n}} \right]_{E_i} \frac{E_{1,n}}{E_i} \bmod E'_j \right)_{j=1}^{n'}$$

and outputs $a + \varepsilon E$ in the base $E'_{1,n'}$ for a small ε. Under certain circumstances, we can consider εE as part of the homomorphic error. Otherwise, we remove it using an error correction method such as BEHZ [5] or HPS [20].

Number theoretic transform.

The forward and inverse NTT are variants of the Fast Fourier Transform over a finite field evaluating a polynomial in the $2N$-th roots of unity ξ^j in time $\mathcal{O}(N \log N)$. For $\mathcal{R}_{q_i}$, we require $q_i = 1 \pmod{2N}$, and the NTT corresponds to the factorization $\prod(X - \xi^j) \bmod q_i$ into linear terms; another perspective on the factorization is as generalized CRT, hence the

name DCRT representation. For two polynomials in the NTT domain, we compute their product in time $\mathcal{O}(N)$ using coefficient-wise multiplication. Overall, NTT-based polynomial multiplication has an asymptotic complexity of $\mathcal{O}(N \log N)$. Due to the linearity of the NTT, polynomial addition and constant multiplication can be performed in either domain. However, when interacting with polynomials of other primes, such as during the fast base extension, we require a polynomial to be in the coefficient domain, requiring costly inverse and forward NTT operations.

2.7 Modulus switching

We define a unified version of modulus switching for $a \in \mathcal{R}_q$ to $a' \in \mathcal{R}_{q'}$ for all BGV-like schemes based on the generalized rounding as

$$a' = \left[\left\lfloor \frac{q'}{q} a \right\rceil_t\right]_{q'} = \left[\frac{q'a + \delta}{q}\right]_{q'} = \left[\frac{q'a - t\left[t^{-1}q'a\right]_q}{q}\right]_{q'}.$$

The above is RNS-friendly since, by definition, $q \mid (q'a+\delta)$, and multiplying with the multiplicative inverse corresponds to the required division. For $t = 1$, this scales the encrypted data in the least significant bits by roughly q'/q (the error for BFV and the approximate message for CKKS), and, with $t = p$ as in BGV, we have

$$-t\left[t^{-1}q'a\right]_q = 0 \pmod p,$$

only scaling the error by q'/q and keeping the encrypted message intact.

2.8 Key switching

Key switching transforms a BGV-like ciphertext monomial $c_i s'$ to a polynomial

$$c_i s' = \tilde{c}_0 + \tilde{c}_1 s + te$$

for a small error e. The following summarizes the current state-of-the-art on key switching [25, 26]. We need to generate key switching keys, and each distinct s' requires a distinct key switching key. For example, key switching after a multiplications needs one key switching key with $s' = s^2$, while a fixed permutation π requires another key switching key with $s' = \pi(s)$. A BGV-like key switching key is the tuple

$$\mathsf{ksw} = \left(\left[as + te + s'\right]_q, \left[-a\right]_q\right)$$

for $a \leftarrow \mathcal{R}_q$, $e \leftarrow \chi_e$. For naïve key switching, we compute

$$\begin{aligned} c_i \mathsf{ksw}(s) &= c_i \mathsf{ksw}_0 + c_i \mathsf{ksw}_1 s \pmod q \\ &= c_i s' + t c_i e \pmod q \end{aligned}$$

Note, however, that the error tc_ie is, in fact, not small as initially claimed. Hence, we need to modify key switching to reduce this error term.

Single-decomposition technique.

Current state-of-the-art, which we also refer to as the single-decomposition technique, employs two different approaches to control the error: First, we decompose the ciphertext into ω groups, reducing the bound to the decomposition (also known as BV technique) [8], and second, we temporarily operate on an extended modulus qP, scaling the error by P afterward (also known as GHS technique) [19]. Kim, Polyakov, and Zucca provide an excellent summary of each approach individually as well as the generalization to their combination, the hybrid approach, in the extended version of their work, which we recommend for an in-depth description [25]. In the following, we summarize the generalization using our notation.

For a decomposition $\mathcal{D}(\cdot)$, its inverse $\mathcal{D}^{-1}(\cdot)$, and $a, b \in \mathcal{R}_{qP}$, we require $ab = \langle \mathcal{D}(a), \mathcal{D}^{-1}(b) \rangle$. In DCRT-based implementations, we decompose towards the RNS basis, dividing the primes into ω groups $q_{\mathcal{D}_j}$ with up to $\lceil \ell/\omega \rceil$ primes in each group and use the CRT as $\mathcal{D}^{-1}(\cdot)$. Computing $\mathcal{D}(a)$ corresponds to $[a]_{q_i}$ for $q_i \in q_{\mathcal{D}_j}$ and is free [20]. We define a BGV-like key switching key as

$$\mathsf{ksw} = \left(\left[a_j s + t e_j + P \mathcal{D}_j^{-1}(s') \right]_{qP}, [-a]_{qP} \right)_{j=1}^{\omega}$$

where $a \leftarrow \mathcal{R}_{qP}^{\omega}$, $e \leftarrow \chi_e^{\omega}$, and $\mathcal{D}_j^{-1}(\cdot)$ is the jth entry of $\mathcal{D}^{-1}(\cdot)$. Slightly abusing notation, we switch keys with

$$\left\lceil \frac{[\langle \mathcal{D}(c_i), \mathsf{ksw} \rangle (s)]_{qP}}{P} \right\rfloor_t = c_i s' + \left\lceil \frac{[t \langle \mathcal{D}(c_i), e \rangle]_{qP}}{P} \right\rfloor_t .$$

In practice, the resulting error is negligible for $k = \lceil \ell/\omega \rceil$ (and hence $P = P_{1,k}$ with each $P_j \approx \max_i q_i$) [25]. Thus, when selecting key switching parameters, the key switching error is always negligible if we either choose ω and compute k accordingly, or choose k and compute ω accordingly.

Choosing $\omega = 1$ corresponds to only using the GHS techinque, and choosing $\omega = \ell$ with $P = 1$ would correspond to the BV technique. However, the latter would require a second decomposition to a small digit base as only decomposing toward the RNS base does not sufficiently reduce the error [25]. As this is rather inefficient, we reasonably assume $P \geq \max_i q_i$ (with $k = 1$ and $P \approx \max_i q_i$ for $\omega = \ell$); this corresponds to current state-of-the-art. Kim, Polyakov, and Zucca [25] analyze the complexity of key switching with respect to the number of NTT operations (the most costly building block of key switching), the number of multiplications, and the memory requirements for a key switching key in bits. However, they do not further explore the parameter space or discuss further implications for parameter selection. We summarize their results in Table 2.

Table 2: Single-decomposition key switching complexity according to the analysis of Kim, Polyakov, and Zucca [25] in terms of forward and inverse NTT (`ntt`) as well as modular (constant) multiplications (`mul`) with input and output in the same domain, coefficient for BFV and NTT for BGV. Complexity $\mathcal{O}(N \log N)$ for `ntt` and $\mathcal{O}(N)$ for `mul` are implicit.

