CKKS Key Switching and its Implementation in Liberate-FHE

Eric Crockett

May 7, 2025

List of Corrections

1 Introduction

This article delves into all operations that manipulate an FHE ciphertext modulus, and focuses on the algorithms implemented in Liberate-FHE. In particular, it covers basis extension, rescaling, and key-switching. Section 2 describes the notation, and Section 3 gives the intuition for the fundamentals of key-switching. Sections 4 and 5 describe two key operations required for the actual key-switching algorithm, and Section 6 describes the actual key-switching procedure.

2 Notation

We represent $x \in \mathbb{Z}_q$ by its *canonical representative* $[x]_q \in [\lceil -q/2 \rceil, \lfloor q/2 \rfloor) \subset \mathbb{Z}$. We also use the *standard representative* $|x|_q \in [0, q) \subset \mathbb{Z}$.

Most homomorphic encryption implementation utilize the Chinese Remainder Theorem (CRT), also called the Residue Number System (RNS) to avoid confusion with other uses of the CRT in lattice cryptography. Let $q_0, \ldots, q_{k-1}$ be co-prime moduli, and let $Q_j = \prod_{i=0}^{j} q_i$. Define $Q := Q_{k-1}$ and $Q_{-1} := 1$. We denote $Q_i^* = Q/q_i \in \mathbb{Z}$ and $\tilde{Q}_i = (Q_i^*)^{-1} \in \mathbb{Z}_{q_i}$. In CRT/RNS, we identify $x \bmod Q$ with $(x_0, \ldots, x_{k-1})$ where $x_i \in \mathbb{Z}_{q_i}$. We can reconstruct $x = |x|_Q$ as $\sum_{i=0}^{k-1} |x_i|_{q_i} \cdot \tilde{Q}_i \cdot Q_i^* \bmod Q$. We call $\{q_0, q_1, \ldots, q_{k-1}\}$ the *RNS basis* of x. By "an element x in basis $\mathcal{B}$", we mean x in RNS form with respect to the moduli contained in $\mathcal{B}$. By the Chinese Remainder Theorem (CRT), there is precisely one integer x such that x in basis $\mathcal{B}$ is $(x_0, x_1, \ldots, x_{k-1})$.

3 Intuition

This section describes the intuition for key switching, but does *not* accurately describe the actual key switching algorithm, which we leave to later sections. Key switching is a technique for converting a ciphertext encrypted under a key s_1 into a ciphertext encrypted under some other key s_2, originally described in [BV11]. It is primarily used in *relinearization* and *rotations*.

3.1 Key-Switch Key Generation

Roughly, the key-switching key ksk is an LWE ciphertext encrypting s_1 under s_2:

$$ksk = (-a \cdot s_2 + s_1 + e, a)$$

where a is random and e is drawn from the LWE error distribution.

3.2 Key-Switching for Quadratic Ciphertexts (Relinearization)

The result of multiplying two linear ciphertexts is a quadratic (i.e., three-term) ciphertext (c_0, c_1, c_2) such that $c_0 + c_1 \cdot s + c_2 \cdot s^2 \approx m$. To prevent exponential[1] growth in the ciphertext size, we can apply key-switching from the key $s_1 = s^2$ to the key $s_2 = s$, a process called *relinearization*.

Relinearization outputs a new (linear[2]) ciphertext

$$ct' = (c_0, c_1) + c_2 \cdot ksk = (c_0' = c_0 + c_2 \cdot (-a \cdot s + s^2 + e), c_1' = c_1 + c_2 \cdot a).$$

We can verify that ct' decrypts to m under s:

$$\begin{aligned} c_0' + c_1' \cdot s &= (c_0 + c_2 \cdot (-a \cdot s + s^2 + e)) + (c_1 + c_2 \cdot a) \cdot s \\ &= c_0 - c_2 \cdot a \cdot s + c_2 \cdot s^2 + c_2 \cdot e + c_1 \cdot s + c_2 \cdot a \cdot s \\ &= c_0 + c_1 \cdot s + c_2 \cdot s^2 + c_2 \cdot e \end{aligned}$$

Note that this is the same as decrypting the original ciphertext under s, plus an error term $c_2 \cdot e$.

[1] in the depth of the circuit
[2] in the secret key

3.3 Key-Switching for Linear Ciphertexts (Rotations)

Given a ciphertext ct encrypting a message m under a key s, there is a CKKS operation which rotates the "slots" of a message cyclically by i places. As described in [GHS12, Section 2.2], this is achieved by evaluating the polynomial $f(X)$ at X^j for some $j(i)$. The exact description of Galois automorphisms is beyond the scope of this paper. Let κ_j denote this transformation. Let $ct^{(j)}$, $m^{(j)}$, and $s^{(j)}$ be the result of applying κ_j to ct, m, and s, respectively. The rotation operation produces $ct^{(j)}$, which encrypts $m^{(j)}$ under $s^{(j)}$. We seek to use key-switching to obtain an encryption of $m^{(j)}$ under s.

Given a linear ciphertext $(c_0^{(j)}, c_1^{(j)})$ satisfying $c_0^{(j)} + c_1^{(j)} \cdot s^{(j)} \approx m^{(j)}$, we can construct a new ciphertext (c'_0, c'_1) satisfying $c'_0 + c'_1 \cdot s \approx m^{(j)}$ by constructing a temporary ciphertext $t = (c_0^{(j)}, 0, c_1^{(j)})$, and then applying the relinearization procedure as described above, which outputs the ciphertext

$$ct' = (c_0^{(j)}, 0) + c_1^{(j)} \cdot ksk = (c'_0 = c_0^{(j)} + c_1^{(j)} \cdot (-a \cdot s + s^{(j)} + e), c'_1 = c_1^{(j)} \cdot a).$$

We can verify that ct' decrypts to m under s:

$$\begin{aligned} c'_0 + c'_1 \cdot s &= (c_0^{(j)} + c_1^{(j)} \cdot (-a \cdot s + s^{(j)} + e)) + c_1^{(j)} \cdot a \cdot s \\ &= c_0^{(j)} - c_1^{(j)} \cdot a \cdot s + c_1^{(j)} \cdot s^{(j)} + c_1^{(j)} \cdot e + c_1^{(j)} \cdot a \cdot s \\ &= c_0^{(j)} + c_1^{(j)} \cdot s^{(j)} + c_1^{(j)} \cdot e \end{aligned}$$

Note that this is the same as decrypting the original ciphertext under $s^{(j)}$, plus an error term $c_1^{(j)} \cdot e$. See [GHS12, Appendix B] for more details.

