A New Perspective on Key Switching for BGV-like Schemes

Johannes Mono ● Tim Güneysu ●●
johannes.mono@rub.de tim.gueneysu@rub.de
0000-0002-0839-058X 0000-0002-3293-4989

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

Fully homomorphic encryption is a promising approach when computing on encrypted data, especially when sensitive data is involved. 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 optimizing other aspects of these schemes, key switching has gone largely untouched. One exception has been a recent work [26] introducing a new double-decomposition technique. Its contributions are a great addition to the current state-of-the-art with one flaw: The authors take a skewed perspective on key switching parameters and their asymptotic complexity leading to incorrect conclusions about how effective their approach really is. In this work, we deep dive into key switching and correct, enhance, and improve the current state-of-the-art. We provide a new perspective on the key switching parameters P, ω, and $\tilde{\omega}$ resulting in the asymptotic bounds $\mathcal{O}(\omega\ell)$ and $\mathcal{O}(\omega\ell/\tilde{\omega} + \tilde{\omega}\ell/\omega)$ for the single- and double-decomposition technique, respectively. We also revisit an idea by Gentry, Halevi, and Smart [18] to reduce the number of multiplications, which speeds up key switching by up to 63 % and up to 11.6 %, respectively.

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 variety of other applicable areas. One such area 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; others many additions, but only a few multiplications. For arbitrary computations, however, any amount of additions *and* multiplications was needed.

In 2009, Gentry introduced an ingenious idea, bootstrapping, and gave birth to the first ever encryption scheme for privacy homomorphism [16]. 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 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 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 primitive, 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 [5, 14], BGV [6], and CKKS [10]. 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 power-of-two 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 e for a known scaling factor t:

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

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 co-prime 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 concerns the output ciphertext and its decryption. The sum

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

only requires s for decryption. The product

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

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

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

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

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

and only requires s for decryption. We also switch keys after our third primitive operation, rotations. To rotate an encrypted vector, we apply a permutation π on the ciphertext as

$$\pi(c(s)) = \pi(c_0) + \pi(c_1) \cdot \pi(s).$$

As with a multiplication, we transform $\pi(c_1) \cdot \pi(s)$ to $\tilde{c}_0 + \tilde{c}_1 \cdot s$ 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. 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, the same cannot be said for the key switching parameters P and ω [2, 11, 1, 12, 28].

1.1 Related Work

Key switching is the most expensive primitive in BGV-like scheme. It occupies roughly 40 % of execution time during bootstrapping and is $11\times$ slower compared to a naïve ciphertext multiplication[1]. But, despite its high costs, works on key switching are a rare sight. In the appendix of their extended version, Kim, Polyakov, and Zucca [25] explore the current state-of-the-art on key switching. They describe two different techniques, the BV technique [7] and the GHS technique [17] as well as their combination to the hybrid technique (which we will refer to as single-decomposition technique). They analyse computational and memory complexity, but do not extend their analysis from correct parameter selection to optimal parameter selection. Han and Ki [21] shortly discuss trade-offs for the parameter P on a high-level, but do not show how to choose parameters optimally. Kim et al. at [26] propose an extension to the current state-of-the-art with a double-decomposition technique, but with one major shortcoming: a flawed comparison with the single-decomposition technique.

1.2 Contributions

In this work, we take an in-depth look at key switching and provide answers to the following open questions:

---

1 using our benchmarking setup with OpenFHE and fhelib (see also Section 4)

1 Do I want to implement the more complex double-decomposition technique? If yes, when do I want to use it?

2 Do I always want to stick with a given N and q in the single-decomposition technique? Or can I adjust them to get better performance?

3 How do I set the parameters P and ω for best performance?

In the process, we make the following contributions:

- We provide a new perspective on the single-decomposition technique with the bound $\mathcal{O}(\omega\ell)$. We confirm our theoretical results with benchmarks and introduce new guidelines for parameter selection.
- We extend the original work [26] on the double-decomposition technique with the bound $\mathcal{O}(\omega\ell/\tilde{\omega} + \tilde{\omega}\ell/\omega)$ and correct the comparison with the single-decomposition technique.
- We integrate an idea by Gentry, Halevi, and Smart [18] with key switching resulting in up to 63 % faster execution times.
- We highlight new opportunities for folding multiplications in key switching resulting in up to 11.6 % faster execution times.

Preliminaries
Section 2

Understanding our contributions requires an understanding of key switching and related concepts. For experts, we provide a short summary at the end including commonly used notation (see Subsection 2.6). For more unfamiliar readers, we will travel through the world of key switching to reach such an understanding.

2.1 RLWE Encryption

In the beginning, there simply is a vector of numbers (our message) that we want to encrypt: integers modulo p for BFV/BGV and approximate numbers for CKKS. Using different paths for the different schemes, our message ends up in a plaintext polynomial $m \in \mathcal{R}_q$; for each coefficient in the most significant bits for BFV and in the least significant bits for BGV and CKKS. But, while plaintext encoding is an interesting journey in itself [10, 20], our journey simply starts with the ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^N + 1)$ and a plaintext polynomial $m \in \mathcal{R}_q$. Our encryption toolbox contains a secret key distribution χ_s, an error distribution χ_e, the error scaling factor t (different for each scheme[2]), and the uniform random distribution over $\mathcal{R}_q$. We sample a secret

2 For BFV, we set $t = 1$ and keep the error in the least significant bits. For BGV, we set $t = p$ and move the error above the message bits. For CKKS, we set $t = 1$ and

key $s \leftarrow \chi_s$, a small error $e \leftarrow \chi_e$, and a random polynomial $a \leftarrow \mathcal{R}_q$ from our toolbox and encrypt m:

$$(c_0, c_1) = (a \cdot s + te + m, -a) \in \mathcal{R}_q^2.$$

To decrypt, we evaluate the ciphertext $c = (c_0, c_1)$ as polynomial in s:

$$c(s) = c_0 + c_1 \cdot s = m + te \in \mathcal{R}_q.$$

The relevant related works [5, 14, 6, 25, 10, 9, 24] analyze correctness and security.

A BGV-like public key is an encryption of nothing:

$$\mathsf{pk} = (\mathsf{pk}_0, \mathsf{pk}_1) = (a \cdot s + te, -a) \in \mathcal{R}_q^2.$$

For public key encryption, we sample a temporary secret $u \leftarrow \chi_s$, two small errors $e_0, e_1 \leftarrow \chi_e$, and a random polynomial $a \leftarrow \mathcal{R}_q$ from our toolbox and encrypt as

$$(c_0, c_1) = (\mathsf{pk}_0 \cdot u + te_0 + m, \mathsf{pk}_1 \cdot u + te_1) \in \mathcal{R}_q^2.$$

As with a secret key encryption, $c(s) = m + te_{\mathsf{pk}} \in \mathcal{R}_q$ with a slightly larger error e_{pk}.

A BGV-like key switching key is an encryption of a secret s':

$$\mathsf{ksk} = (\mathsf{ksk}_0, \mathsf{ksk}_1) = (a \cdot s + te + s', -a) \in \mathcal{R}_q^2.$$

Key switching aims to output $(\tilde{c}_0, \tilde{c}_1)$ such that

$$\tilde{c}_0 + \tilde{c}_1 \cdot s + t\tilde{e} = c_i \cdot s'$$

adding a small key switching error $\tilde{e}$. To transform $c_i s^2$ after a multiplication, we set $s' = s^2$; to transform $c_i \pi(s)$ after a rotation π, we set $s' = \pi(s)$. In the following, we will continue our journey with the key switching key in our pocket and will discover how to switch keys from s' to s.

2.2 Key Switching

The idea of key switching is rather straightforward:

$$(\tilde{c}_0, \tilde{c}_1) = (c_i \cdot \mathsf{ksk}_0, c_i \cdot \mathsf{ksk}_1).$$

Then,

$$\begin{aligned} \tilde{c}(s) &= c_i \cdot \mathsf{ksk}_0 + c_i \cdot \mathsf{ksk}_1 \cdot s \\ &= c_i \cdot (a \cdot s + te + s') - c_i \cdot a \cdot s \end{aligned}$$

consider the error as part of the approximation.

$$= c_i \cdot s' + tc_i \cdot e$$

But, it does not quite work: While e is small, the added error $\tilde{e} = c_i \cdot e$ is not because c_i behaves like a random element in $\mathcal{R}_q$. Current state-of-the art employs two strategies to control the error: modulus extension and decomposition [25]. On our two detours, we will highlight the changes to naïve key switching.