Metric	Scheme	Cost
`ntt`	BFV/BGV	$(\omega+2)(\ell+k)$
`mul`	BFV	$\ell(\ell+2\omega+2k+5)+2k$
`mul`	BGV	$\ell(\ell+2\omega+2k+7)+4k$
bit size	BFV/BGV	$2\omega N \log qP$

For a correct implementation, we also have to consider the fast base extension as well as forward and inverse NTT operations. For completeness, we also take into account that we add the output of key switching to (parts of) an existing ciphertext. For example, considering BFV (input and output in the coefficient domain), key switching is as follows:

1 Ciphertext extension: Perform a fast base extension on the ω ciphertext decompositions modulo $q_{\mathcal{D}_j}$ as

$$c_{\text{ext}} = \mathsf{BaseExt}(\mathcal{D}_j(c_i), q_{\mathcal{D}_j}, qP).$$

2 Dot product: With a key switching key in the NTT domain, perform the dot product as

$$c'_i = \langle \mathsf{NTT}_{\mathsf{fwd}}(c_{\text{ext}}), \mathsf{ksw}_i \rangle \pmod{qP}.$$

3 Delta computation: Compute δ for scaling by P as

$$\delta_i = -t\, \mathsf{BaseExt}\left(\mathsf{NTT}_{\mathsf{inv}}\left(\left[t^{-1}c'_i\right]_P\right), P, q\right) \pmod{q}.$$

4 Modulus switching: Scale down $c'_i + \delta_i$ and add the result to the input as

$$\left(c_0 + \frac{\mathsf{NTT}_{\mathsf{inv}}(c'_0)+\delta_0}{P}, c_1 + \frac{\mathsf{NTT}_{\mathsf{inv}}(c'_1)+\delta_1}{P}\right).$$

When we switch for c_1, for example, after a rotation, we set $c_1 = 0$ in this final step.

Double-decomposition technique.

Kim et al. recently proposed a double-decomposition technique building upon single-decomposition key switching [26]. The technique only changes the algorithmic approach to key switching and does not influence the error. Their idea is as follows: In the second step, instead of

computing the dot product $\langle \mathcal{D}(c_i), \mathsf{ksw} \rangle$ in $\mathcal{R}_{qP}$, we add a second decomposition over qP into $\tilde{\omega}$ groups such that each group has up to $\lceil (\ell + k)/\tilde{\omega} \rceil$ primes. Then, switching to a shared RNS basis $E = \{E_1, \ldots, E_r\}$ and computing the dot product in $\mathcal{R}_E$ can improve execution time. For more details, we refer to the original publication [26].

In their evaluation, the authors assume CKKS with input and output in the coefficient domain and without the modulus switching costs for scaling down by P. We list their results, including modulus switching costs, in Table 3 to keep consistency with Kim, Polyakov, and Zucca [25]. For an implementation, again considering input and output in the coefficient domain, we compute

1 Ciphertext extension:

$$c_{\mathsf{ext}} = \mathsf{BaseExt}(\mathcal{D}_j(c_i), q_{\mathcal{D}_j}, E).$$

2 Dot product:

$$c_i'' = [\langle \mathsf{NTT}_{\mathsf{fwd}}(c_{\mathsf{ext}}), \mathsf{ksw}_i \rangle]_E$$
$$c_i' = \mathsf{BaseExt}(\mathsf{NTT}_{\mathsf{inv}}(c_i''), E, qP).$$

3 Delta computation:

$$\delta_i = -t\, \mathsf{BaseExt}\left(\left[t^{-1} c_i'\right]_P, P, q\right) \pmod{q}.$$

4 Modulus switching:

$$\left(c_0 + \frac{c_0' + \delta_0}{P}, c_1 + \frac{c_1' + \delta_1}{P}\right).$$

As for the single-decomposition technique, when switching c_1, we set $c_1 = 0$ in this final step.

In contrast to the single-decomposition technique, where we can always consider the error as part of the ciphertext error [28], the fast base extension in step (2) of the double-decomposition technique requires error correction.

Single-decomposition key switching
Section 3

In this and the following section, we introduce our theoretical contributions for the single- and double-decomposition key switching techniques, respectively. Both sections follow the same blueprint: We start by generalizing key switching to arbitrary input and output domains. We then analyze computational and memory complexity introducing our optimizations for constant folding. Afterward, we discuss strategies for optimizing parameters.

Table 3: Double-decomposition key switching complexity, including modulus switching costs, in terms of forward and inverse NTT (`ntt`) as well as modular (constant) multiplications (`mul`) for input and output in the coefficient domain.

Metric	Cost
`ntt`	$(\omega + 2\tilde{\omega})r$
`mul`	$(3\ell + 2\omega\tilde{\omega} + 2k + 2)r + \ell(2k + 7) + 6k$
bit size	$2\omega\tilde{\omega}N \log E$

3.1 Generalizing NTT complexity to arbitrary domains

An annoying limitation of current analyses is the coupling with a specific scheme such as BFV or BGV. Instead, we choose to analyze key switching for a given input and a target output domain, generalizing analysis to all BGV-like schemes. For example, in BFV, the modulus switching before a multiplication requires input in the coefficient domain, and the scaling and rounding afterward results in output in the coefficient domain. Therefore, the most common input and output domain in BFV is the coefficient domain; however, depending on use-case-specific circumstances, one might also want to work with input or produce output in the NTT domain. In contrast, in BGV and CKKS, multiplication requires input in the NTT domain and produces output in the NTT domain, and thus, the most common input and output domain for key switching is the NTT domain. For rotations, efficient variants exist for permuting the ciphertext polynomials in either domain, and homomorphic rotations do not influence the common domain for a specific scheme.

Kim, Polyakov, and Zucca [25] show that ciphertext extension, dot product, and delta computation require $\omega(\ell + k) + 2k$ NTT operations independent of the input domain. Then, for modulus switching, we compute

$$\left(c_0 + \frac{c_0' + \delta_0}{P},\ c_1 + \frac{c_1' + \delta_1}{P}\right),$$

where c_i is the ciphertext input, c_i' the output of the dot product in the NTT domain, and δ_i the output of the delta computation in the coefficient domain. For rotations, we set $c_1 = 0$.

For a matching input and output domain, it is relatively straightforward to get the desired result with 2ℓ additional NTT operations and an overall complexity of $(\omega + 2)(\ell + k)$: For c_i in the coefficient domain, we apply ℓ inverse NTTs on each c_i', one for each ciphertext prime, while for c_i in the NTT domain, we apply ℓ forward NTTs on each δ_i. For a non-matching input and output domain, we can achieve the same complexity

in terms of NTT operations with

$$\left(\frac{Pc_0 + c_0' + \delta_0}{P}, \frac{Pc_1 + c_1' + \delta_1}{P}\right),$$

scaling c_i by P. Depending on the input domain, we add Pc_i to c_i' or δ_i, afterward applying 2ℓ forward or inverse NTT operations as required for the desired output domain. While this increases the number of multiplications for now, we show in Subsection 3.3 how to avoid these additional costs.

3.2 A new perspective on NTT complexity

In their work, Kim et al. [26] claim that the asymptotic key switching complexity for the single-decomposition technique is $\mathcal{O}(\ell^2)$. The authors reason as follows: By choosing $k \in \mathcal{O}(1)$, we must choose $\omega \in \mathcal{O}(\ell)$ and the bound follows accordingly. However, this implicitly limits the parameter space for k which results in a skewed perspective on key switching. In this work, we choose a different perspective on key switching. We consider $\omega \leq \ell$ as a parameter in the security level, which we can choose as we desire. Then, assuming $b \approx \beta \approx B$ and, for simplicity, $\omega \mid \ell$, the number of primes in the key switching modulus follows as $k = \ell/\omega$ to keep the key switching error small (see also Subsection 2.8).

Thus, the number of NTT operations is

$$(\omega + 2)(\ell + k) = \omega\ell + 3\ell + \frac{2\ell}{\omega}.$$

For $\omega_2 = 2, \omega_1 = 1$,

$$\omega_2\ell + 3\ell + \frac{2\ell}{\omega_2} = \omega_1\ell + 3\ell + \frac{2\ell}{\omega_1},$$

and for $\omega_2 > \omega_1 > 1$,

$$\omega_2\ell + 3\ell + \frac{2\ell}{\omega_2} > \omega_1\ell + 3\ell + \frac{2\ell}{\omega_1}.$$

A simple fact follows: Increasing ω increases the computational complexity of key switching, and for better performance, we want to choose $k \in \mathcal{O}(\ell/\omega)$. This results in an asymptotic key switching complexity of

$$\omega\ell + 3\ell + \frac{2\ell}{\omega} = \mathcal{O}(\omega\ell).$$

3.3 Improving multiplication complexity via constant folding

We split multiplication types in key switching into four context-based groups:

1 the coefficient-wise multiplication with the key switching key during the dot product step;

2 multiplications with a constant scalar such as P^{-1} during modulus switching;

3 during base extension, multiplication with $E_i/E_{1,n} \bmod E_i$, for example, for $E = P$; and

4 during base extension, multiplication with $E_{1,n}/E_i \bmod E'_j$.