3.4 Controlling Noise Growth

The problem with the intuition above comes down to the additional error term. Informally, error terms have small coefficients, while c_2 is an arbitrary ring element mod Q, hence it has coefficients in $[0, Q)$. As a result, $c_2 \cdot e$ (or $c_1^{(j)} \cdot e$ for rotations) does *not* have small coefficients. This results in fast noise growth, which limits the number of homomorphic operations that can be performed before the message is lost. As a result, we need a way to apply the key-switching procedure *without* incurring the large noise growth. The solution is to "decompose" c_2 into several "small" pieces, which will reduce the final additional error.

3.4.1 Base-b Decomposition

Traditionally, the approach was to decompose c_2 with respect to some base (radix) b; i.e. write $c_2 = \sum_{i=0}^{t} c_{2,i} \cdot b^i$ where $t = \lceil \log_b Q \rceil$ and $c_{2,i}$ has coefficients in $[0, b)$. Let $\text{Decompose}(c_2) = \{c_{2,i}\}_{0 \le i \le t}$. Then we make a corresponding series of key-switch keys: rather than encrypting s_1, we instead encrypt *powers* of s_1, i.e., $\{b^i \cdot s_1\}_{0 \le i \le t}$. Let $\text{Power}(s) = \{s \cdot b^i\}_{0 \le i \le t}$ The key feature is that the error term of each key-switch key is *not* scaled up by the same power. Let ksk_i denote the encryption of $b^i \cdot s_1$. Now, instead of computing $c_2 \cdot ksk$, we compute $\langle \text{Decompose}(c_2), ksks \rangle$. Now we can see that this combination of operations allows us to compute $c_2 \cdot ksk$, but with a smaller error:

$$\begin{aligned} \langle \text{Decompose}(c_2), ksks \rangle &= \langle \{c_{2,i}\}_{0 \le i \le t}, \{ksk_i\}_{0 \le i \le t} \rangle \\ &= \sum_{i=0}^{t} c_{2,i} \cdot ksk_i \\ &= \sum_{i=0}^{t} \big(c_{2,i} \cdot (-a_i \cdot s + b^i \cdot s^2 + e_i),\ c_{2,i} \cdot a_i\big) \\ &= \left(-s \cdot \sum_{i=0}^{t} (c_{2,i} \cdot a_i) + s^2 \cdot \sum_{i=0}^{t} \big(c_{2,i} \cdot b^i\big) + \sum_{i=0}^{t} c_{2,i} \cdot e_i,\ \sum_{i=0}^{t} c_{2,i} \cdot a_i \right) \\ &= \left(-s \cdot \sum_{i=0}^{t} (c_{2,i} \cdot a_i) + s^2 \cdot c_2 + \sum_{i=0}^{t} c_{2,i} \cdot e_i,\ \sum_{i=0}^{t} c_{2,i} \cdot a_i \right) \end{aligned}$$

This is roughly an encryption of $c_2 \cdot s^2$, but the error incurred from key switching is only $\sum_{i=0}^{t} c_{2,i} \cdot e_i$, which has coefficients like $\lceil \log_b Q \rceil \cdot b \cdot |e|)$ compared to $e \cdot c_2$ (with coefficients like $Q \cdot |e|$).

The base-b decomposition is natural, especially when using a BigNum implementation. One downside to this approach is that instead of needing a single key-switching key, we now need $t = \lceil \log_b Q \rceil$ for some small b, which dramatically increases the cost of key switching.

3.4.2 CRT Decomposition

When using RNS representation for ring elements, the base-b decomposition requires a CRT reconstruction in order to extract the base-b digits. [Baj+16] introduced a new mechanism for decomposition based on the CRT structure of the modulus Q. Per the CRT reconstruction formula, we can write

$$x = \sum_{i=0}^{k-1} |x|_k \cdot \tilde{Q}_i \cdot Q_i^* = \sum_{i=0}^{k-1} |x \cdot \tilde{Q}_i|_{q_i} \cdot Q_i^*.$$

where the latter equality holds because $\tilde{Q}_i \in \mathbb{Z}_{q_i}$. We can use this instead of the base-b decomposition because the coefficients of $|x \cdot \tilde{Q}_i|_{q_i}$ are in $[0, q_i)$. If each q_i is roughly ν bits, then this yields the same noise growth as a radix-2^ν decomposition.

Note that we $|x \cdot \tilde{Q}_i|_{q_i} = |x|_{q_i} \cdot \tilde{Q}_i$, and we already know $|x|_{q_i}$ from the RNS representation of x, thus this step amounts to multiplying each RNS component by a scalar.

3.4.3 Special Primes

[GHS12] introduced the concept of artificially raising the modulus during key-switching to control noise rather than a base-b decomposition. The intuition behind this approach is to pick a large modulus P and set the key-switch key to be an encryption of $P \cdot s_1$. In the basic relinearization operation described above, relinearization outputs $(c_0, c_1) + c_2 \cdot ksk$. When using a special prime, we output $ct' = (c_0, c_1) + \left\lfloor \frac{c_2}{P} \cdot ksk' \right\rceil$, where ksk' is an encryption of $P \cdot s$. We can verify that ct' decrypts to m under s:

$$\begin{aligned}
c'_0 + c'_1 \cdot s &= (c_0 + \left\lfloor \frac{c_2}{P} \cdot (-a \cdot s + P \cdot s^2 + e) \right\rceil) + (c_1 + \left\lfloor \frac{c_2}{P} \cdot a \right\rceil) \cdot s \\
&= c_0 + c_2 \cdot s^2 + \left\lfloor -\frac{c_2}{P} \cdot a \cdot s + \frac{c_2}{P} \cdot e \right\rceil + c_1 \cdot s + \left\lfloor \frac{c_2}{P} \cdot a \cdot s \right\rceil \\
&= c_0 + c_2 \cdot s^2 + \left\lfloor -\frac{c_2}{P} \cdot a \cdot s \right\rceil + \left\lfloor \frac{c_2}{P} \cdot e \right\rceil + c_1 \cdot s + \left\lfloor \frac{c_2}{P} \cdot a \cdot s \right\rceil + e_{\text{round}} \\
&= c_0 + c_1 \cdot s + c_2 \cdot s^2 + \left\lfloor \frac{c_2}{P} \cdot e \right\rceil + e_{\text{round}}
\end{aligned}$$