For modulus extension, we temporarily compute in $\mathcal{R}_{qP}$ with the extension modulus P. We modify our naïve key switching key to

$$(\mathsf{ksk}_0, \mathsf{ksk}_1) = (a \cdot s + te + Ps', -a) \in \mathcal{R}_{qP}$$

where we sample $a \leftarrow \mathcal{R}_{qP}$. We compute key switching as before, but over $\mathcal{R}_{qP}$, thus:

$$c_i \cdot \mathsf{ksk}_0 + c_i \cdot \mathsf{ksk}_1 \cdot s = c_i \cdot Ps' + tc_i \cdot e \in \mathcal{R}_{qP}.$$

Finally, we switch back to $\mathcal{R}_q$ by scaling by $1/P$ resulting in

$$c_i \cdot s' + t\frac{c_i \cdot e}{P} \in \mathcal{R}_q.$$

As you might have noticed, keeping track of the different rings can get confusing. Implicitely, we will only use $\mathcal{R}_q$ and denote other ring moduli explicitely using the notation $[\cdot]_{qP}$. Hence, we switch keys with modulus extension computing

$$(\tilde{c}_0, \tilde{c}_1) = \left(\frac{[c_i \cdot \mathsf{ksk}_0]_{qP}}{P}, \frac{[c_i \cdot \mathsf{ksk}_1]_{qP}}{P}\right).$$

For $P \approx q$, the error $c_i \cdot e/P$ is negligibly small and key switching is correct [25].

For decomposition, we decompose c_i with respect to some base; for example using a power-of-two β with

$$c_i = \sum_{l=0}^{\omega-1} \beta^l c_i^{(l)};$$

the base is $(1, \beta, \beta^2, \ldots)$ and the decomposition $\mathcal{D}(c_i) = (c_i^{(0)}, c_i^{(1)}, c_i^{(2)}, \ldots)$. Next, we modify our naïve key switching key

$$(\mathsf{ksk}_0, \mathsf{ksk}_1) = (a_l \cdot s + te_l + \mathcal{B}(s')_l, -a_l)_{l=0}^{\omega-1} \in \left(\mathcal{R}_q^2\right)^\omega$$

with $a_l \leftarrow \mathcal{R}_q$, $e_l \leftarrow \chi_e$, and $\mathcal{B}(s') = (s', \beta s', \beta^2 s', \ldots)$. We also modify key switching itself:

$$(\tilde{c}_0, \tilde{c}_1) = \left(\langle \mathcal{D}(c_i), \mathsf{ksk}_0 \rangle, \langle \mathcal{D}(c_i), \mathsf{ksk}_1 \rangle\right).$$

Then,

$$\tilde{c}(s) = \langle \mathcal{D}(c_i), \mathsf{ksk}_0 \rangle + \langle \mathcal{D}(c_i), \mathsf{ksk}_1 \rangle \cdot s$$

$$= \langle \mathcal{D}(c_i), \mathcal{B}(s') \rangle + t \langle \mathcal{D}(c_i), (e_l)_{l=0}^{\omega-1} \rangle$$
$$= c_i \cdot s' + t \sum_{l=0}^{\omega-1} c_i^{(l)} \cdot e_l;$$

here, $c_i^{(l)}$ behaves like a random element in $\mathcal{R}_\beta$ instead of $\mathcal{R}_q$ and e_l is small. For small β, key switching is correct [25].

Combining both strategies looks like this:

$$(\mathsf{ksk}_0, \mathsf{ksk}_1) = \left([a_l \cdot s + te_l + P\mathcal{B}(s')_l]_{qP}, [-a_l]_{qP}\right)_{l=0}^{\omega-1} \in (\mathcal{R}_{qP})^\omega$$

and

$$(\tilde{c}_0, \tilde{c}_1) = \left(\frac{\left[\langle \mathcal{D}(c_i), \mathsf{ksk}_0 \rangle\right]_{qP}}{P}, \frac{\left[\langle \mathcal{D}(c_i), \mathsf{ksk}_1 \rangle\right]_{qP}}{P} \right).$$

Now, the key switching error is negligible as long as P is roughly as large as the largest element in $\mathcal{D}(c_i)$ [25]. Interestingly, for the decomposition, we only need $\mathcal{D}(c_i)$ to be small(ish) and $\langle \mathcal{D}(c_i), \mathcal{B}(s') \rangle = c_i \cdot s'$. And luckily, implementations have another decomposition naturally available that we can use!

2.3 DCRT Representation

In the land of BGV-like implementations, we commonly spot two decompositions via the Chinese Remainder Theorem (CRT): the residue number system (RNS) and the number theoretic transform (NTT). The former enables native integer arithmetic modulo large q, and the latter speeds up polynomial multiplication. Surprisingly, combining the two CRT representations is known as Double CRT (DCRT) representation.

Let $q = \prod_{i=1}^{\ell} q_i$ with co-prime primes q_i. The RNS representation of a polynomial $a \in \mathcal{R}_q$ is $a_i = [a]_{q_i} \in \mathcal{R}_{q_i}$. We can perform additions and multiplications over $\mathcal{R}_q$ in each ring $\mathcal{R}_{q_i}$ individually. Division and modular reduction, however, do not generally map to the RNS space. A notable exception is a scalar $\gamma \in \mathbb{Z}_q$ which divides each coefficient in a, then $a/\gamma = \gamma^{-1} a \in \mathcal{R}_q$. We can reconstruct $a \in \mathcal{R}_q$ from the individual a_i using the CRT:

$$a = \left(\sum_{i=0}^{\ell} \left[a_i \frac{q_i}{q} \right]_{q_i} \frac{q}{q_i} \right) \in \mathcal{R}_q.$$

The forward and inverse NTT are variants of the Fast Fourier Transform over a finite field and run in time $\mathcal{O}(N \log N)$. The forward NTT transforms a polynomial to the NTT domain, the inverse NTT transforms it back to the coefficient domain. Multiplication in $\mathcal{R}_q$ in the NTT domain corresponds to coefficient-wise multiplication, hence running in time $\mathcal{O}(N)$. Because the NTT is linear, we can move addition and scalar multiplication around independent of a polynomial's domain (within the laws of mathematics, of course). In combination with the RNS, we perform the forward and inverse

NTT for each ring $\mathcal{R}_{q_i}$ individually. The CRT view on the NTT is as polynomial factorization over $\mathcal{R}_{q_i}$[3]. Since the DCRT representation has a significant impact on key switching, we have to go back and visit it again.

2.4 Key Switching, Again

Recall our key switching key

$$(\mathsf{ksk}_0, \mathsf{ksk}_1) = \left([a_l \cdot s + te_l + P\mathcal{B}(s')_l]_{qP}, [-a_l]_{qP}\right)_{l=0}^{\omega-1} \in \left(\mathcal{R}_{qP}\right)^{\omega}$$

and key switching itself:

$$(\tilde{c}_0, \tilde{c}_1) = \left(\frac{\left[\langle \mathcal{D}(c_i),\, \mathsf{ksk}_0\rangle\right]_{qP}}{P}, \frac{\left[\langle \mathcal{D}(c_i),\, \mathsf{ksk}_1\rangle\right]_{qP}}{P}\right).$$

For the key switching key, we use the RNS with

$$q = \prod_{i=1}^{\ell} q_i \quad \text{and} \quad P = \prod_{j=1}^{k} P_j$$

computing over each $\mathcal{R}_{q_i}$ and $\mathcal{R}_{P_j}$, respectively, and store the result in the NTT domain (we can also sample a_k in the NTT domain which is nice). However, instead of decomposing s' to a power-of-two basis β, we recycle our RNS decomposition over q: We split the ℓ primes q_i into ω groups $\{\tilde{q}_1, \ldots, \tilde{q}_\omega\}$ with up to $\lceil \ell/\omega \rceil$ primes per group. To keep it simple, we will assume $\omega \mid \ell$ for the remainder of this work; hence, each group has ℓ/ω primes. Conceptually, we decompose as

$$\mathcal{D}(c_i) = \left(\left[c_i \frac{\tilde{q}_1}{q}\right]_{\tilde{q}_1}, \ldots, \left[c_i \frac{\tilde{q}_\omega}{q}\right]_{\tilde{q}_\omega}\right) \text{ with } \tilde{q}_l = \prod_{i=1}^{\ell/\omega} q_{(l-1)\ell/\omega+i}.$$

We reconstruct with the CRT, hence

$$\mathcal{B}(s') = \left(\left[s' \frac{q}{\tilde{q}_1}\right]_q, \ldots, \left[s' \frac{q}{\tilde{q}_\omega}\right]_q\right)$$

such that $\langle \mathcal{D}(c_i), \mathcal{B}(s')\rangle = c_i \cdot s'$. The key switching error is negligibly small with $P \approx \tilde{q}_l$ (or simply $k = \ell/\omega$) and key switching is correct, again [25]. Funnily enough, we can move the factors $[\tilde{q}_l/q]_{\tilde{q}_l}$ from $\mathcal{D}$ to the key switching key. Thus, $\mathcal{D}$ conceptually transforms the RNS over $\mathcal{R}_q$ to the RNS of each $\mathcal{R}_{\tilde{q}_l}$, respectively, but simply reuses the values in $\mathcal{R}_{q_i}$ [19].