Note that, for group (1), we can only fold fixed values such as t or P since the ciphertext modulus can change as in BGV or CKKS. For brevity, the following only outlines our optimizations for input and output in the coefficient domain. However, the ideas also apply to other combinations of input and output domain as well as optimizations introduced later in this work; we refer the interested reader to our implementation in which all folded variants are implemented and tested.

Our crucial observation is the following: After the dot product, we use $c' = (c'_0, c'_1) \bmod P$ only to compute $\delta = (\delta_0, \delta_1)$ and use $c' \bmod q$ only for switching the modulus. Thus, we can merge known multiplications (group 2–4) for $c' \bmod q$ with the key switching key $\mathsf{ksw} \bmod q$ and with $\mathsf{ksw} \bmod P$ separately (group 1). Additionally, for the delta computation, we can merge the multiplication by $-t$ over q_i (group 2) with the second part of the base extension from P to q (group 4). Below, we list our algorithmic modifications to the key switching algorithm starting at the dot product. For readability, we exclude the required NTT operations, as constant multiplications can be performed in either domain:

$$\begin{array}{ll} [\cdot]_q & [\cdot]_P \\ c'_{\text{fold}} = \langle c_{\text{ext}}, P^{-1}\mathsf{ksw} \rangle & \delta'_{\text{fold}} = \langle c_{\text{ext}}, (tP/P_j)^{-1}\mathsf{ksw} \rangle \\ \delta_{\text{fold}} = P^{-1}tP/P_j \, \delta'_{\text{fold}} & \\ c + c'_{\text{fold}} + \delta_{\text{fold}} & \end{array}$$

The above reduces the number of constant multiplications down to a unified bound of

$$\ell \left(\ell + 2\omega + 2\frac{\ell}{\omega} + 3 \right)$$

for any t and any combination of input and output domain, each multiplication with complexity $\mathcal{O}(N)$.

Given a degree N and a number of primes ℓ, finding the minimal number of multiplications depends only on $2\omega + 2\ell/\omega$, which is minimal for $\omega = \sqrt{\ell}$. Due to the implicit scaling by P^{-1}, we can add c without additional multiplications to either c'_{fold} in the NTT domain or to δ_{fold} in the coefficient domain. During precomputation, we simplify $P^{-1}tP/P_j = t/P_j$. Our folding ideas also apply to the fast base extension when correcting the error with the HPS method, which we use for implementation.

3.4 Analyzing memory costs

In the single-decomposition technique, each key switching key requires ω polynomial pairs for the modulus qP. Thus, assuming B bits of storage for each prime, the size of one key switching key requires

$$2\omega\,(\ell + k)\,NB = 2(\omega\ell + \ell)NB$$

bits of memory. Trivially, increasing ω increases storage requirements.

3.5 Optimizing ω by increasing the degree N

Our previous analysis assumes a fixed degree N. But, we can also set ω as we please, choosing N securely for qP afterward. However, we then have to consider the complexity of the NTT operation itself. Although using the asymptotic bound $\mathcal{O}(N \log N)$ at face value does not quite match reality due to the hidden factors, it is sufficient for our following argument.

An optimal choice of $\omega = 2$ results in 6ℓ NTT operations for key switching. If N has to be increased to the next power-of-two degree to match a given security level, the count of NTT operations increases to $12\ell N \log(N + 1) \approx 12\ell N \log N$. Therefore, increasing the polynomial degree to reduce key switching costs is approximately worth it once

$$\left(\omega\ell + 3\ell + \frac{2\ell}{\omega}\right) N \log N > 12\ell N \log N$$

and thus $\omega^2 - 9\omega + 2 > 0$, and $\omega > 8.77$ follows. Increasing the polynomial degree to $2N$ with $\omega = 2$ (which usually performs better than $\omega = 1$, see Section 5) reduces the memory requirements for each key once $2(\omega\ell + \ell)NB > 12\ell NB$, hence $\omega > 5$. However, increasing the polynomial degree does increase the computational complexity of all other homomorphic computations, as these do not depend on the key switching parameters. We evaluate the above in Subsection 5.6 and discuss the implications for implementations holistically in Subsection 6.2.

3.6 Optimizing ω with a second secret key

A common optimization is instantly switching the modulus after an encryption. This reduces the fresh encryption error and, in the long run, can be benefitial for noise growth in a given use case. This frees b bits of the ciphertext modulus q which we can use for the key switching modulus P, which in turn enables us to possibly reduce ω. However, we do have to be careful to stay secure with respect to the RLWE assumption. To use the additional b bits securely, we set our key switching parameters P and ω as usual, then generate our key material as required for a secret key s. Now, we choose a second set of key switching parameters P' and ω' for $\log q_{1,L-1}$ and generate our key material for N and $q_{1,L-1}P'$ with a second secret key s'. For the second parameter set, we can choose

P' with $\log P' \leq \log P + b$, possibly resulting in an $\omega' < \omega$ and better key switching performance.

In addition to an initial modulus switching, we thus also perform an initial key switching to the secret key s' using the worse-performing parameters P and ω only once. Afterward, we can use the parameters P' and ω' for better key switching performance for all remaining key switchings. Note that this idea generalizes to any number of initial levels; however, this might lead to a large inflation in key switching keys depending on the use case. For example, by doing so, we might require a key switching key to remove c_2 after a multiplication for parameters qP and ω as well as for parameters P' and ω'.

3.7 Optimizing ℓ by choosing (mostly) large RNS primes

So far, our analysis assumes $b \approx \beta \approx B$, that is, RNS primes for q and P close to the maximum size B. In general, choosing the RNS primes as large as possible is beneficial for two apparent reasons:

- Each additional prime increases the number of polynomial operations during homomorphic evaluation. The larger each prime, the fewer primes we need, reducing computation time and memory costs.
- Assuming a maximum prime size of B bit, using fewer bits usually wastes computational and memory resources as operations are still performed over B-sized numbers.

Given a bound $\log q$ (the same idea extends to P), we would like to choose primes as follows: Compute $\ell = \lceil \log q/B \rceil$, choose b such that $\log q \approx \ell \cdot b$, and generate ℓ primes close to 2^b. However, for BGV and CKKS, the size of q_i significantly impacts the error growth, and scaling by roughly 2^b during modulus switching is not necessarily the optimal choice. Note that no such limitations exist for P and the above approach always works.

To introduce the desired flexibility, we revisit an idea by Gentry, Halevi, and Smart and add a few small primes, which we constantly switch in and out of the modulus during homomorphic evaluation to control the error [19]. Since their work only describes the high-level idea without any details, we introduce some additional notation for the RNS setting, integrate their idea with key switching to minimize costs, and analyze the complexity of our integration.

Switching primes in and out.

We choose $\ell - \kappa$ primes close to 2^b and $\kappa > \mu$ smaller primes with their product close to $2^{\mu b}$, such that $\log q \approx (\ell + \mu)b$; here, $2^{\mu b/\kappa}$ should correspond to the required size for controlling the error (BGV) or rescaling (CKKS). A typical homomorphic evaluation then would look as follows:

1 Receive a fresh encryption from the client.

2 Perform the desired homomorphic operations such as additions, multiplications, or rotations up to the level boundary; note that we perform a key switching after a multiplication or a rotation.