Note that this is the same as decrypting the original ciphertext under s, plus an error term $\left\lfloor \frac{c_2}{P} \cdot e \right\rceil + e_{\text{round}}$. The coefficients of c_2 are independent of P, hence by making P larger, we can reduce the affect of that term. Note that e_{round} is very small, with coefficients in $[-1/2, 1/2)$.

3.4.4 Hybrid Key-Switching

[GHS12] points out that, in order to minimize the noise incurred during key-switching with the special-modulus approach, we need $P > Q$, since the coefficients of c_2 are in $[0, Q)$. For a fixed number of levels, this requires doubling the ring dimension, which has a super-linear affect on performance. The authors note that we don't have to choose between expensive key-switching (via the base-b decomposition) and low utility (via the use of a special modulus): we can instead combine the decomposition and special modulus approaches to get a fine-grained tradeoff between the downsides of both approaches. Specifically, we use base-B decomposition (for some largish B), and then use a special modulus $P > B$ to reduce the noise incurred by using a large radix. The large radix significantly reduces the cost of key-switching (and the size of the key-switching keys) relative to radix-b, and using $P \approx B$ yields much more utility than $P \approx Q$. Note that this extends to the CRT decomposition approach in a straightforward fashion: one might consider q_i to be a "large" radix anyway, but we can make it even larger by grouping multiple q_is together. This directly leads to the algorithms used in practice today.

4 Basis Conversion

One of the fundamental operations required for key-switching is basis conversion. Let $\mathcal{B} = \{q_i\}_{0 \le i < k}$ be a basis, and $x \in \mathbb{Z}_Q$ be represented in basis $\mathcal{B}$. By "basis conversion", we mean that we wish to find $[x]_Q$ in basis $\mathcal{C}$ (which we can assume is disjoint from $\mathcal{B}$). We denote this as $\text{Conv}_{\mathcal{B} \to \mathcal{C}}(\cdot)$. One simple way to do this is to use CRT reconstruction to compute $[x]_Q$ over the integers, and then reduce it mod p_i for each $p_i \in \mathcal{C}$. However computing the CRT reconstruction over the integers may require BigInteger arithmetic, so we seek to convert x in basis $\mathcal{B}$ to $[x]_Q$ in basis $\mathcal{C}$ *directly*, without doing a full CRT reconstruction of x. Note that we can obtain basis *extension* by concatenating the CRT components of disjoint bases $\mathcal{B}$ (i.e., the input) and $\mathcal{C}$ (i.e., the output); we generally use these terms interchangeably.

4.1 Garner's Algorithm

Liberate-FHE uses a tweak of Garner's algorithm [GCL92, Section 5.6] for basis conversion, which is based on the mixed-radix CRT reconstruction algorithm:

$$[x]_Q = \sum_{i=0}^{k-1} [c_i]_{q_i} \cdot Q_{i-1} \in \mathbb{Z}.$$

where

$$c_i = \left(x_i - \sum_{j=0}^{i-1} [c_j]_{q_j} \cdot Q_{j-1} \right) \cdot Q_{i-1}^{-1} \in \mathbb{Z}_{q_i}.$$

Note that the c_i can be computed without the use of BigInteger arithmetic since all individual terms are reduced mod q_i, and all of the arithmetic is in $\mathbb{Z}_{q_i}$.

For the purposes of basis extension, we are interested in computing $[x]_p$ for some $p \in \mathcal{C}$. Since the sum for x is over the integers,

$$[x]_Q \equiv \sum_{i=0}^{k-1} [c_i]_{q_i} \cdot [Q_{i-1}]_p \bmod p,$$

where $[Q_{i-1}]_p$ can be pre-computed. This gives us a way to compute $[x]_Q \bmod p$ without using BigInteger arithmetic since each term is mod a small prime, the product is mod p, and the sum is mod p.

This algorithm has two steps:

1. Compute the c_i
2. Compute $[x]_Q \bmod p \in \mathbb{Z}_p$ for each $p \in \mathcal{C}$

The first step is inherently quadratic in $|\mathcal{B}|$. The second step is linear for each $p \in \mathcal{C}$, giving an overall cost of $\mathcal{O}(|\mathcal{B}|^2 + |\mathcal{B}| \cdot |\mathcal{C}|)$.

The algorithm is described over the integers, but it trivially extends component-wise to ring elements. Also note that this algorithm is described using the canonical representative and outputs $[x]_Q \bmod p$, but it works equally well with standard representatives to obtain $|x|_Q \bmod p$ by replacing all canonical representatives with standard representatives.

4.1.1 Implementation

The following refers to the DeSilo C++ code provided to Cornami. Liberate builds the c_i one term of the sum at a time. Specifically, `pre_extend` *should* maintain the following invariants at the start of iteration i (named `index` in the code):

- pre_extended$_j$ holds $[c_j]_{q_j}$ for $j \leq i$
- pre_extended$_j$ holds $\sum_{k=0}^{i} [c_k]_{q_j} \cdot [Q_{k-1}]_{q_j}$ for $j > i$

Then in iteration i, `pre_extend` computes

$$\begin{aligned}
\text{pre_extended}_{i+1} &= \left[(\text{partition}_{i+1} - \text{pre_extended}_{i+1}) \cdot Q_i^{-1}\right]_{q_i} \\
&= \left[(\text{partition}_{i+1} - \sum_{k=0}^{i} [c_k]_{q_{i+1}} \cdot [Q_{k-1}]_{q_{i+1}}) \cdot Q_i^{-1}\right]_{q_i} \\
&= [c_i]_{q_i}
\end{aligned}$$

This maintains the invariant that state$_{i+1}$ holds c_{i+1}. The inner `for` loop maintains the second invariant. However, there is a problem: `pre_extend` sets pre_extended$_i$ to the output of `mont_enter`, which outputs a value in $[0, 2q_i)$. It should output a value in $[0, q_i]$, since that is the range of $[c_i]_{q_i}$.