For $\mathcal{D}$, going from $\mathcal{R}_q$ to $\mathcal{R}_{\tilde{q}_l}$ is easy because $\tilde{q}_l \mid q$. However, converting from one RNS basis to another is not always that easy: For $[\langle \mathcal{D}(c_i), \mathsf{ksk}_0\rangle]_{qP}$,

3 For $q_i = 1 \bmod 2N$, the polynomial $X^N + 1$ splits into N factors $X - \xi^j$ where ξ^j are the $2N$-th roots of unity. The NTT is the CRT with respect to these N factors.

we need to convert each element in $\mathcal{D}(c_i)$ from $\mathcal{R}_{\tilde{q}_l}$ to $\mathcal{R}_{qP}$, but $qP \nmid \tilde{q}_l$. In general, for two RNS bases

$$E = \prod_{i=1}^{n} E_i \quad \text{and} \quad E' = \prod_{j=1}^{n'} E_j$$

and $a \in \mathcal{R}_E$, the fast base extension

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

outputs $a_j = [a + \varepsilon E]_{E'_j}$ for a small ε using only native integer arithmetic modulo E_i and E'_j. Often enough, we can consider εE as part of the homomorphic error. If needed, we remove it using an error correction technique such as BEHZ [4] or HPS [19]. For a fast base extension, we need the input in the coefficient domain because we interact between different RNS primes. For key switching, we base extend each element in the decomposition:

$$c_{\mathsf{ext}} = (c_{\mathsf{ext},l})_{l=1}^{\omega} \text{ with } c_{\mathsf{ext},l} = \mathsf{BaseExt}(\mathcal{D}(c_i)_l, \tilde{q}_l, qP).$$

Here, we can consider $\varepsilon \tilde{q}_l$ as part of the negligibly small key switching error [25]. But, this scaling is a division (sadly not mapping to the RNS space) which is the last hurdle to clear during our visit to DCRT key switching.

Let c'_i be a polynomial in $\mathcal{R}_{qP}$. Our goal is to compute $\tilde{c}_i = [c'_i/P]_q$. While we could simply multiply by $P^{-1} \in \mathbb{Z}_q$ if P would divide every coefficient in c'_i, this is not true in general. But, we can make it true: We just add

$$\delta_i = -t[t^{-1}c'_i]_P \text{ with } [c'_i + \delta_i]_P = 0.$$

Because $t \mid \delta_i$ over $\mathcal{R}_q$, it only affects the error. It will also be small because it behaves like a random element in $\mathcal{R}_P$ (roughly P-sized), which we then divide by $1/P$ afterward [25]. Now, P divides every coefficient in $c'_i + \delta_i$ and

$$\tilde{c}_i = \frac{c'_i + \delta_i}{P} = P^{-1}(c'_i + \delta_i).$$

We now conclude our key switching journey with a final decomposition.

2.5 Decomposing, Again

Until now, we only used one decomposition $\mathcal{D}$. Recently, Kim et al. [26] suggested a new algorithmic approach using a second decomposition on top. We therefore use the terms single- and double-decomposition technique to differentiate between the two (if you guessed that we use the former for key switching with one decomposition and the latter for key switching with two, you are correct!). Their idea is rather straightforward and only uses concepts we already went past.

Recall again single-decomposition key switching:

$$(\tilde{c}_0, \tilde{c}_1) = \left(\frac{\left[\langle c_{\text{ext}},\, \mathsf{ksk}_0 \rangle\right]_{qP}}{P}, \frac{\left[\langle c_{\text{ext}},\, \mathsf{ksk}_1 \rangle\right]_{qP}}{P} \right).$$

The double-decomposition technique performs the dot product over $\mathcal{R}_{qP}$ in another ring $\mathcal{R}_E$ for a new RNS base E. Initially, we split the $\ell + k$ primes in qP into $\tilde{\omega}$ groups (we will assume $\tilde{\omega} \mid \ell + k$). We decompose ksk toward these $\tilde{\omega}$ groups, afterward extending from each group to $\mathcal{R}_E$ (here, we need to correct the error when using the fast base extension). During key switching, we extend each element in $\mathcal{D}(c_i)$ from $\mathcal{R}_{\tilde{q}_l}$ to $\mathcal{R}_E$ (error correction required) and compute the dot product in $\mathcal{R}_E$[4]. Then, we extend the result to $\mathcal{R}_{qP}$ (no error correction required) and finish up as before by scaling by $1/P$. In the following, we will provide algorithmic descriptions of both techniques after summarizing our notation collected along the way. Thus, our journey through the current state-of-the-art on key switching and its related concepts comes to an end.

2.6 Summary

Common notation:

$\mathcal{R}_m$, $[\cdot]_m$	$\mathcal{R}_m = \mathbb{Z}_m[X]/(X^N + 1)$ for a power-of-two degree N where $[\cdot]_m$ denotes arithmetic in $\mathcal{R}_m$ (arithmetic in $\mathcal{R}_q$ is mostly implicit)
q, ℓ, b	$q = \prod_{i=1}^{\ell} q_i$ as ciphertext modulus with ℓ co-prime primes q_i with b bits each
P, k, β	$P = \prod_{j=1}^{k} P_j$ as extension modulus with $k = \ell/\omega$ co-prime primes with β bits each; also co-prime to q
E, r, $\tilde{\beta}$	$E = \prod_{i=1}^{r}$ as double-decomposition modulus with r co-prime primes E_i with $\tilde{\beta}$ bits each; also co-prime to q and P
$\mathcal{D}$, $\mathcal{B}$, ω	RNS decomposition $\mathcal{D}(\cdot)$ over $\mathcal{R}_q$ into ω groups $\tilde{q}_l$ such that $\langle \mathcal{D}(c_i), \mathcal{B}(s') \rangle = c_i \cdot s'$
$\tilde{\mathcal{D}}$, $\tilde{\mathcal{B}}$, $\tilde{\omega}$	RNS decomposition $\tilde{\mathcal{D}}(\cdot)$ over $\mathcal{R}_{qP}$ into $\tilde{\omega}$ groups $\tilde{Q}_l$ such that $[\langle \tilde{\mathcal{D}}(c_i), \tilde{\mathcal{B}}(s') \rangle]_{qP} = [c_i \cdot s']_{qP}$
B	upper bound on b, β, and $\tilde{\beta}$
t	the error scaling factor

4 E needs to be able to store the dot product without "overflow modulo E".

Common algorithms:

$\mathsf{NTT}_{\mathsf{fwd}}, \mathsf{NTT}_{\mathsf{inv}}$	the forward and inverse NTT, respectively
$\mathsf{BaseExt}$	the fast base extension of $a \in \mathcal{R}_E$ to $\mathcal{R}_{E'}$

$$\mathsf{BaseExt}(a, E, E') = \left[\sum_i \left[a_i \frac{E_i}{E} \right]_{E_i} \frac{E}{E_i} \right]_{E'_j}$$