3 Apply modulus switching to reduce the error (for BGV) or rescale the message (for CKKS) using one of the κ small primes.

4 Repeat steps (2) and (3) until all κ small primes have been used for modulus switching.

5 Perform homomorphic operations for the next level. During the last key switching, replace any μ large primes with the κ small primes; this restores our capabilities for scaling with small primes.

6 Continue with step (3), using one of the κ small primes for modulus switching.

Integrating prime switching with key switching.

If we want to replace the primes $\{q_{\ell-\mu}, \ldots, q_\ell\}$ with $\{q_1, \ldots, q_\kappa\}$, we can switch the modulus of $a \in R_{q_{\kappa+1,\ell}}$ as

$$\begin{aligned} \left\lceil \frac{q_{1,\ell-\mu}}{q_{\kappa+1,\ell}} a \right\rfloor_t &= \frac{q_{1,\ell-\mu} a - t[t^{-1} q_{1,\ell-\mu} a]_{q_{\kappa+1,\ell}}}{q_{\kappa+1,\ell}} \pmod{q_{1,\ell-\mu}} \\ &= \frac{q_{1,\kappa} a - t[t^{-1} q_{1,\kappa} a]_{q_{\ell-\mu,\ell}}}{q_{\ell-\mu,\ell}} \pmod{q_{1,\ell-\mu}} \\ &\in \mathcal{R}_{q_{1,\ell-\mu}}. \end{aligned}$$

We can then integrate prime switching with modulus switching during key switching by simply switching out the large primes $\{q_{\ell-\mu}, \ldots, q_\ell\}$ in addition to the primes $\{P_1, \ldots P_k\}$. Switching for $c_i' \in R_{q_{\kappa+1,\ell} P_{1,k}}$ transforms the above to

$$\frac{q_{1,\kappa} c_i' - t[t^{-1} q_{1,\kappa} c_i']_{q_{\ell-\mu,\ell} P_{1,k}}}{q_{\ell-\mu,\ell} P_{1,k}} \pmod{q_{1,\ell-\mu}}.$$

Compared to switching out only $P_{1,k}$ and switching in no primes, the number of inverse NTT operations increases from $2k$ to $2(\mu + k)$ for base extending $\delta_i = -t[t^{-1} q_{1,\kappa} c_i']_{q_{\ell-\mu,\ell} P_{1,k}}$. However, we also save 2μ NTT operations on either c_i' or δ_i since we reduce the number of primes for the modulus switching output from ℓ to $\ell - \mu$. For output in the coefficient domain, switching primes in and out requires no additional NTT operations. For output in the NTT domain, we need to perform 2κ additional forward NTT operations (κ operations per δ_i). The former is free regarding the number of NTT operations, and, with $\kappa \in \mathcal{O}(1)$, the latter has only a small overhead.

Choosing the primes.

Based on this more generic setup, we adjust the process for choosing the primes q_i as follows: Compute the number of RNS primes $\ell' = \lceil \log q/B \rceil$, and choose b such that $\log q \approx \ell' \cdot b$. Then, we generate $\ell' - \mu$ primes close to 2^b and κ primes close to $2^{\mu b/\kappa}$. This results in $\ell = \ell' - \mu + \kappa$ ciphertext primes. Compared to the more naïve approach, that is choosing all primes of size $\mu b/\kappa$, we reduce ℓ (and thus computational and memory complexity) as long as

$$\left\lceil \frac{\log q}{\mu b/\kappa} \right\rceil \approx \left\lceil \frac{\ell'}{\mu/\kappa} \right\rceil > \ell \;\Leftrightarrow\; \kappa\ell' - \mu\ell \geq \mu \;\Leftrightarrow\; \ell \geq \kappa + \frac{\mu}{\kappa - \mu}.$$

For example, with $B = 60$, we consider a use-case scaling each level with 36-bit primes; then, choosing $\mu = 2$ and $\kappa = 3$ results in $b = 54$, which reduces the overall number of primes as soon as $\ell \geq 5$ (equivalent to $\log q > 144$). Evaluating the above idea generically for all parameter settings is rather difficult due to the use-case-specific nature of parameters for BGV-like schemes; however, choosing primes as large as possible with a small number of additional scaling factors can also be adapted to more complex scenarios.

Double-decomposition key switching
Section 4

We now extend our previous analysis to the double-decomposition technique following the same structure as before: We start with generalizing input and output domain, follow with complexity analysis and conclude with strategies for parameter optimization.

4.1 Generalizing NTT complexity to arbitrary domains

In contrast to the single-decomposition technique, the double-decomposition technique always requires input and output in the coefficient domain. For our initial ciphertext extension, we require the input in the coefficient domain. Thus, for input in the NTT domain, we apply ℓ inverse NTTs for c_1 (after a rotation) or c_2 (after a multiplication) before extending the ciphertext[2].

As with the single-decomposition technique, we switch a modulus with

$$\left(c_0 + \frac{c'_0 + \delta_0}{P},\; c_1 + \frac{c'_1 + \delta_1}{P}\right);$$

2 In the single-decomposition technique, we save ℓ forward NTT operations later on by re-using the input, hence, the number of NTT operations stays the same; this is not possible in the double-decomposition technique. For the precise details, we refer to the in-depth description of Kim, Polyakov, and Zucca [25] and our accompanying key switching implementation.

Table 4: Double-decomposition key switching complexity regarding NTT operations generalized to arbitrary input and output domains, either applied after a rotation or a multiplication.

Operation	Domain		Cost
Rotation	coef	coef	$(\omega + 2\tilde{\omega})r$
	coef	ntt	$(\omega + 2\tilde{\omega})r + 2\ell$
	ntt	coef	$(\omega + 2\tilde{\omega})r + 2\ell$
	ntt	ntt	$(\omega + 2\tilde{\omega})r + 3\ell$
Multiplication	coef	coef	$(\omega + 2\tilde{\omega})r$
	coef	ntt	$(\omega + 2\tilde{\omega})r + 2\ell$
	ntt	coef	$(\omega + 2\tilde{\omega})r + 3\ell$
	ntt	ntt	$(\omega + 2\tilde{\omega})r + 3\ell$

however, in contrast to before, c_i' and δ_i are both in the coefficient domain. For output in the NTT domain, we thus require 2ℓ forward NTTs for $c_i' + \delta_i$ (possibly adding Pc_i for input in the coefficient domain). For input in the NTT domain and output in the coefficient domain, we apply ℓ inverse NTTs for c_0 and, for multiplications, another ℓ inverse NTTs for c_1. We summarize the generalized complexity of the double-decomposition technique in Table 4.

4.2 Analyzing NTT and multiplication complexity

As with the single-decomposition technique, Kim et al. [26] argue as follows about the asymptotic complexity of the double-decomposition technique: By choosing the $k, r \in \mathcal{O}(1)$, we choose $\omega, \tilde{\omega} \in \mathcal{O}(\ell)$ and the bound $\mathcal{O}(\ell)$ follows accordingly. To extend our new perspective to the double-decomposition technique, we first need to estimate the number of primes r in E. In the original work, the authors use the infinity norm; we use the canonical norm which tends to result in tighter bounds. Assuming $b \approx \beta \approx B$, E needs to be large enough to hold a product $ab \in R$ with $V_a \approx 2^{\ell/\omega b}$ and $V_b \approx 2^{(\ell+k)/\tilde{\omega} b}$. Then, for $N = 2^n$, we can roughly upper bound $\log E$ as

$$\log\left(D\sqrt{NV_{ab}}\right) = (n-1) + (\ell/\omega + \ell/\tilde{\omega} + \ell/(\omega\tilde{\omega}))b.$$

With n/b negligible in practice, a reasonable estimate for the number of primes in E is

$$r = \frac{\omega\ell + \tilde{\omega}\ell + \ell}{\omega\tilde{\omega}}.$$

NTT complexity.