`extend` uses the $[c_i]_{q_i}$ to compute $x \bmod p_j$ for each $p_j \in \mathcal{C}$. We note that if $p_j \in \mathcal{B}$ is also in $\mathcal{C}$ (as it is in key-switching), we do *not* need to run `extend` for p_j, since $y_j = x_j$.[3]

4.2 Fast Basis Conversion

[HPS18] introduced an alternative to Garner's algorithm which they call "fast basis conversion" (FBC). Like Garner's algorithm, FBC enables CRT basis conversion/extension without doing a CRT reconstruction. FBC utilizes floating-point arithmetic. Due to the finite precision inherent in the use of floating-point arithmetic, FBC has a low probability of introducing small approximation errors, which "in the context of homomorphic operations are inconsequential".

Garner's algorithm is based on the mixed-radix CRT reconstruction formula. By comparison, FBC uses the following reconstruction formula:

$$[x]_Q = \left(\sum_{i=1}^{k} [x_i \cdot \tilde{Q}_i]_{q_i} \cdot Q_i^*\right) - \left\lfloor \sum_{i=i}^{k} \frac{[x_i \cdot \tilde{Q}_i]_{q_i}}{q_i} \right\rceil \cdot Q \in \mathbb{Z},$$

where the division is floating-point.

Thus, we compute $y_i = [x_i \cdot \tilde{Q}_i]_{q_i} \in \mathbb{Z}_{q_i}$, the rationals $z_i = y_i/q_i$, and $v = \left\lfloor \sum_{i=i}^{k} z_i \right\rceil \in \mathbb{Z}$. Then

$$[x]_Q \equiv \left(\sum_{i=1}^{k} y_i \cdot [Q_i^*]_p\right) - v \cdot [Q]_p \bmod p.$$

When $p = \prod_{j=1}^{k'} p_j$, we compute $[x]_{p_j}$ for each j, reusing the precomputed value of v.

This process is exact up to floating-point precision, which is insignificant relative to the noise added to ciphertexts for security purposes.

This is actually used in practice; see [Bad+18] for more details.

FBC is a two-step process:

1. Compute y_i, v_i, and v
2. Compute $[x]_p$ for each $p \in \mathcal{C}$

The first step is $\mathcal{O}(|\mathcal{B}|)$. The second step is $\mathcal{O}(|\mathcal{B}| \cdot |\mathcal{C}|)$. Thus this approach is better than Garner's algorithm, especially in cases where $|\mathcal{B}|$ is large (which is true when k is large), at the cost of the use of floating point arithmetic and a small amount of error.

[3] However, an implementation *may* choose to do so, as it is mathematically correct. Liberate-FHE takes this approach.

4.3 Adding Error without Floating Point Arithmetic

[Baj+16] proposed an alternate algorithm that is similar to Section 4.2, but it does not require floating-point arithmetic. Essentially, it skips the step in Section 4.2 that subtracts off a multiple of q. As a result, the answer isn't quite correct, and correction is needed, which takes additional operations.

It defines

$$\mathrm{FBC}(x, q, \mathcal{B}) = \left(\sum_{i=1}^{k} \left| x_i \cdot \tilde{Q}_i \right|_{q_i} \cdot Q_i^* \bmod m \right)_{m \in \mathcal{B}}$$

where $|\cdot|_q$ denotes the standard representative.

[HPS18] concludes:

> Compared to [Baj+16], our procedures and simpler and faster, and introduce a lower amount of noise.

[Bad+18] compares the approaches in [Baj+16] and [HPS18], and essentially concludes the same thing.

This is used in [Che+18a]. It is also used in [HK19][4], which appears to *not* include any of the correction steps. It says:

> It includes the noise which can be ignored in the case of the HEAAN-RNS scheme. Even though the effect of this noise is negligible, we can reduce its size further by adapting the algorithms introduced in [HPS18].

4.4 Reduce Error without Floating Point Arithmetic

Recently, [QX24] gave an alternate to FBC that replaces the floating-point arithmetic in [HPS18] with integer addition. It requires some knowledge of how the RNS primes are selected (relative to Δ). Eric: I think this has roughly the same complexity as FBC, but without floating-point arithmetic, and without the possibility of floating-point error. Investigate. Eric!

4.5 Comparison

Assume basis conversion from $\mathcal{B}$ to $\mathcal{C}$, where B and C are disjoint.

The complexity of Garner's algorithm is $\mathcal{O}(|\mathcal{B}|^2 + |\mathcal{B}||\mathcal{C}|)$.

The complexity of FBC is $\mathcal{O}(|\mathcal{B}| + |\mathcal{B}||\mathcal{C}|)$, but requires the use of floating-point arithmetic.

5 Rescaling

Rescaling is an operation that divides a ciphertext by some constant c, rounding the result. Specifically, the goal is to convert $x \in \mathbb{Z}_Q$ to $y = \lfloor Q'/Q \cdot x \rceil \in \mathbb{Z}_{Q'}$ for some integer $Q' \geq 2$. In all the cases we consider, $Q' \mid Q$, so let $\mathcal{B} \cup \mathcal{C}$ be the basis of Q and $\mathcal{B}$ be the basis of Q'. In the CKKS scheme the main reason for rescaling is not to manage the noise, as in the case of the Brakerski-Gentry-Vaikuntanathan (BGV) scheme, but to scale down the encrypted message and truncate some least significant bits. This operation is used after (or before, if using [KPP20]) multiplication to reduce the scale factor from Δ^2 to Δ. In this context in an RNS implementation, $|\mathcal{C}| = 1$. Rescaling is also used in key-switching, as described in Section 3.4.3. In that context in an RNS implementation, $\mathcal{C}$ is the basis of the special primes, and $k = |\mathcal{C}| \geq 1$.