Single-decomposition key switching[5]:

key generation	$\mathsf{ksk} = \left([a_l \cdot s + te_l + P\mathcal{B}(s')_l]_{qP}, [-a_l]_{qP} \right)_{l=0}^{\omega-1}$
input extension	$c_{\mathsf{ext}} = \left(\mathsf{BaseExt}(\mathcal{D}(c_i)_l, \tilde{q}_l, qP) \right)_{l=0}^{\omega-1}$
dot product	$c'_i = \left[\langle \mathsf{NTT}_{\mathsf{fwd}}(c_{\mathsf{ext}}), \mathsf{ksk}_i \rangle \right]_{qP}$
scaling	$\delta_i = -t\, \mathsf{BaseExt}(\mathsf{NTT}_{\mathsf{inv}}([t^{-1} c'_i]_P), P, q)$
	$\tilde{c}_i = P^{-1}(\mathsf{NTT}_{\mathsf{inv}}(c'_i) + \delta_i)$

Double-decomposition key switching[6]:

key generation	$\widetilde{\mathsf{ksk}} = \left(\mathsf{BaseExt}(\tilde{\mathcal{D}}(\mathsf{ksk})_l, \tilde{Q}_l, E) \right)_{l=0}^{\tilde{\omega}-1}$
input extension	$c_{\mathsf{ext}} = \left(\mathsf{BaseExt}(\mathcal{D}(c_i)_l, \tilde{q}_l, E) \right)_{l=0}^{\omega-1}$
dot product	$c''_i = \left[\langle \langle \tilde{\mathcal{B}}(\mathsf{NTT}_{\mathsf{fwd}}(c_{\mathsf{ext}})), \widetilde{\mathsf{ksk}}_i \rangle \rangle \right]_E$
	$c'_i = \mathsf{BaseExt}\left([\mathsf{NTT}_{\mathsf{inv}}(c''_i)]_E, E, qP \right)$
scaling	$\delta_i = -t\, \mathsf{BaseExt}([t^{-1} c'_i]_P, P, q)$
	$\tilde{c}_i = P^{-1}(\mathsf{NTT}_{\mathsf{inv}}(c'_i) + \delta_i)$

5 We assume that the key switching key ksk is in the NTT domain and do not include the calls to $\mathsf{NTT}_{\mathsf{fwd}}$ and $\mathsf{NTT}_{\mathsf{inv}}$ for key generation. We assume an input c_i in the coefficient domain and output $\tilde{c}_i$ in the coefficient domain. We make the same assumptions for the double-decomposition technique.

6 With slight abuse of notation, $\langle\langle \cdot \rangle\rangle$ performs a dot product instead of a multiplication for each pair in the outer dot product.

Table 1: Current state-of-the-art on key switching complexity.

	Scheme	Decomposition	
		single	double
$\mathcal{O}$	*	$\mathcal{O}(\ell^2)$	$\mathcal{O}(r^2)$
`ntt`	BFV BGV/CKKS	$(\omega+2)(\ell+k)$	$(\omega+2\tilde{\omega})r$
`mul`	BFV BGV/CKKS	$\ell(\ell+2\omega+2k+5)+2k$ $\ell(\ell+2\omega+2k+7)+4k$	$r(3\ell+2\omega\tilde{\omega}+2k)$
`ksk`	*	$2\omega N(\ell+k)$	$2\omega\tilde{\omega}Nr$

Key Switching in Theory
Section 3

Our current understanding of key switching complexity mostly stems from two works. In their extended version, Kim, Polyakov, and Zucca [25] analyse the single-decomposition technique and count the number of forward and inverse NTTs `ntt` (each time $\mathcal{O}(N \log N)$) as well as the number of multiplications `mul`, either coefficient-wise with another polynomial or with a scalar (each time $\mathcal{O}(N)$). They also determine the number of primes `ksk` in a key switching key. In general, they differentiate between BFV and BGV for two reasons:

- The default domain for a ciphertext is different for BFV, BGV, and CKKS. For BFV, a ciphertext is usually in the coefficient domain. For BGV and CKKS, a ciphertext is usually in the NTT domain. Across domains, `ntt` is still the same due to a neat trick that Kim, Polyakov, and Zucca use [25].
- For BFV and CKKS, $t = 1$ reduces the number of scalar multiplications.

For the double-decomposition technique, Kim et al. [26] also provide `ntt`, `mul`, and `ksk`. Additionally, they provide asymptotic complexities for both techniques. We summarize the current state-of-the-art in Table 1. But, there are several problems:

1. For the single-decomposition technique, the perspective on $\mathcal{O}$ is not optimal.
2. There is no estimate for the number of primes r in E that only depends on ℓ, ω, and $\tilde{\omega}$; this complicates comparing both techniques.
3. We can reduce the number of multiplications `mul` in both tech-

niques. Also, Kim et al. [26] exclude the scaling step from `mul`.

4 Kim et al. [26] always assume input and output in the coefficient domain which is only sensible for BFV.

In the following, we will tackle and resolve each issue. Along the way, we will collect questions to evaluate in Section 4.

3.1 A New Perspective

In BGV-like schemes, security depends on the distributions χ_s and χ_e, the degree N, and the modulus qP. For the distributions, common choices are are a uniform ternary distribution for χ_s and a centered Gaussian distribution with variance $\sigma = 3.19$ for χ_e [1]. The parameters N, q, and P differ for each use case. Increasing N increases security, but decreases performance. Increasing q (and hence qP) decreases security, but it increases the space where the error can grow and thus permits more homomorphic operations before bootstrapping becomes necessary. In fact, we mostly try to avoid bootstrapping in BGV-like schemes because it is so expensive, and we instead often choose a large enough q for a use case [18]. Then, we fix a power-of-two N as small as possible for performance, but as large as needed for security[7] [28]. Finally, we choose P and ω such that key switching is correct and secure.

What are the consequences for the key switching parameters P and ω? Essentially, we need to answer the following question: Given N and q, how do we set these parameters for best performance? Kim et al. [26] argue as follows for the single-decomposition key technique (recall that $k = \ell/\omega$): By choosing $k \in \mathcal{O}(1)$, we require $\omega \in \mathcal{O}(\ell)$ and

$$(\omega + 2)(\ell + k) \in \mathcal{O}\big(\ell^2\big)$$

follows accordingly for `ntt`. But, this implicitly limits k. We take a new perspective: We consider $\omega \leq \ell$ as parameter in the security level which we can choose as we desire. Then,

$$(\omega + 2)(\ell + k) = \omega\ell + 3\ell + \frac{2\ell}{\omega} \quad \Rightarrow \quad \texttt{ntt} \in \mathcal{O}(\omega\ell).$$

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: The smaller ω, the better our performance. We also collect our first question to evaluate: If possible, is it better to use $\omega = 1$ or $\omega = 2$?

7 This almost works: N impacts error growth and hence q, but not significantly.

While the above fact is simple, reality is not, of course. Decreasing ω increases $k = \ell/\omega$ (and hence P (and hence qP)), thus decreasing security for fixed N. Luckily, there are two solutions: For fixed N, we choose a secure lower bound $\mathcal{W}$ for ω, then set $k \in \mathcal{O}(\ell/\mathcal{W})$ large. Or, and this will sound crazy if you ever implemented and benchmarked a BGV-like scheme, we consider increasing the degree N. For the degree $2N$, we then use $\omega' = 1$ or $\omega' = 2$; we choose $\omega' = 2$. In terms of memory, that is `ksk`, it is worth it to increase the degree once

$$2\omega N(\ell + k) = 2N(\omega\ell + \ell) > 12N\ell = 4N(\omega'\ell + \ell) \quad \Rightarrow \quad \omega > 5.$$

Considering `ntt`, the main computational bottleneck, it gets more complex; we cannot simply use the asymptotic complexity $N \log N$ due to the hidden factors. To keep it simple, we still do: With $2N\log(N+1) \approx 2N \log N$, we get

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

We need to solve $\omega^2 - 9\omega + 2 > 0$ and $\omega > 8.77$ follows. At some point (and most likely not $\omega > 8.77$), increasing the degree should become worth it not only in terms of memory, but also in terms of running time. Does it? A second question to evaluate.