To minimize the number of NTT operations, we want to minimize the term

$$(\omega + 2\tilde{\omega})r = \left(\frac{\omega^2 + 3\omega\tilde{\omega} + 2\tilde{\omega}^2 + \omega + 2\tilde{\omega}}{\omega\tilde{\omega}}\right)\ell.$$

However, minimizing this term for ω and $\tilde{\omega}$ over $\mathbb{R}$ results in values outside the desired target range, and we instead bound ℓ and use an exhaustive search over the integers to find solutions for ω and $\tilde{\omega}$. For $\ell \leq 200$, a generous bound on the number of ciphertext primes, the number of NTT operations is minimal for

$$\omega = \ell \quad \text{and} \quad \tilde{\omega} = \sqrt{\left(\frac{\ell+1}{2}\right)\ell}.$$

Multiplication complexity.

We also apply our folding techniques to the double-decomposition technique. Overall, it requires $(\ell + 2\omega\tilde{\omega})r + \ell(2k+3)$ multiplications, which, using the estimates on k and r, transforms to

$$\ell\left(2\omega + 2\tilde{\omega} + \frac{3\ell}{\omega} + \frac{\ell}{\tilde{\omega}} + \frac{\ell}{\omega\tilde{\omega}} + 5\right).$$

As with the NTT complexity, we exhaustively search for a local minimum for $\ell \leq 200$. The resulting optimal choices for ω and $\tilde{\omega}$ are close to $\sqrt{\ell}$.

4.3 Analyzing memory costs

For the double-decomposition technique, one key switching key requires

$$2\omega\tilde{\omega}rNB = 2(\omega\ell + \tilde{\omega}\ell + \ell)NB$$

bits, and increasing ω or $\tilde{\omega}$ increases the size of the key switching key. In contrast to the single-decomposition technique, where reducing ω benefits NTT and memory complexity, we now have a trade-off between memory size and NTT performance.

4.4 Optimizing parameters

Due to the trade-off between NTT and memory complexity, choosing optimal parameters for the double-decomposition technique is more complex. One reason is that increasing the size of the key switching key also increases execution time since we have to read more data from memory. This is also observable in hardware accelerators significantly reducing the execution time for the NTT; this results in key switching being limited by the speed of reading data from memory [9]. Still, the easiest strategy for choosing parameters in the double-decomposition technique is setting $\omega = \ell$ and $\tilde{\omega} = \sqrt{\ell(\ell+1)/2}$. A better, but platform- and implementation-specific strategy is as follows:

1. Benchmark execution times for a NTT operation and a multiplication on the chosen platform using the selected implementation.
2. Benchmark reading NB bits from memory.
3. Exhaustively search for optimal solutions by combining the number of operations/the amount of data to be read with the previously obtained benchmark results.

Note that the idea of choosing mostly large primes to reduce ℓ also applies to the double-decomposition technique.

Evaluation
Section 5

We evaluate our contributions using a comprehensive set of benchmarks to answer the following questions:

- How costly is it to replace primes during key switching?
- How fast is key switching with mostly large primes?
- When is the double-decomposition technique better?
- Is $\omega = 1$ or $\omega = 2$ better for the single-decomposition technique?
- At which point is it worth it to increase the degree N in order to reduce ω?
- How much speed-up do we gain with our novel ideas on constant folding?

We start by describing our benchmarking setup and explaining our library choice, afterward answering each question in a dedicated subsection.

5.1 Benchmarking setup

We run all our benchmarks on Ubuntu 20.04.5. The Central Processing Unit (CPU) is an Intel Core i9-7900X CPU at 3.3 GHz with 20 cores, and the system features 64 GiB of available memory. We disable Intel turbo boost, and pin the benchmarking execution to a single CPU core. For the polynomial degrees $N \in \{2^{14}, \dots 2^{17}\}$, the closed formula by Mono et al. [28] outputs $\{433, 867, 1735, 3470\}$ as upper bounds for $\log qP$, respectively[3]. For our benchmarks, we only consider secure parameter sets.

3 These bounds are slightly tighter than in the Homomorphic Encryption Standard [1], which only provides bounds for $N \in \{2^{10}, \dots, 2^{15}\}$. Since the standard's initial release, more and more published use cases require $N \geq 2^{15}$, which we account for in our evaluation by shifting our degree range to larger degrees.

Our state-of-the-art implementation[4] uses the open-source BGV library fhelib which we choose for its highly flexibly Application Programming Interface (API) and ease-of-use. For fast polynomial arithmetic, fhelib uses a HEXL-based ring layer [22] which is a common denominator with homomorphic encryption libraries such as OpenFHE [4] or SEAL [30], and implementing our improvements in these libraries should result in speed-ups comparable to the ones we observe in fhelib. Note that we do not measure key switching key sizes because, in fhelib, these match the theoretical values. This, however, obviously depends on the specific data structures used and their overhead in a given implementation.

5.2 Replacing primes during key switching

To find out how costly replacing primes is, we compare execution times for our optimized implementation (`folded`) with an optimized implementation replacing μ large primes with κ small primes with the single-decomposition technique (`switch`). We run benchmarks for parameter sets using $1/2$, $2/3$, and $3/4$ of the available modulus space for the ciphertext modulus q with ω as small as possible and at least $\kappa = 3$ primes of size 36 bit and as many 54 bit primes as possible. For $\log q = 1116$, for example, we use four primes of size 36 bit and 18 primes with size 54 bit.

We average execution times across all combinations of input and output domain and report the results in Table 5; the overhead reports the absolute difference between between the `folded` and the `switch` version. Overall, switching primes does have a low overhead, especially considering the fact that we only have to do so every κ levels. For a given degree N, the overhead is relatively constant. The relative overhead, as expected, gets smaller the more primes we use.

5.3 Choosing mostly large primes

We now measure how large our performance gains are by choosing mostly large primes. We re-use the parameter sets as defined in Subsection 5.2, that is using $1/2$, $2/3$, and $3/4$ of the maximum modulus size for q. We now compare how choosing only small primes with $b = 36$ and choosing mostly large primes with $b = 54$ and $\kappa = 3$ affects execution times.

As before, we average execution times across all combinations of input and output domain, the results are in Table 6; we also report the relative speed-up. On average, choosing mostly large primes provides a speed-up of $36.9\,\%$ across all measured parameter sets. Note that the speed-ups here are much larger than the overhead of switching in primes resulting in a highly effective method for improving performance.

---

4 https://github.com/Chair-for-Security-Engineering/owl

Table 5: Execution times for parameter sets using 1/2, 2/3, and 3/4 of the available modulus space for the ciphertext modulus q. For each parameter set, we choose mostly primes with 54 bit, but at least $\kappa = 3$ primes with 36 bit. The reported execution times are an average over all combinations of input and output domain for the folded implementation.

Parameters		Time (ms)		
$\log N$	$\log q$	`folded`	`switch`	overhead
14	252	4.0	4.7	0.7
14	288	5.0	5.7	0.7
15	396	15.9	18.1	2.2
15	540	21.9	24.3	2.4
15	612	29.4	32.1	2.7
16	828	95.7	104.9	9.2
16	1116	170.1	178.2	8.1
16	1260	209.8	216.5	6.7
17	1692	783.0	818.0	35.0
17	2268	1013.1	1046.0	32.9
17	2556	1262.1	1302.6	40.5

5.4 Comparing decomposition techniques

Our goal is to compare how well both decomposition techinques would perform in real-world parameter settings. To do so, we fix a degree N and compare performance in both techniques for different moduli sizes; this increases the lower bound on ω the closer we get to the maximum modulus size. We generate parameter sets using 50 %, 65 %, 75 %, 80 %, 85 %, 90 %, and 95 % of the available modulus space for the ciphertext modulus q. We choose key switching parameters to optimize for the NTT complexity of each technique. As before, we compare execution times across all combinations of input and output domain.