In the original (non-RNS) CKKS scheme [Che+16], $q_\ell = p^\ell \cdot q_0$. This paper introduces a function $\mathrm{RS}_{\ell \leftarrow \ell'}(\mathbf{c}) = \left\lfloor \frac{q_{\ell'}}{q_\ell} \mathbf{c} \right\rceil \bmod q_{\ell'}$. This actually multiples (each coefficient of) the ciphertext by the rational $\frac{q_{\ell'}}{q_\ell} = p^{\ell' - \ell}$, and then rounds to the nearest integer. This algorithm assumes that the input is represented mod the full modulus Q. This is the algorithm used in the first two (non-RNS) bootstrapping papers for CKKS [Che+18b; CCS18].

When using RNS representation, an easy way to implement rescaling is via CRT reconstruction. Given an input in RNS representation, reconstruct it mod the full modulus Q, then divide and round the coefficients, and convert back to RNS representation. As with basis conversion, we seek an algorithm that allows us to rescale directly in RNS representation, without an expensive CRT reconstruction.

5.1 Algorithms

We give three algorithms for rescaling. See Appendix B for a proof of equivalence.

5.1.1 Rescaling as Basis Conversion

In the RNS version of CKKS [Che+18a] (also [HK19; Baj+16]), rescaling is defined for an RNS ring element $x \in R_{q_\ell}$ where $x = (x_0, x_1, \ldots, x_\ell)$. It defines $\mathrm{RS}(x) = \{q_\ell^{-1} \cdot (x_i - x_\ell) \bmod q_i\}_{0 \leq i < \ell}$. When applied component-wise to a ciphertext encrypting a message m at level ℓ, this algorithm outputs an encryption of $q_\ell^{-1} \cdot m \approx \Delta^{-1} \cdot m$ at level $\ell - 1$. There is some error introduced due to the difference between q_ℓ and Δ. This procedure computes the same thing as the rescale procedure in [Che+16].

The algorithm above requires multiple calls to remove multiple moduli, which is relevant in the context of key-switching. [HK19] (and possibly others) note that we can generalize this function using basis extension. It's not a different algorithm, but it does have a simple description. Let $\mathcal{B} \cup \mathcal{C}$ be the basis of the input modulus Q, and $\mathcal{B}$ be the basis of the output modulus Q'. In short, [HK19] describes Rescale as

$$(|x|_{\mathcal{B}} - \mathrm{Conv}_{\mathcal{C} \to \mathcal{B}}(|x|_{\mathcal{C}})) \cdot \left| C^{-1} \right|_B,$$

[4] It defines the algorithm using the canonical representative, without comment

where B (resp. C) is the product of the primes in the basis $\mathcal{B}$ (resp. $\mathcal{C}$), and $|x|_{\mathcal{B}}$ is a short-hand for x in RNS representation with respect to basis $\mathcal{B}$.

This is essentially the same as calling Rescale multiple times, and it's easy to see when $|\mathcal{C}| = 1$ that this is identical to the algorithm from [Che+18a]. Any basis conversion algorithm from Section 4 can be used to implement $\text{Conv}_{\mathcal{C}\to\mathcal{B}}$, and the complexity of this operation is $|\mathcal{B}|$ plus that of the basis conversion algorithm.

5.1.2 (Fast) Scaling in CRT Representation

[HPS18] gives yet another algorithm for rescaling, based on floating-point arithmetic. For this functionality, we consider a third CRT reconstruction formula:

$$[x]_Q = \left(\sum_{i=1}^{k} x_i \cdot \tilde{Q}_i \cdot Q_i^*\right) - v' \cdot Q \in \mathbb{Z}$$

Note: $x_i, \tilde{Q}_i \in [-q_i/2, q_i/2)$ and $Q_i^* = Q/q_i \in \mathbb{Z}$, so the product $x_i \cdot \tilde{Q}_i \cdot Q_i^* \in [-\frac{q_i^2}{4}\frac{Q}{q_i}, \frac{q_i^2}{4}\frac{Q}{q_i}) = [-\frac{q_i \cdot Q}{4}, \frac{q_i \cdot Q}{4})$.

Suppose we want to "scale down" x by a Q'/Q factor, i.e., to compute $y = \lfloor Q'/Q \cdot x \rceil \in \mathbb{Z}_t$ (where we assume $Q' \mid Q$). Then [HPS18] shows that we can use the reconstruction formula above to write:

$$\left\lfloor Q'/Q \cdot [x]_Q \right\rceil \equiv \left\lfloor \sum_{i=1}^{k} [x_i]_{q_i} \cdot \left(\tilde{Q}_i \cdot \frac{Q'}{q_i}\right) \right\rceil \bmod Q'$$

Note that this formula assumes the input is in RNS form, and the output is a standard integer mod Q'. We can get the output in RNS form by computing the sum multiple times as

$$\left\lfloor \sum_{i=1}^{k} [x_i]_{q_i} \cdot \left(\tilde{Q}_i \cdot \frac{Q'}{q_i}\right) \right\rceil \bmod t$$

for each t in the basis of Q'.

As in Section 4.2, we write $\tilde{Q}_i \cdot \frac{Q'}{q_i}$ as $\omega_i + \theta_i$ with $\omega_i \in \mathbb{Z}_t$ and $\theta_i \in [0, 1)$.[5] We can precompute the $v = \lfloor \sum_i [x_i]_{q_i} \theta_i \rceil \in \mathbb{Z}$ using floating-point arithmetic[6] and reuse this for each t in the basis of Q'. Then we compute $w = \sum_i \left([x_i]_{q_i} \omega_i \bmod t \right) \in \mathbb{Z}_t$ using integer arithmetic, and output $w + (v \bmod t) \in \mathbb{Z}_t$.