3.2 Estimating r

Kim et al. [26] provide a lower bound for $\log_2 E$ based on the infinity norm. An element $a \in \mathcal{R}_m$ has coefficients in $[-m/2, m/2)$, thus $\|a\|_\infty \leq m/2$. For a product $[a \cdot b]_m$, we have

$$\|ab\|_\infty \leq \delta_{\mathcal{R}} \|a\|_\infty \|b\|_\infty$$

for the ring expansion factor $\delta_{\mathcal{R}} = N$ [18]. E needs to be large enough to store a dot product of ω elements in $\mathcal{R}_{\tilde{q}_l}$ and $\mathcal{R}_{\tilde{Q}_l}$. Thus, the bound is

$$\log_2 E \geq \log_2 \left(\frac{\omega N}{4} \max \tilde{q}_l \max \tilde{Q}_l\right).$$

By definition, we have $\max_l \tilde{q}_l = \ell b/\omega$ and $\max_l \tilde{Q}_l = (\ell b + k\beta)/\tilde{\omega}$. Assuming $b \approx \beta \approx \tilde{\beta} \approx B$, we get

$$r \geq \frac{\log_2(\omega N/4)}{B} \left(\frac{\ell}{\omega} + \frac{\ell + k}{\tilde{\omega}}\right).$$

In practice, $\log_2(\omega N/4)/B \in \mathcal{O}(1)$ is negligibly small. With $k = \ell/\omega$, a good estimate is

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

For `ntt`, we get

$$(\omega + 2\tilde{\omega})r = \left(\frac{\omega^2 + 3\omega\tilde{\omega} + 2\tilde{\omega}^2 + \omega + 2\tilde{\omega}}{\omega\tilde{\omega}}\right)\ell \quad \Rightarrow \quad \texttt{ntt} \in \mathcal{O}(\omega\ell/\tilde{\omega} + \tilde{\omega}\ell/\omega).$$

As before, we want to minimize ntt, the main computational bottleneck. But, over $\mathbb{R}$, we get negative solutions for ω and $\tilde{\omega}$ (if you can figure out how to negatively decompose, please let us know). Instead, we exhaustively search for optimal solutions for $\ell \leq 200$, a generous bound for the number of primes in q. For $\ell \leq 200$, we minimize ntt with

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

But choosing ω and $\tilde{\omega}$ as above is a double-edged sword: It blows up the number of primes in the key switching key

$$2\omega\tilde{\omega}Nr = 2(\omega\ell + \tilde{\omega}\ell + \ell)N.$$

In contrast to the single-decomposition technique, we have to use much more memory to get the best computational complexity. We collect another question: For optimal parameters, which technique performs better?

3.3 Multiplication Folding

The number of multiplications mul consists of two types: coefficient-wise multiplication of two polynomials and scalar multiplication with each coefficient of one polynomial. We need the former only for the dot product with the key switching key. The latter is either a multiplication with a known scheme constant (such as t or P^{-1}) or takes place during the fast base extension

$$\mathsf{BaseExt}(a, E, E') = \left[\sum_i \left[a_i \frac{E_i}{E}\right]_{E_i} \frac{E}{E_i}\right]_{E'_j},$$

one in the source ring $\mathcal{R}_E$ and one in the destination ring $\mathcal{R}_{E'}$ (we obviously precompute $[E_i/E]_{E_i}$ and $[E/E_i]_{E'_j}$). Hence, a multiplication belongs in one of four groups:

1. polynomial multiplication with the key switching key;
2. scalar multiplication with a known scheme constant;
3. scalar multiplication during BaseExt in the source ring; and
4. scalar multiplication during BaseExt in the destination ring.

Given the right circumstances, we can merge multiplications. And actually, we already encountered an example during our journey in Section 2: Moving the factors $[\tilde{q}_l/q]_{\tilde{q}_l}$ from $\mathcal{D}$ (group 3) to the key switching key (group 1). Here, the invidivudal q_i for the source rings $\mathcal{R}_{\tilde{q}_l}$ happen to be all part of the destination ring $\mathcal{R}_{qP}$ and, while conceptually different, they both boil down to computations over all $\mathcal{R}_{q_i}$. In general, the following applies:

- We cannot get around the multiplication with the key switching key (group 1) (and to be fair, anything else would be really strange).

- We can move a scalar to the key switching key (group 1) as long as it is needed in every key switching which uses this key switching key; especially known scheme constants (group 2).
- We can move multiplications to and from BaseExt (group 3 and 4) if they are moved for all used RNS primes.

It also does not matter that we have to transform the result of the fast base extension from the coefficient to the NTT domain due to the linearity of the NTT and we now temporarily remove calls to $\mathsf{NTT}_{\mathsf{fwd}}$ and $\mathsf{NTT}_{\mathsf{inv}}$. Recall single-decomposition key switching starting at the dot product over $\mathcal{R}_{qP}$, and we inline the fast base extension BaseExt:

$$\begin{array}{cc}
\mathcal{R}_q & \mathcal{R}_P \\
c'_i = \langle c_{\mathsf{ext}}, \mathsf{ksk}_i \rangle & c'_i = \langle c_{\mathsf{ext}}, \mathsf{ksk}_i \rangle \\
 & \delta_{i,j} = t^{-1} c'_i P_j / P \\
\delta_i = -t \sum_j \delta_{i,j} P / P_j & \\
\tilde{c}_i = P^{-1}(c'_i + \delta_i). &
\end{array}$$

Our crucial observation is that we use the result of the dot product c'_i disjointed over $\mathcal{R}_q$ and $\mathcal{R}_P$: We use $[c']_q$ only to compute $[\tilde{c}_i]_q$ and we use $[c'_i]_P$ only to compute $[\delta_{i,j}]_P$. Thus, we can move scalars around differently for $\mathcal{R}_q$ and $\mathcal{R}_P$, respectively:

$$\begin{array}{cc}
\mathcal{R}_q & \mathcal{R}_P \\
c'_i = \left\langle c_{\mathsf{ext}}, P^{-1}\,\mathsf{ksk}_i \right\rangle & \delta_{i,j} = \left\langle c_{\mathsf{ext}}, t^{-1} P_j / P\,\mathsf{ksk}_i \right\rangle \\
\delta_i = -t \sum_j \delta_{i,j} / P_j & \\
\tilde{c}_i = c'_i + \delta_i. &
\end{array}$$

Overall, our insights reduce `mul` down to

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

for any t, that is across all schemes. For a given ℓ, we minimize $2\omega + 2\ell/\omega$, and hence `mul`, with $\omega = \sqrt{\ell}$. Our next question: How much time do we gain?

We also apply our folding techniques to the double-decomposition technique:

$$\begin{array}{ccc}
\mathcal{R}_q & \mathcal{R}_P & \mathcal{R}_E \\
 & & c''_{i,\iota} = \langle c_{\mathsf{ext}}, E_\iota / E\,\mathsf{ksk}_i \rangle \\
c'_i = P^{-1} \sum_\iota c''_{i,\iota} E / E_\iota & \delta_{i,j} = t^{-1} P_j / P \sum_\iota c''_{i,\iota} E / E_\iota & \\
\delta_i = -t \sum_j \delta_{i,j} / P_j & & \\
\tilde{c}_i = c'_i + \delta_i. & &
\end{array}$$

Including scaling, we reduce the number of multiplications down to

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

The local minima of ω and $\tilde{\omega}$ for $\ell \leq 200$ are close to $\sqrt{\ell}$.

3.4 Input and Ouput Domains

Our new foldings have another benefit as part of a bigger picture because we moved the scaling by P^{-1} away from the last step:

$$(\tilde{c}_0, \tilde{c}_1) = (c_0' + \delta_0, c_1' + \delta_1).$$

Consider a multiplication where we switch the key of c_2 and output

$$(c_0 + \tilde{c}_0, c_1 + \tilde{c}_1) = (c_0 + c_0' + \delta_0, c_1 + c_1' + \delta_1).$$

In the single-decomposition technique, c_i' is in the NTT domain and δ_i is in the coefficient domain. For BFV, we transform c_i' to the coefficient domain and, for BGV and CKKS, transform δ_i to the NTT domain to match the respective input domain. But sometimes, it can be useful to ignore the default domain. For c_i in the coefficient domain, we can choose the output domain:

$$\mathsf{NTT}_{\mathsf{inv}}(c_i') + c_i + \delta_i \quad \text{or} \quad c_i' + \mathsf{NTT}_{\mathsf{fwd}}(c_i + \delta_i).$$

For c_i in the NTT domain, the same idea works:

$$\mathsf{NTT}_{\mathsf{inv}}(c_i' + c_i) + \delta_i \quad \text{or} \quad c_i' + c_i + \mathsf{NTT}_{\mathsf{fwd}}(\delta_i).$$

Trivially, we can also choose output domains for a rotation where we switch c_1 and output $(c_0 + \tilde{c}_0, \tilde{c}_1)$. Fun fact: We could even choose different output domains for each prime q_i individually.