We compare both decomposition techniques in Figure 1, the blue line represents the single-decomposition technique while the green line represents the double-decomposition technique. As we can see, the double-decomposition technique only becomes competitive when we get close to the maximum modulus size for q. In fact, in our experiments, the single-decomposition technique outperforms the double-decomposition technique for all parameter sets except for $\log N = 16$ with $\log q = 1624$. We therefore recommend to use the single-decomposition technique for key switching; an exception might be for very large ω.

Table 6: Execution times for parameter sets using 1/2, 2/3, and 3/4 of the available modulus space for the ciphertext modulus q. For each parameter set, we either choose only small primes with $b = 36$ or choose mostly primes with $b = 54$, but at least $\kappa = 3$ primes of size 36 bit. The reported execution times are an average over all combinations of input and output domain for the folded implementation.

Parameters		Time (ms)		Speed-up
$\log N$	$\log q$	small	large	
14	252	4.7	4.0	17.5 %
14	288	5.9	5.0	18.0 %
15	396	18.8	15.9	18.2 %
15	540	29.3	21.9	33.8 %
15	612	38.3	29.4	30.3 %
16	828	156.4	95.7	63.4 %
16	1116	241.1	170.1	41.7 %
16	1260	301.9	209.8	43.9 %
17	1692	1065.6	783.0	36.1 %
17	2268	1529.5	1013.1	51.0 %
17	2556	1917.1	1262.1	51.9 %

5.5 Choosing $\omega = 1$ or $\omega = 2$

We decide whether $\omega = 1$ or $\omega = 2$ is better for the single-decomposition technique using 50 % of the available modulus space for q and comparing the optimized key switching implementations; we list our results in Table 7. For small N and q, the difference between setting $\omega = 1$ and $\omega = 2$ is negligible and within the margin of measurement error. For larger N and q, however, the difference is rather large. There are most likely two reason: First, $\omega = 2$ is in general closer to the optimal multiplication complexity $\omega = \sqrt{l}$ than $\omega = 1$; this reduces the time spent on modular multiplication. Second, the key switching keys are smaller for $\omega = 1$ and reading the keys from memory has a larger impact on performance for smaller parameters. Overall, we recommend choosing $\omega \geq 2$ as small as possible for best performance.

5.6 Increasing the degree N

To evaluate increasing the degree N, we re-use the parameter sets from Subsection 5.4 using 90 % and 95 % of the available modulus space for q. For these parameter sets, we create a sibling set for the degree $2N$ and $\omega = 2$. We list execution times averaged over all combinations of input and output domain in Table 8 and list speed-ups in percent.

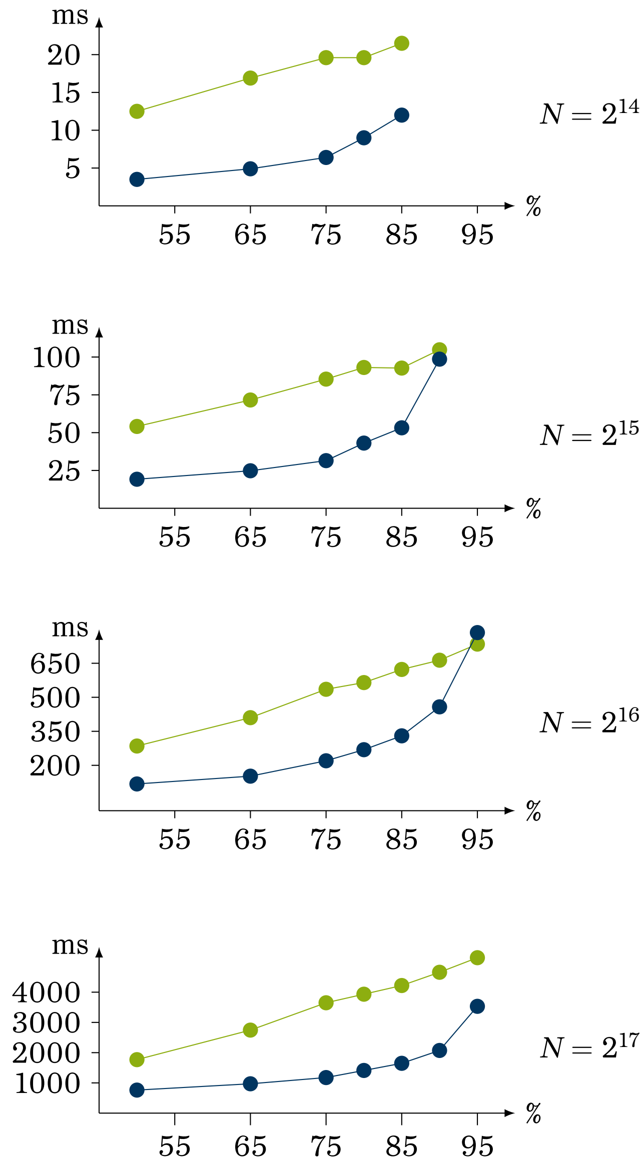

Figure 1: Comparison of execution times for optimized implementations of the single- and double-decomposition technique. We use different percentages of the available modulus space for the ciphertext modulus q, choosing key switching parameters optimally with respect to the NTT complexity. Execution times are averaged across all combinations of input and output domain.

While increasing the degree can speed up execution times by up to 25 %, it can also do more harm then good even when only considering key switching. Since increasing the degree increases costs everywhere else, we only recommend doing so with benchmarking results confirming a speed-up for a use-case specific implementation. We refer to Subsection 6.2 for additional remarks on increasing the degree.

Table 7: Execution times for parameter sets using 50 % of the available modulus space for the ciphertext modulus q with $\omega = 1$ and $\omega = 2$. The reported execution times are an average over all combinations of input and output domain for the folded implementation. We also report the speed-up in percent.

Parameters		Time (ms)		Speed-up
$\log N$	$\log q$	$\omega = 1$	$\omega = 2$	
14	216	3.5	3.6	-2.7 %
15	432	19.2	19.1	0.5 %
16	855	118.0	113.3	4.1 %
17	1711	762.1	671.0	13.6 %

Table 8: Execution times for parameter sets using 90 % and 95 % of the available modulus space for the ciphertext modulus q. For each parameter set, we also generate a parameter set for the degree $2N$ with $\omega = 2$. The reported execution times are an average over all combinations of input and output domain for the folded implementation. We also report the speed-up in percent.

Parameters			Time (ms)		Speed-up
$\log N$	$\log q$	ω	N	$2N$	
15	780	10	98.6	80.7	22.2 %
16	1560	10	457.9	564.6	-18.9 %
16	1624	19	785.3	626.5	25.3 %
17	3120	9	2072.6	3172.6	-34.7 %
17	3300	19	3532.2	3462.4	2.0 %

5.7 Speed-ups from constant folding

We measure speed-ups from constant folding across a range of parameters. As in Subsection 5.4, we use a certain percentage of the available modulus space for q ranging from 50 % to 85 %. We use a non-optimized implementation of key switching (`naïve`) and compare execution times to our optimized implementation (`folded`) averaged across all combinations of input and output domain; the results are in Table 9. As we can observe, we improve execution times for key switching by up to 11.6 % using our constant folding techniques and, on average, speed up key switching by 4.8 %.

Table 9: Execution times for parameter sets using 50 %, 65 %, 75 %, 80 %, and 85 % of the available modulus space for the ciphertext modulus q using optimal key switching parameters with respect to NTT complexity. The reported execution times are an average over all combinations of input and output domain comparing the `naïve` implementation with the `folded` implementation with the speed-up in percent.