This algorithm also works with standard representatives:

$$\lfloor Q'/Q \cdot x \rfloor \equiv \left\lfloor \sum_{i=1}^{k} |x_i|_{q_i} \cdot \left(\tilde{Q}_i \cdot \frac{Q'}{q_i}\right) \right\rfloor \bmod Q'$$

This requires rounding v with $\lfloor \cdot \rfloor$.

5.2 Comparison

For now, we consider both basis conversion algorithms, plus the algorithm from Section 5.1.2. Note that when considering basis conversion algorithms, rescaling requires conversion *from* $\mathcal{C}$ *to* $\mathcal{B}$, which is backwards from the standard notation of basis conversion.

Using Garner's algorithm, we can achieve rescaling in $\mathcal{O}(|\mathcal{C}|^2 + |\mathcal{B}||\mathcal{C}|)$ (the extra $|\mathcal{B}|$ term is subsumed by larger terms).

Using FBC, we can achieve rescaling in $\mathcal{O}(|\mathcal{B}| + |\mathcal{B}||\mathcal{C}|)$ (but the constant in front of $|\mathcal{B}|$ is now at least 2).

Using Section 5.1.2, we can achieve rescaling in $\mathcal{O}(|C| + |B| \cdot |C|)$. We should prefer this over FBC, especially since $|C|$ is typically very small.

5.3 Rescaling with Standard Representatives

Many software libraries natively use the "standard representative" $|x|_q \in [0, q)$ as the $\mathbb{Z}$-representative of an element of $\mathbb{Z}_q$, rather than the canonical representative in $[x]_q \in [\lceil -q/2 \rceil, \lfloor q/2 \rfloor)$. With this notation, we can now consider rescaling with standard representatives.

If $|x_\ell|_{q_\ell} \leq \lfloor q_\ell/2 \rfloor$, then $|x_\ell|_{q_\ell} = [x_\ell]_{q_\ell}$, hence nothing changes.

If, $|x_\ell|_{q_\ell} > \lfloor q_\ell/2 \rfloor$, $|x_\ell|_{q_\ell} = [x_\ell]_{q_\ell} + q_\ell$. Now $\frac{x - |x_\ell|_{q_\ell}}{q_\ell} = \frac{x - [x_\ell]_{q_\ell} - q_\ell}{q_\ell} = \frac{x - [x_\ell]_{q_\ell}}{q_\ell} - 1$.

Thus, if we rescale with the standard representative, we need to adjust the output by adding

$$\hat{x_\ell} = \begin{cases} 0 & |x_\ell|_{q_\ell} \leq \lfloor q/2 \rfloor \\ 1 & |x_\ell|_{q_\ell} > \lfloor q/2 \rfloor \end{cases}$$

With this definition, we can succinctly define rescaling of a single element in the standard basis as $q_\ell^{-1} \cdot (x - |x_\ell|_{q_\ell}) + \hat{x_\ell} \bmod Q_{\ell-1}$.

[5] Note that the exact range of θ_i does not impact the computation; we can use $\theta_i \in [0, 1)$ with canonical or standard representatives/nearest-integer or floor rounding. The purpose of splitting the constant is to ensure higher precision of the computation.

[6] Note that the rounding strategy here should match the rounding strategy of the output, i.e., $\lfloor \cdot \rceil$ if using canonical representatives, and $\lfloor \cdot \rfloor$ if using standard representatives.

5.3.1 Rescaling with Floor

We can view rescaling with standard representatives as rescaling with floor rounding instead of nearest-integer rounding. This is pretty easy to see: the standard representative is just the remainder, hence $\frac{x-|x|_q}{q} = \lfloor x/q \rfloor$. This definition of rescale introduces some additional error, as explored below.

In key switching, we round the *ciphertext*. Thus rounding error is applied *to the ciphertext*, not to the message. In other words, after rounding, the decryption equation becomes

$$(c_0 + e_{\text{round}}) \cdot s + (c_1 + e_{\text{round}}) = m + e_{\text{ct}} + s \cdot e_{\text{round}} + e_{\text{round}}$$

We generally ignore the isolated e_{round} term since it is small relative to $s \cdot e_{\text{round}}$.

When performing nearest-integer rounding, e_{round} has coefficients in $[-\frac{1}{2}, \frac{1}{2})$. When performing floor rounding, e_{round} has coefficients in $[0, 1)$, hence using floor essentially doubles the rounding error compared to using nearest-integer.

5.4 Implementation

For the purposes of key-switching, we want to remove the special primes and scale down by their product. Algorithm 1 implements the removal of a single prime; this algorithm is iterated to remove multiple primes. In the implementation, we assume that the input is in RNS form, and that we want to remove and scale down by the last prime.

Algorithm 1 Rescale

procedure RESCALE($x = (x_0, x_1, \ldots, x_\ell)$, $\mathcal{B} = (q_0, q_1, \ldots, q_\ell)$)
  for i in range(ℓ) **do**
    $q_\ell^{-1} \leftarrow$ pow(q_ℓ, -1, q_i)
    $x_i \leftarrow (x_i - x_\ell) \cdot q_\ell^{-1}$
                ▷ Do the fixup if $x_i = |x|_{q_i}$ *and* you want $\lfloor \cdot \rceil$
  fixup $\leftarrow$ [1 if $x_\ell[i] > \lfloor q_\ell/2 \rfloor$ else 0 for i in range(N)] ▷ Skip the fixup if you want $\lfloor \cdot \rfloor$ or if $x_i = [x]_{q_i}$
  for i in range(ℓ) **do**
    $x_i \leftarrow x_i +$ fixup
  return $(x_0, x_1, \ldots, x_{\ell-1}$

This extends component-wise to ring elements.

5.4.1 A Word on ModDown

Liberate-FHE includes both a rescale function and a mod_down function. Mathematically, these should be the same function. However, there are some semantic differences, as well as a practical implementation difference.

Semantically, rescale divides the input by a *normal* prime, strips that prime from the (composite) modulus, and returns an output at the next level. mod_down, on the other hand, is about removing *special* primes from a "scaled-up" input. The output is at the same level as the input, since no normal primes were lost. Hence rescale is called after (or before) a multiplication, while mod_down is used in key-switching.