For the double-decomposition technique, we also moved scaling by P^{-1}:

$$(\tilde{c}_0, \tilde{c}_1) = (c_0' + \delta_0, c_1' + \delta_1);$$

but now, c_i' and δ_i are both in the coefficient domain. For output in the NTT domain, we need 2ℓ additional NTTs:

$$\left(\mathsf{NTT}_{\mathsf{fwd}}(c_0' + \delta_0), \mathsf{NTT}_{\mathsf{fwd}}(c_1' + \delta_1)\right),$$

possibly adding c_0 and/or c_1 before performing the forward NTT. To make things worse, the trick that keeps `ntt` the same across input domains does not work for the double-decomposition technique. For BGV and CKKS, we overall need 3ℓ additional NTTs, 2ℓ forward and ℓ inverse.

3.5 Large Primes, Mostly

So far, we (and all cited literature) assumed $b \approx \beta \approx \tilde{\beta} \approx B$ for the primes in q, P, and E. But is this true? Generally, choosing primes as close to the upper bound B as possible is beneficial for two straightforward reasons:

- Each additional prime increases the number of polynomial operations during homomorphic evaluation. The larger each prime, the fewer primes we need, reducing compute and memory costs.
- Using small primes typically wastes compute and memory resources: operations are still performed over B-sized numbers.

Given a bound $\log_2 q$ (or $\log_2 P$ or $\log_2 E$), we ideally want to:

1. Set $\ell = \lceil \log_2 q/B \rceil$.
2. Choose b such that $\log_2 q \approx \ell \cdot b$.
3. Generate ℓ primes close to 2^b.

For P and E, this is actually exactly what we do! For BFV, this also works because our input to key switching is always in $\mathcal{R}_q$ so the size of each q_i does not really matter [25]. For BGV and CKKS, however, we only start in $\mathcal{R}_q$ and remove individual primes over time.

We already encountered the idea of removing primes in a specific version: the scaling by $1/P$ during key switching, also known as modulus switching. In BGV, we use modulus switching to reduce the absolute error after a multiplication. We scale by one of the primes q_i and remove it from the ciphertext modulus $q/q_i = q'$. Afterward, we continue in the ring $\mathcal{R}_{q'}$. Relative to q', the error has the same size as before. But the absolute size is scaled by $1/q_i$ and, for properly chosen q_i, we stop the error from growing exponentially in the following multiplications [6]. In CKKS, we solve a different problem with rescaling. During a multiplication, the approximate message moves from the least significant bits of the ciphertext modulus q into higher bits. We then scale by $1/q_i$ to move it back to the least significant bits, also removing q_i from the ciphertext modulus. Here, the size of the primes q_i is even more important than for BGV because scaling has to be pretty precise [9]. For both schemes, scaling by roughly 2^B may not be what we need.

We therefore need to show that we can actually assume $b \approx \beta \approx \tilde{\beta} \approx B$, at least mostly. Luckily, Gentry, Halevi, and Smart already describe an idea which solves our problem: We choose mostly large primes for q; additionally, we choose a few small primes which we constantly switch in and out of q to precisely control scaling [18]. Sadly, they are very sparse on the details. And obviously, their idea is not free and we have to figure out its additional costs. To do so, we will introduce some additional notation for modulus switching, also known as rescaling, up and down scaling, or scaling and rounding.

Switching the modulus.

We scale and round $a \in \mathcal{R}_q$ to the modulus q' with

$$a' = \left[\left\lceil \frac{q'}{q} a \right\rfloor_t \right]_{q'}$$

where $\lceil \cdot \rfloor_t$ rounds to the nearest integer coefficients which are $[0]_t$; this keeps the message intact for BFV and BGV. The RNS-friendly version is

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

with $[\delta]_t = 0$ and $q \mid (q'a + \delta)$ by definition. This scales either the error (BFV and BGV) or the approximate message (CKKS) by roughly q'/q. We will use modulus switching to remove specific primes q_i from q. To easily exclude primes from q, we denote $q_{x,y} = \prod_{i=x}^{y} q_i$ with $q = q_{1,\ell}$.

Switching primes in and out.

For $\ell = \ell' - \mu + \kappa$ and $\log_2 q \approx (\ell' + \mu)b$, we use $\ell' - \kappa$ primes close to 2^b as well as $\kappa > \mu$ smaller primes, each close to $2^{\mu b/\kappa}$. For $b \approx B$, we continuously scale by $2^{\mu b/\kappa}$ as follows:

1 Receive a fresh encryption from the client.
2 Perform the desired homomorphic operations until we need to scale.
3 Switch the modulus using one of the κ small primes.
4 Repeat steps (2) and (3) until all κ small primes are gone.
5 Perform homomorphic operations, during the last key switching before scaling, replace any μ large primes with the κ small primes.
6 Continue with step (3) using the small primes we switched back in.

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 switch the modulus of $a \in R_{q_{\kappa+1,\ell}}$ as

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

We then integrated with either key switching technique during the scaling step by switching in the small primes $\{q_1, \ldots, q_\kappa\}$ and switching out the large primes $\{q_{\ell-\mu}, \ldots, q_\ell, P_1, \ldots, P_k\}$:

$$\left[\left\lceil \frac{q_{1,\ell-\mu}}{q_{\kappa+1,\ell}P_{1,k}} a \right\rfloor_t \right]_{q_{1,\ell-\mu}} = \left[\frac{q_{1,\kappa}c_i' - t[t^{-1}q_{1,\kappa}c_i']_{q_{\ell-\mu,\ell}P_{1,k}}}{q_{\ell-\mu,\ell}P_{1,k}} \right]_{q_{1,\ell-\mu}} .$$

In the single-decomposition technique, the number of inverse NTT 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 $\ell - \mu$. In the double-decomposition technique, we are in the coefficient domain anyway after base extending from E. Hence, 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 for both techniques, κ per δ_i. As $\kappa \in \mathcal{O}(1)$ is usually very small (think $\kappa = 2$ or $\kappa = 3$), this is only a very small overhead. But how large is it in practice? Another question to evaluate.

Choosing the primes.

We modify the parameter selection process as follows:

1 Choose $b \approx B$, μ, and κ to scale by $q_{\text{scale}} = 2^{\mu b/\kappa}$.
2 Set $q = q_{\text{scale}}^L + q_{\text{dec}}$ for L available scalings and a decryption cushion q_{dec}. Also set $\ell' = \lceil \log_2 q/B \rceil$.
3 Generate $\ell' - \mu$ primes close to 2^b.
4 Generate κ primes close to q_{scale}.
5 Set $\ell = \ell' - \mu + \kappa$.

Compared to the more naïve approach, we reduce ℓ as long as

$$\left\lceil \frac{\log_2 q}{\mu b/\kappa} \right\rceil \approx \left\lceil \frac{\ell'}{\mu/\kappa} \right\rceil > \ell \;\Leftrightarrow\; \kappa\ell' - \mu\ell \geq \mu \;\Leftrightarrow\; \ell \geq \kappa + \frac{\mu}{\kappa - \mu}.$$

For example, with $B = 60$, we consider a use case scaling by $q_{\text{scale}} \approx 2^{36}$. Then, with $b = 54 \approx B$, $\mu = 2$, and $\kappa = 3$, we reduce the overall number of primes as soon as $\ell \geq 5$ (equivalent to $\log q > 144$). We collect a final question: How large are our improvements choosing (mostly) large primes?

3.6 Summary

We started with four issues to resolve:

1 For the single-decomposition technique, the perspective on $\mathcal{O}$ is not optimal.
2 There is no estimate for the number of primes r in E that only depends on ℓ, ω, and $\tilde{\omega}$; this complicates comparing both techniques.
3 We can reduce the number of multiplications `mul` in both techniques.
4 Kim et al. [26] always assume input and output in the coefficient domain, which usually only holds for BFV, and exclude the scaling step for `mul`.

In addition to solving these issues, we also showed that we can mostly as-

Table 2: Our update for key switching analysis.