Parameters			Time (ms)		Speed-up
$\log N$	$\log q$	ω	`naïve`	`folded`	
14	216	1	3.9	3.5	11.6 %
14	280	2	5.3	4.9	8.3 %
14	324	3	6.8	6.4	6.6 %
14	342	5	9.3	9.0	3.5 %
15	432	1	20.9	19.2	8.8 %
15	560	2	26.6	24.8	7.4 %
15	649	3	33.0	31.5	4.8 %
15	684	5	45.2	43.1	4.8 %
15	728	6	54.5	53.2	2.5 %
16	855	1	125.5	118.0	6.3 %
16	1121	2	159.2	152.3	4.5 %
16	1298	3	227.0	220.1	3.1 %
16	1380	5	275.6	269.0	2.4 %
16	1475	6	335.6	330.1	1.7 %
17	1711	1	801.5	762.1	5.2 %
17	2242	2	996.4	970.5	2.7 %
17	2580	3	1207.5	1176.4	2.7 %
17	2760	5	1444.6	1411.2	2.4 %
17	2940	6	1665.0	1647.7	1.0 %

Discussion
Section 6

In the following, we discuss multiple aspects of our work, starting with the relevance of key switching for homomorphic use cases. We also discuss the implications of increasing the polynomial degree holistically, highlight some limitations of our work, and explore opportunities for future work.

6.1 Performance impact of key switching

In the FHE community, it is common knowledge that key switching is one of the most expensive parts of homomorphic evaluation for BGV-like

schemes. First, for most use cases, we perform many key switchings, as performing a desired computation often involves many rotations, even for low-level circuits with only a handful of modulus switching. One such example is homomorphic matrix multiplication [27], where a profiling run of a highly optimized implementation on our benchmarking setup shows that more than 50 % of execution time is spent in key switching.

Second, on the server side, key switching and modulus switching are the only homomorphic operations requiring forward and inverse NTT operations, the main computational bottleneck for homomorphic encryption, with respective asymptotic complexities of $\mathcal{O}(\omega\ell)$ and $\mathcal{O}(\ell)$. For modulus switching, however, we often can amortize most of the costs by merging it with key switching.

Third, key switching requires reading large keys from memory, which are only used for a single modular multiplication. This is especially relevant in hardware accelerators, as once the computational bottlenecks are accelerated, memory becomes the main problem even for custom-designed memory architectures [9, 16].

Fourth, we often require key switching due to the algorithmic mappings from the unencrypted domain to the homomorphic realm, where many values are encoded in a single message polynomial. Operations between values encoded in different positions require different permutations on the ciphertext and, hence, a key switching to get back to a shared secret key.

Thus, improving key switching boosts performance significantly in most homomorphic use cases. When optimizing use cases, another result of our work comes in handy: Sometimes, a use case benefits from key switching output with mixed output domains (output for some RNS primes in the NTT domain and some in the coefficient domain), which we show how to achieve for free using our folding optimizations.

6.2 Increasing the polynomial degree

Contrary to current folklore, increasing the polynomial degree can increase performance for otherwise very large ω. However, although execution time decreases for key switching, the computational complexity of other operations increases with larger N. The same holds for memory: we only reduce costs for key switching (such as the key switching keys or the temporary storage for the extended polynomials during key switching), increasing all other ciphertext storage, permanent or temporary.

We believe that, in addition to a large ω, the following conditions should be fulfilled before considering an increase of the polynomial degree:

1. The main bottleneck of the use case is the key switching operation;

2. the use case requires many rotations, each with their own unique key switching key; and
3. the key switching operation is memory-bound, for example, in hardware.

The double-decomposition technique can also be better than increasing the polynomial degree for large ω. However, key switching keys for the double-decomposition technique are by default larger than for the single-decomposition technique with the same parameters or require additional computations for every use [26]; therefore, an increased polynomial degree with a more straightforward key switching implementation and the smaller keys might be more beneficial, especially for hardware implementations.

6.3 Limitations and future work

One limitation of our work is the somewhat simplified approach for selecting key switching parameters ω and $\tilde{\omega}$ for the double-decomposition technique. In the single-decomposition technique, choosing $\omega \geq 2$ as small as possible is beneficial regarding NTT and memory complexity. In contrast, the double-decomposition technique requires a trade-off between NTT complexity ($\omega = \ell$) and memory requirements ($\omega = 1$), which also has performance implications. We therefore recommend to use our platform- and implementation-specific approach for a ciphertext modulus using over 90 % of the available modulus.

Considering future work, one open problem is finding closed formulas for the multiplication complexity in the double-decomposition technique instead of rough estimates for ω and $\tilde{\omega}$. We also did not explore whether our folding optimizations positively impact the error correction after a fast base extension with the BEHZ method. Selecting the best correction method for a given platform could also be part of optimizing parameters for the double-decomposition technique.

Conclusion
Section 7

Int this work, we provide theoretical and practical improvements to key switching, the most expensive primitive in BGV-like schemes. We provide an in-depth analysis for parameter selection and, using a new perspective, introduce an improved asymptotic complexity of $\mathcal{O}(\omega\ell)$. Additionally, we explore multiple opportunities for reducing the parameters ω and ℓ for state-of-the-art key switching: We revisit an idea by Gentry, Halevi, and Smart [19] reducing execution times by up to 63 %, dispel the myth that increasing the degree N always results in worse performance,

and introduce a new idea of using two seperate parameter sets for improved performance.

We also correct analysis by Kim et al. [26], extend our new perspective to their double-decomposition technique, and show the technique's insignificance for practical parameter settings. We also highlight new opportunities for constant folding which speed up key switching execution times by up to 11.6 %. Overall, our work provides a new perspective on the most expensive homomorphic primitive in BGV-like schemes, key switching, and improves the current state-of-the-art in theory and practice.

References

[1] Martin R. Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin E. Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. "Homomorphic Encryption Standard". In: IACR Cryptol. ePrint Arch. 2019.939 (2019). URL: https://eprint.iacr.org/2019/939.

[2] Martin R. Albrecht, Rachel Player, and Sam Scott. "On the concrete hardness of Learning with Errors". In: J. Math. Cryptol. 9.3 (2015), pp. 169–203. URL: http://www.degruyter.com/view/j/jmc.2015.9.issue-3/jmc-2015-0016/jmc-2015-0016.xml.

[3] Jacob Alperin-Sheriff and Chris Peikert. *Practical Bootstrapping in Quasilinear Time*. Oct. 2013. URL: https://eprint.iacr.org/2013/372.

[4] Ahmad Al Badawi, Jack Bates, Flávio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, R. V. Saraswathy, Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. "OpenFHE: Open-Source Fully Homomorphic Encryption Library". In: Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Los Angeles, CA, USA, 7 November 2022. Ed. by Michael Brenner, Anamaria Costache, and Kurt Rohloff. ACM, 2022, pp. 53–63. DOI: 10.1145/3560827.3563379. URL: https://doi.org/10.1145/3560827.3563379.

[5] Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. "A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes". In: Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers. Ed. by Roberto Avanzi and Howard M. Heys. Vol. 10532. Lecture Notes in Computer Science. Springer, 2016, pp. 423–442. DOI: 10.1007/978-3-319-69453-5_23. URL: https://doi.org/10.1007/978-3-319-69453-5_23.

[6] Zvika Brakerski. "Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP". In: Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings. Ed. by Reihaneh Safavi-Naini and Ran Canetti.

Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 868–886. DOI: 10.1007/978-3-642-32009-5_50. URL: https://doi.org/10.1007/978-3-642-32009-5_50.

[7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) Fully Homomorphic Encryption without Bootstrapping". In: ACM Trans. Comput. Theory 6.3 (2014), 13:1–13:36. DOI: 10.1145/2633600. URL: https://doi.org/10.1145/2633600.