Liberate-FHE also (arbitrarily) chose to implement rescale with nearest-integer rounding, while mod_down is implemented with floor rounding.

6 Key Switching

This section describes the key-switching algorithm from [HK19], which is used in Liberate-FHE. We describe the algorithm without reference to RNS to simplify notation, but implicit in this description is that any ring element with a compound modulus (denoted by a capital letter) would be represented in an implementation in RNS form.

Let $\alpha = k$ be a parameter for the number of special primes, ℓ be the level of the input ciphertext, $Q = \prod_{i=0}^{\ell} q_i$, $\mathcal{Q} = \{q_i\}_{0 \le i \le \ell}$, $P = \prod_{i=0}^{k-1} p_i$, and $\mathcal{P} = \{p_i\}_{0 \le i < k}$. We partition the $\ell + 1$ normal primes into β groups of size $\approx \alpha$; note that some groups may have fewer than α primes. The composite modulus for each group is $Q_j = \prod_{i=j\alpha}^{(j+1)\alpha-1} q_i$. We also define $\hat{Q}_j = Q/Q_j$ for $0 \le j < \beta$.

At a high level, key switching involves:

1. **Decompose** Given a ring element a, output $\{a \cdot \hat{Q}_j^{-1} \bmod Q_j\}_{0 \le j < \beta}$.
2. **ModRaise** Raise the modulus of each $a \cdot \hat{Q}_j^{-1} \bmod Q_j$ to $P \cdot Q$, using basis extension (cf. Section 4).
3. **Dot Product** We can view the output of ModRaise as a row vector in $y \in R_{P \cdot Q}^{\beta}$, and we can view the key switch keys as a matrix in $K \in R_{P \cdot Q}^{\beta \times 2}$ We want to compute $y \cdot K \in R_{P \cdot Q}^{2}$.
4. **ModDown** Reduce the modulus of both outputs of the dot product to Q, while simultaneously diving the values by P.

The following subsections describe these steps in more detail.

6.1 Implementation

6.1.1 Decomposition

Given a ring element $a \in R_Q$ in RNS form, i.e. $a = (a_0 \in R_{q_0}, a_1 \in R_{q_1}, \ldots, a_L \in R_{q_L})$ split it into groups of size α corresponding to Q_j for $0 \leq j < \beta$. We now have β "states", each with $\leq \alpha$ ring elements. State j is $s_j = (a_{j\cdot\alpha}, a_{j\cdot\alpha+1}, \ldots, a_{(j+1)\alpha-1})$. Multiply every ring element in s_j by $\hat{Q}_j^{-1}$ (mod the corresponding little q).

Note that I have not been able to find where Liberate-FHE multiples by $\hat{Q}_j^{-1}$.

6.1.2 ModRaise

In the notation of **??**, each state x has basis $\mathcal{B} = [(q_{j\cdot\alpha}, q_{j\cdot\alpha+1}, \ldots, q_{(j+1)\alpha-1})]$. Thus we compute $x_p = \text{ExtendBasis}(x, \mathcal{B}, p)$ for all $p \in (\mathcal{Q} \cup \mathcal{P})/\mathcal{B}$. Note that these calls are done in parallel, *not* sequentially. We create an extended state by appending each new ring element to the input, producing a new state with respect to the basis $\mathcal{Q} \cup \mathcal{P}$.

6.1.3 Dot Product

There are two details to discuss. First, this step involves multiplying ring elements, which requires the ring elements to be in their NTT form. After doing the dot product, we need to mod-down in the non-NTT form, hence we apply NTT$(\cdot)$ before computing the vector/matrix product, and iNTT$(\cdot)$ after computing the product.

The other note is that key-switching keys are with respect to the basis $\mathcal{Q} \cup \mathcal{P} \cup \{q_{\ell+1}, q_{\ell+2}, \ldots, q_L\}$. Simply discard key-switching key components corresponding to $\{q_{\ell+1}, q_{\ell+2}, \ldots, q_L\}$ before multiplying.

6.1.4 ModDown

We simply apply the rescaling algorithm (or, what Liberate calls mod_down, since the level doesn't change). Ideally, we would use nearest-integer rounding to minimize key-switch error, but Liberate-FHE uses floor-rounding.

6.2 Relinearization

Relinearization is a special form of key-switching. After multiplication, a ciphertext c has three components (d_0, d_1, d_2) corresponding to the basis $(1, s, s^2)$. Recall that key-switching outputs two ring elements k_0 and k_1. To relinearize a ciphertext, we output $(d_0 + k + 0, d_1 + k + 1)$.

6.3 Key-Switch Parameterization

6.3.1 Computation

First, some background. To ensure security, the modulus cannot exceed a bound $b(N)$, which is a function of the ring dimension N. During key switching, the modulus reaches its maximum at $Q \cdot P$, so we require $\log(Q \cdot P) \leq b(N)$. Another constraint is that we need $P > Q_j$ for each $0 \leq j < \beta$. The size of Q_j is determined by the CKKS precision parameter, and by the main key-switching parameter k, the number of special primes. Specifically, there are $\beta = \ell/k$ Q_js.

To determine the implications of the parameter k, we need to consider the computational complexity of each of these steps. Let the ciphertext level be ℓ, and the ring dimension N. Then $\beta \approx \ell/k$ and $\alpha = k$.

- Decompose requires $\mathcal{O}(\ell)$ ring operations, which is $\mathcal{O}(\ell \cdot N)$ arithmetic operations.
- ModRaise requires $\mathcal{O}(\ell^2)$ ring operations, or $\mathcal{O}(\ell^2 \cdot N)$ arithmetic operations.
- DotProduct requires $\mathcal{O}(\frac{\ell}{k}(\ell + k))$ ring operations, plus $\mathcal{O}(\frac{\ell}{k}(\ell + k))$ NTTs. The NTTs dominate, hence this step requires $\mathcal{O}(\frac{\ell}{k}(\ell + k) \cdot N \log N)$ arithmetic operations.
- ModDown requires $\mathcal{O}(\ell \cdot k)$ ring operations, or $\mathcal{O}(\ell \cdot k \cdot N)$ arithmetic operations.