	Scheme	Decomposition	
		single	double
$\mathcal{O}$	*	$\mathcal{O}(\omega\ell)$	$\mathcal{O}(\omega\ell/\tilde{\omega}+\tilde{\omega}\ell/\omega)$
`ntt`	BFV BGV/CKKS	$\omega\ell+3\ell+2\ell/\omega$	$\frac{\omega^2+3\omega\tilde{\omega}+2\tilde{\omega}^2+\omega+2\tilde{\omega}}{\omega\tilde{\omega}}\ell$ $\mathtt{ntt}_{\mathsf{BFV}}+3\ell$
`mul`	BFV BGV/CKKS	$\ell(\ell+2\omega+2\ell/\omega+3)$	$\ell(2\omega+2\tilde{\omega}+\frac{\omega\ell+3\tilde{\omega}\ell+\ell}{\omega\tilde{\omega}}+5)$
`ksk`	*	$2(\omega\ell+\ell)N$	$2(\omega\ell+\tilde{\omega}\ell+\ell)N$

sume $b \approx \beta \approx \tilde{\beta} \approx B$. We update the previous state-of-the-art from Table 1 with our new analysis in Table 2.

Key Switching in Practice
Section 4

In the previous section, we posed six questions to evaluate:

1. Is $\omega = 1$ or $\omega = 2$ better if we can choose (single-decomposition)?
2. Can increasing N actually be worth it (single-decomposition)?
3. Is the single- or the double-decomposition technique better?
4. How large is the speed-up from constant folding?
5. How costly is replacing large with small primes?
6. How large is the speed-up using mostly large primes?

We do so on an Ubuntu 20.04.5 with an Intel Core i9-7900X at 3.3 GHz and 64 GiB of available memory. We disable Intel turbo boost and pin program execution to a single core. Our implementation[8] uses the open-source library fhelib [13] which uses the state-of-the-art library HEXL [22] for fast polynomial arithmetic. We average execution times over all combinations of input and output domain. If not otherwise noted, we use the single-decomposition technique with our folding improvements, $B = 60$, $b \approx B$, $\beta \approx B$, and $\tilde{\beta} \approx B$. For $N \in \{2^{14}, 2^{15}, 2^{16}, 2^{17}\}$, we use the respective upper bounds $\log_2 qP \leq \{443, 867, 1735, 3470\}$ for at least 128 bit security [28].

4.1 Is $\omega = 1$ or $\omega = 2$ better if we can choose (single-decomposition)?

To be able to choose between $\omega = 1$ and $\omega = 2$, we need $q \approx P$ and thus $\log_2 q$ needs to use $\leq 50\,\%$ of the available space for $\log_2 qP$. For each N, we use

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

Table 3: Execution times for $\omega = 1$ and $\omega = 2$ where q uses 50 % of the available modulus space. We also report the speed-up in percent.

Parameters		Time (*ms*)		Speed-up
$\log_2 N$	$\log_2 q$	$\omega = 1$	$\omega = 2$	
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 4: Execution times for parameter sets where q uses 90 % or 95 %. Each parameter set has a sibling with degree $2N$ and $\omega' = 2$. We also report the speed-up of $2N$ compared to N in percent.

Parameters				Time (*ms*)		Speed-up
$\log_2 N$	$\log_2 q$	ω	ω'	N	$2N$	
15	780	10	2	98.6	80.7	22.2 %
16	1560	10	2	457.9	564.6	-18.9 %
16	1624	19	2	785.3	626.5	25.3 %
17	3120	9	2	2072.6	3172.6	-34.7 %
17	3300	19	2	3532.2	3462.4	2.0 %

exactly 50 % of the available modulus space for q. Using the results in Table 3, we recommend to use $\omega = 2$; it performs better most of the time. This is somewhat expected because $\omega = 2$ usually results in less multiplications (minimal for $\sqrt{\omega}$). The exception is $N = 2^{14}$ with $\log_2 q = 216$ where $\ell = \lceil \log_2 q/B \rceil = 4$ is rather small, and thus also the number of multiplications. Also, for $\omega = 2$, the key switching key is larger than for $\omega = 1$ and reading it from memory impacts running time more if `ntt` and `mul` are small.

4.2 Can increasing N actually be worth it (single-decomposition)?

Increasing the degree N only makes sense for rather large ω. We only get rather large ω if the ciphertext modulus q uses most of the available modulus space. For evaluation, we use two parameter set for each degree occupying 90 % and 95 % of the available space, respectively. We also create a sibling set with the degree $2N$ and $\omega' = 2$. We exclude $N = 2^{14}$, the available modulus space is way too small, and we exclude one set for $N = 2^{15}$ where ω is the same for 90 % and 95 %. Table 4 shows that, contrary to current folklore, increasing the degree can actually be worth it. But, doing so has also costs outside of key switching: memory-wise, a larger public encryption key and larger ciphertexts and compute-wise, more coefficients to compute on for all other homomorphic operations. We believe that increasing the degree is only worth it if

1 ω is large;
2 the main bottleneck of the use case is the key switching operation;
3 the use case requires many different rotations, and each rotation has its own unique key switching key; and
4 the key switching operation is memory-bound.

The last point is especially interesting in the context of hardware accelerators where the hardware for a larger degree might already exist and be otherwise unused. Also, hardware accelerators of BGV-like schemes tend to be memory-bound reading the key switching key and reducing the key size (already smaller for $\omega > 5$) could be more significant.

4.3 Is the single- or the double-decomposition technique better?

Kim et al. [26] already compare both techniques using a comprehensive set of benchmarks. They find that the double-decomposition technique mostly outperforms the single-decomposition technique. And this is true if you choose the same decomposition parameters for both techniques. But this is not how we would choose parameters: Given N and q, we would actually choose ω for the single-decomposition technique optimal or ω and $\tilde{\omega}$ for the double-decomposition technique. Also, we want to cover a large possible set of use cases: We generate parameter sets using 50 %, 65 %, 75 %, 80 %, 85 %, 90 %, and 95 % of the available modulus space for each degree N. For each technique, we choose the decomposition parameters optimal with respect to the number of NTTs `ntt`. In Figure 1, we compare the single-decomposition and double-decomposition technique, lower times are better. As we can see, the double-decomposition technique only gets competitive close to the maximum modulus size for q. In fact, the single-decomposition technique outperforms the double-decomposition technique for all parameter sets except for $N = 2^{16}$ with $\log_2 q = 1624$. Our answer to the question: In almost all cases, the single-decomposition technique is the better choice.

4.4 How large is the speed-up from constant folding?

As before, we want to cover many parameter sets to measure our improvements with constant folding. We reuse the parameters from Subsection 4.3 using 50 %, 65 %, 75 %, 80 %, and 85 % of the available modulus space for each degree N. In Table 5, we compare a non-optimized implementation (`naïve`) and with an optimized implementation (`folded`). We improve execution times by up to 11.6 % and, on average, by 4.8 %.

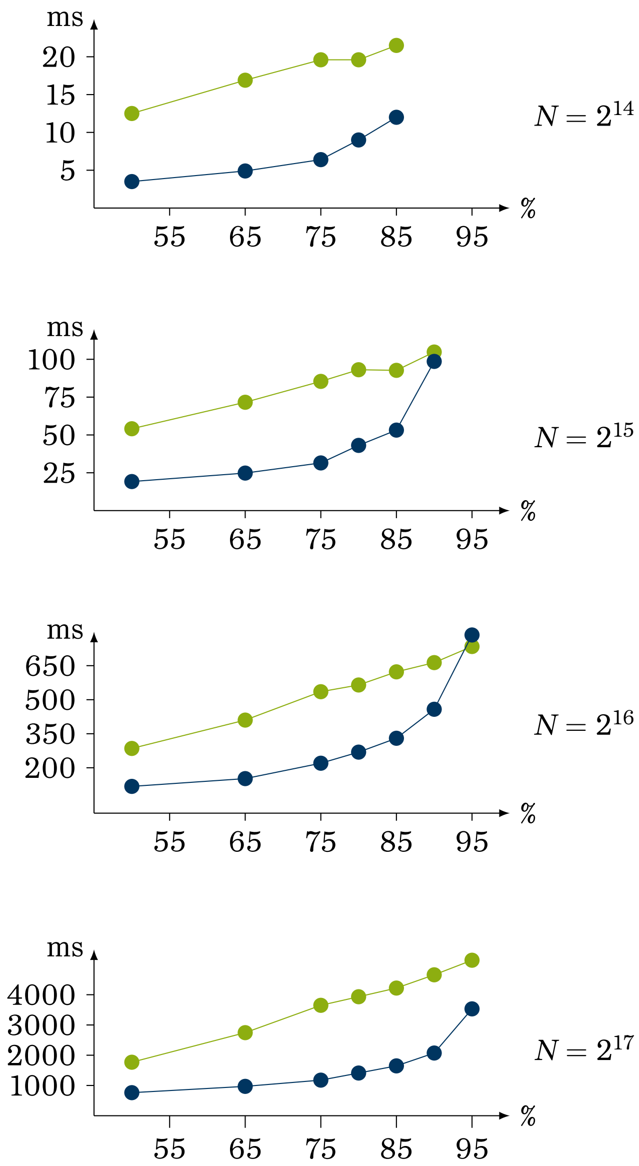

Figure 1: Execution times for the ● single-decomposition and ● double-decomposition technique. Given N and q, we choose parameters optimal with respect to the number of NTTs, respectively.