[8] Zvika Brakerski and Vinod Vaikuntanathan. "Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages". In: Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings. Ed. by Phillip Rogaway. Vol. 6841. Lecture Notes in Computer Science. Springer, 2011, pp. 505–524. DOI: 10.1007/978-3-642-22792-9_29. URL: https://doi.org/10.1007/978-3-642-22792-9%5C_29.

[9] Leo de Castro, Rashmi Agrawal, Rabia Tugce Yazicigil, Anantha P. Chandrakasan, Vinod Vaikuntanathan, Chiraag Juvekar, and Ajay Joshi. "Does Fully Homomorphic Encryption Need Compute Acceleration?" In: IACR Cryptol. ePrint Arch. 2021.1636 (2021). URL: https://eprint.iacr.org/2021/1636.

[10] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. "A Full RNS Variant of Approximate Homomorphic Encryption". In: Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers. Ed. by Carlos Cid and Michael J. Jacobson Jr. Vol. 11349. Lecture Notes in Computer Science. Springer, 2018, pp. 347–368. DOI: 10.1007/978-3-030-10970-7_16. URL: https://doi.org/10.1007/978-3-030-10970-7_16.

[11] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. "Homomorphic Encryption for Arithmetic of Approximate Numbers". In: Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10624. Lecture Notes in Computer Science. Springer, 2017, pp. 409–437. DOI: 10.1007/978-3-319-70694-8_15. URL: https://doi.org/10.1007/978-3-319-70694-8_15.

[12] Ana Costache and Nigel P. Smart. "Which Ring Based Somewhat Homomorphic Encryption Scheme is Best?" In: Topics in Cryptology - CT-RSA 2016 - The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings. Ed. by Kazue Sako. Vol. 9610. Lecture Notes in Computer Science. Springer, 2016, pp. 325–340. DOI: 10.1007/978-3-319-29485-8_19. URL: https://doi.org/10.1007/978-3-319-29485-8_19.

[13] Anamaria Costache, Kim Laine, and Rachel Player. "Evaluating the Effectiveness of Heuristic Worst-Case Noise Analysis in FHE". In: Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part II. Ed. by Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider. Vol. 12309. Lecture Notes in Computer Science. Springer, 2020,

pp. 546–565. DOI: 10.1007/978-3-030-59013-0_27. URL: https://doi.org/10.1007/978-3-030-59013-0_27.

[14] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. "Multiparty Computation from Somewhat Homomorphic Encryption". In: Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 643–662. DOI: 10.1007/978-3-642-32009-5_38. URL: https://doi.org/10.1007/978-3-642-32009-5_38.

[15] Junfeng Fan and Frederik Vercauteren. "Somewhat Practical Fully Homomorphic Encryption". In: IACR Cryptol. ePrint Arch. (2012), p. 144. URL: http://eprint.iacr.org/2012/144.

[16] Robin Geelen, Michiel Van Beirendonck, Hilder V. L. Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios Dimou, Ingrid Verbauwhede, Frederik Vercauteren, and David W. Archer. "BASALISC: Flexible Asynchronous Hardware Accelerator for Fully Homomorphic Encryption". In: IACR Cryptol. ePrint Arch. 2022.657 (2022). URL: https://eprint.iacr.org/2022/657.

[17] Craig Gentry. "Fully homomorphic encryption using ideal lattices". In: Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009. Ed. by Michael Mitzenmacher. ACM, 2009, pp. 169–178. DOI: 10.1145/1536414.1536440. URL: https://doi.org/10.1145/1536414.1536440.

[18] Craig Gentry, Shai Halevi, and Nigel P. Smart. "Fully Homomorphic Encryption with Polylog Overhead". In: Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. Lecture Notes in Computer Science. Springer, 2012, pp. 465–482. DOI: 10.1007/978-3-642-29011-4_28. URL: https://doi.org/10.1007/978-3-642-29011-4_28.

[19] Craig Gentry, Shai Halevi, and Nigel P. Smart. "Homomorphic Evaluation of the AES Circuit". In: Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 850–867. DOI: 10.1007/978-3-642-32009-5_49. URL: https://doi.org/10.1007/978-3-642-32009-5_49.

[20] Shai Halevi, Yuriy Polyakov, and Victor Shoup. "An Improved RNS Variant of the BFV Homomorphic Encryption Scheme". In: Topics in Cryptology - CT-RSA 2019 - The Cryptographers' Track at the RSA Conference 2019, San Francisco, CA, USA, March 4-8, 2019, Proceedings. Ed. by Mitsuru Matsui. Vol. 11405. Lecture Notes in Computer Science. Springer, 2019, pp. 83–105. DOI: 10.1007/978-3-030-12612-4_5. URL: https://doi.org/10.1007/978-3-030-12612-4_5.

[21] Kyoohyung Han and Dohyeong Ki. "Better Bootstrapping for Approximate Homomorphic Encryption". In: Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings. Ed. by Stanislaw Jarecki. Vol. 12006. Lecture Notes in Computer Science. Springer, 2020, pp. 364–390. DOI: 10.1007/978-3-030-40186-3_16. URL: https://doi.org/10.1007/978-3-030-40186-3_16.

[22] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, Vinodh Gopal, et al. *Intel HEXL (release 1.2)*. https://github.com/intel/hexl. Sept. 2021.

[23] Xiaoqian Jiang, Miran Kim, Kristin E. Lauter, and Yongsoo Song. "Secure Outsourced Matrix Computation and Application to Neural Networks". In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM, 2018, pp. 1209–1222. DOI: 10.1145/3243734.3243837. URL: https://doi.org/10.1145/3243734.3243837.

[24] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. "Approximate Homomorphic Encryption with Reduced Approximation Error". In: Topics in Cryptology - CT-RSA 2022 - Cryptographers' Track at the RSA Conference 2022, Virtual Event, March 1-2, 2022, Proceedings. Ed. by Steven D. Galbraith. Vol. 13161. Lecture Notes in Computer Science. Springer, 2022, pp. 120–144. DOI: 10.1007/978-3-030-95312-6_6. URL: https://doi.org/10.1007/978-3-030-95312-6%5C_6.

[25] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. "Revisiting Homomorphic Encryption Schemes for Finite Fields". In: Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part III. Ed. by Mehdi Tibouchi and Huaxiong Wang. Vol. 13092. Lecture Notes in Computer Science. Springer, 2021, pp. 608–639. DOI: 10.1007/978-3-030-92078-4_21. URL: https://doi.org/10.1007/978-3-030-92078-4_21.

[26] Miran Kim, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. *Accelerating HE Operations from Key Decomposition Technique*. June 2023. URL: https://eprint.iacr.org/2023/413.

[27] Johannes Mono and Tim Güneysu. "Implementing and Optimizing Matrix Triples with Homomorphic Encryption". In: Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS 2023, Melbourne, VIC, Australia, July 10-14, 2023. Ed. by Joseph K. Liu, Yang Xiang, Surya Nepal, and Gene Tsudik. ACM, 2023, pp. 29–40. DOI: 10.1145/3579856.3590344. URL: https://doi.org/10.1145/3579856.3590344.

[28] Johannes Mono, Chiara Marcolla, Georg Land, Tim Güneysu, and Najwa Aaraj. "Finding and Evaluating Parameters for BGV". In: Progress in Cryptology - AFRICACRYPT 2023 - 14th International Conference on Cryptology in Africa, Sousse, Tunisia, July 19-21, 2023, Proceedings. Ed. by Nadia El Mrabet, Luca De Feo, and Sylvain Duquesne. Vol. 14064. Lecture Notes in Computer Science. Springer, 2023, pp. 370–394. DOI: 10.1007/978-3-031-37679-5_16. URL: https://doi.org/10.1007/978-3-031-37679-5_16.

[29] Ronald Rivest, Len Adleman, and Michael Dertouzos. *On Data Banks and Privacy Homomorphism*. 1978.

[30] *Microsoft SEAL (release 4.1).* https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA. Jan. 2023.