Table 5: Execution times for parameter sets where q uses 50 %, 65 %, 75 %, 80 %, and 85 % of the available modulus space. We also report the speed-up of `folded` compared to `naïve` in percent.

Parameters			Time (ms)		Speed-up
$\log_2 N$	$\log_2 q$	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 %

Table 6: Execution times for parameter sets where q uses 50 %, 66 %, and 75 % of the available modulus space. For each parameter set, we choose mostly primes with 54 bit, but at least $\kappa = 3$ primes with 36 bit. We also report the absolute overhead in *ms* of `switch` compared to `folded`.

Parameters		Time (*ms*)		
$\log_2 N$	$\log_2 q$	`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

4.5 How costly is replacing large with small primes?

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 generate parameter sets using 50 %, 66 %, and 75 % of the available modulus space for q with $b = 54$, $\kappa = 3$, and $\mu = 2$. We use at least κ 36 bit primes, all other primes are 54 bit. For example, for $N = 2^{16}$ and $\log q = 1116$, example, we use 4 primes with 36 bit and 18 primes with 54 bit. We report execution times and the absolute overhead of key switching replacing primes (`switch`) to the one without (`folded`) in Table 6. Overall, replacing primes does indeed have a low overhead. As expected, the relative overhead gets smaller the more primes we use.

4.6 How large is the speed-up using mostly large primes?

We re-use the parameter sets from Subsection 4.5 using 50 %, 66 %, and 75 % of the available modulus space for q with $b = 54$, $\kappa = 3$, and $\mu = 2$ with at least κ 36 bit primes, all other primes are 54 bit. Additionally, we generate a corresponding set using only small primes with 36 bit. As shown in Table 7, the speed-ups can be massive for using mostly large primes: On average, we speed up key switching by 36.9 % across all measured parameter sets. Since the speed-ups here are much larger than the overhead of replacing primes, using mostly large primes is highly effective for high performance.

Table 7: Execution times for parameter sets where q uses 50 %, 66 %, and 75 % of the available modulus space. For each with mostly large primes parameter set, we choose mostly primes with 54 bit, but at least $\kappa = 3$ primes with 36 bit. For parameter sets with small primes, we only use 36 bit primes. We also report the speed-up of using mostly large primes compared to only small primes in percent.

Parameters		Time (ms)		Speed-up
$\log_2 N$	$\log_2 q$	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 %

We therefore recommend that libraries of BGV-like schemes implement key switching with support for switching primes in and out.

4.7 Additional Remarks

In the beginning of our work, we mentioned that key switching occupies roughly 40 % of execution time during bootstrapping and is 11× slower compared to a naïve ciphertext multiplication. In the following, we want to expand on how and why key switching is considered the most significant bottleneck for most homomorphic use cases. We would argue there are three main reasons:

1 Most use cases require many key switching, even if the number of levels is rather low. Recall that BGV-like schemes encrypt a vector of integers or approximate numbers. We can only operate on numbers in different slots using rotations, each of which requires a key switching afterward. For example, homomorphic matrix multiplication [27] spends 50 % of execution time switching keys on our benchmarking setup. It encodes one matrix per ciphertext in vector form, but for multiplication, all elements in the encoded vector have to interact which needs many rotations.

2 On the server side, modulus switching and key switching are the

only homomorphic operations that require forward and inverse NTTs, the main computational bottleneck for FHE. As we have shown earlier, we can mostly integrate modulus switching with key switching, and hence key switching remains as an expensive bottleneck.

3 Reading key switching keys from memory is expensive, and we only use them for a single multiplication; usually, we then throw them away because the cache holds other data. This is especially relevant for hardware accelerators where reading the keys from memory becomes the main bottleneck [8, 15].

Improving key switching boosts performance for almost all homomorphic use cases. Thus, our work significantly impacts performance for BGV-like schemes.

4.8 Limitations and Future Work

We want to mention three limitations of our work:

1 For the single-decomposition technique, choosing $\omega \geq 2$ is optimal and will (mostly) result in best performance. For the double-decomposition technique, the trade-off between the number of NTTs and the key switching key size complicates parameter selection. Optimally, we would benchmark the individual operations ($\mathsf{NTT}_{\mathsf{fwd}}$, $\mathsf{NTT}_{\mathsf{inv}}$, coefficient-wise polynomial multiplication, coefficient-wise scalar multiplication, fetching ksk from memory) and optimize parameters specifically to an implementation on the target platform. But, doing so is hard to scale and we would still expect similar results when comparing both techniques.

2 We use the HPS method [19] for correcting the error after $\mathsf{BaseExt}$ if needed; the HPS method tends to perform better [3]. However, benchmarking results using the BEHZ method [4] would still be interesting.

3 An open problem is finding closed formulas for an optimal multiplication complexity in the double-decomposition technique, our best approximation remains roughly $\sqrt{\ell}$.

A great opportunity for future work is comparing the single- and double-decomposition techniques in hardware. Hardware accelerators shifts more costs to reading the key switching keys. We expect the single-decomposition technique to also perform better in hardware as it tends to have smaller keys, but it warrants an investigation nonetheless.

Conclusion

Section 5

In this work, we deep dive into the current state-of-the-art in key switching and thoroughly analyse its complexity. Along the way, we provided a new perspective on the asymptotic complexity with the bounds $\mathcal{O}(\omega\ell)$ and $\mathcal{O}(\omega\ell/\tilde{\omega}+\tilde{\omega}\ell/\omega)$ for the single- and double-decomposition technique, respectively. We also provide an estimate for the number of primes r in E (double-decomposition technique), reduce the number of scalar multiplications (both techniques), and correct the number of NTTs for BGV and CKKS (double-decomposition technique). We also revisit an idea by Gentry, Halevi, and Smart [18], integrate it with key switching with low overhead, and show that $b \approx \beta \approx \tilde{\beta} \approx B$ is a reasonable assumption in practice. Our results are not only theoretical, we confirm them with benchmarks: Reducing the number of multiplication speeds up key switching by up to 11.6 % and choosing mostly large primes by up to 63 %. In the beginning, we also promised answers to three questions. Here they are:

1 Do I want to implement the more complex double-decomposition technique? If yes, when do I want to use it?

Probably not. In most cases, the single-decomposition technique outperforms the double-decomposition technique. But please implement it if you want to, after all, it is a cool idea and maybe future work can improve upon it.

2 Do I always want to stick with a given N and q in the single-decomposition technique? Or can I adjust them to get better performance?

Yes and no. You most likely do not want to increase N. But you definitely want to adjust q for BGV and CKKS: Use as many large primes as possible! Add some small ones for scaling and replace them with large ones during key switching; it can massively boost performance.

3 How do I set the parameters P and ω for best performance?

It is actually rather easy: Given N and q, simply choose $\omega \geq 2$ as small as possible within your security requirements.

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 Sa-

hai, 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] Ahmad Al Badawi, Yuriy Polyakov, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Kurt Rohloff. “Implementation and Performance Evaluation of RNS Variants of the BFV Homomorphic Encryption Scheme”. In: IEEE Trans. Emerg. Top. Comput. 9.2 (2021), pp. 941–956. DOI: 10.1109/TETC.2019.2902799. URL: https://doi.org/10.1109/TETC.2019.2902799.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Cryptography Research Centre. *fhelib*. https://github.com/Crypto-TII/fhelib. 2023.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Shai Halevi and Victor Shoup. "Design and implementation of HElib: a homomorphic encryption library". In: IACR Cryptol. ePrint Arch. (2020), p. 1481. URL: https://eprint.iacr.org/2020/1481.

[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.