Overall, this process is dominated by the number of NTTs, which is depends on k: if $k \approx \ell$, we only need to do $\approx \ell$ NTTs, while if $k = 1$, we have to do $\approx \ell^2$ NTTs. The table below estimates the number of arithmetic operations for the two extreme choices $k = 1$ and $k = \ell$. In order to do a direct comparison, we fix $\bar{\ell}$ to be the number of normal primes at level ℓ when $k = 1$. For the "$k = \ell$" parameterization, we cannot set $k = \ell = \bar{\ell}$ due to the bound on $P \cdot Q$. Instead, we set $k = \ell = \frac{\bar{\ell}}{2}$. The table expresses parallelism as $c \cdot \mathcal{O}(d)$, which means that we have do $\mathcal{O}(d)$ serial work in parallel c times.

While the "$k = \ell$" parameter choice significantly reduces the cost of key switching relative to $k = 1$, it also reduces the *utility* of the FHE scheme. Specifically, the $k = 1$ choice supports $\bar{\ell}$ sequential multiplications before bootstrapping, while "$k = \ell$" only supports $\bar{\ell}/2$ sequential multiplications before bootstrapping. In many cases, it makes sense to use a value of k somewhere between the two extremes. In practice, the best choice for k is application-specific, determined by (at least):

- The (multiplicative) depth of the circuit
- The cost of bootstrapping
- The number of non-relinearization key-switches in the application

Note that this paper describes a hybrid approach, meaning that $1 < k < L$ where k is the number of special moduli and L is the number of ciphertext moduli.

6.3.2 Error Analysis

Let d be the input to key switching. There are three kinds of error that contribute additively to the overall ciphertext error after key switching:

- Input error. This is the error of the ciphertext before key switching. This could be relevant to key switching because we don't want to do more work than necessary to make the key-switch error very small if the input error is somewhat large. However, we will ignore this for now.
- Error introduced by the key-switch keys. Ideally, this will be roughly the same size as the rounding error.
- Error introduced by rounding when dividing by P. This error is (depending on details) roughly $\frac{1}{2}s$, where s is the secret key. Note that the secret key is the same size (or perhaps smaller if we use a sparse secret key) than a fresh error term. See Section 5.3.1 for more details.

It suffices[7] to consider the magnitude of coefficients in the error term (and of ring elements multiplied into it).

$k = \ell$ In this case, there is one KSK $K = (k_0, k_1)$, where $k_0 \cdot s_2 + k_1 = P \cdot s_1 + e$. There is no decomposition step in this parameterization, so in the dot product step, $d \cdot k_0 \cdot s_2 + d \cdot k_1 = d \cdot P \cdot s_1 + d \cdot e$. d has coefficients smaller than Q, so after dividing by P, the error term is $\approx \frac{Q \cdot e}{P}$. We pick P to be slightly larger than Q, so this error is somewhat smaller than the original error e.

Concretely, Liberate's silver parameters with $k = 8$ would result in $L = 8$, and at level 1, $\ell = 7$. Then $Q \approx 2^{300}$, $P \approx 2^{480}$, and the error term is $2^{-180}e$. This is really much smaller than it needs to be, so it's wasteful; see Section 6.4.

$k = 1$ In this case, there are many KSKs $K_j = (k_{j,0}, k_{j,1})$, where $k_{j,0} \cdot s_2 + k_{j,1} = P \cdot \hat{Q}_j \cdot s_1 + e$. After decomposition, $\tilde{d} = \hat{Q}_j^{-1} \bmod Q_j$ has coefficients smaller than $Q_j = q_j$. After dot product and division by P, we get $\tilde{e} \approx \sum_{i=0}^{L} \frac{q_i \cdot e}{P}$. In Liberate, $q_0 \approx p = P \approx 2^{60}$, so this sum is $\approx e + \frac{L \cdot q \cdot e}{P}$, where $P > L \cdot q$.

Concretely, for Liberate's silver parameters with $k = 1$, we have 18 normal primes and one special prime. $\beta = 17$ at level 1, so $\tilde{e} = \frac{2^{60}e}{2^{60}} + \sum_{i=1}^{16} \frac{2^{40}e}{2^{60}} \approx e + 2^{-16} \cdot e$.

Summary Technically, $k = \ell$ results in a smaller error, but in practice, both should produce roughly the same key-switch error.

6.3.3 Implementation Complexity

I consider the difference in implementation complexity between these two parameter choices to be essentially zero. Given my observation in Section 6.4, the difference actually *is* zero: all values of k result in multiple states, which means the only difference between different choices of k is how many states we are adding together (minimum of two, maximum of many).

6.3.4 Utility Tradeoff

As noted elsewhere, *security* is determined by the size of the ciphertext modulus relative to the size of the error. Key-switch keys represent the most extreme situation: they have fresh encryption error (which is as small as it gets) and the biggest modulus ($P \cdot Q$). Put very simply, $P \cdot Q$ is upper-bounded by a function $B(N)$ of the ring dimension. Thus, for a fixed security level, the size of P (in Liberate, 60 bits times the number of special primes) determines the number of bits we can allocated to Q, which in turn determines the number of "levels" (sequential multiplications) that can be computed before bootstrapping is required.

In summary, small $k \Rightarrow$ small $P \Rightarrow$ big $Q \Rightarrow$ many levels before bootstrapping. Recall that this choice of k also results in a *quadratic* number of NTTs, which is expensive.

By contrast, $k = \ell \Rightarrow$ big $P \Rightarrow$ small $Q \Rightarrow$ fewer levels before bootstrapping. Recall that this choice of k results in a *linear* number of NTTs.

6.3.5 Summary

	$\ell = \bar{\ell}$, $k = 1$	$\ell = k = \bar{\ell}/2$
Decompose	$\bar{\ell} \cdot \mathcal{O}(N)$	$\frac{\bar{\ell}}{2} \cdot \mathcal{O}(N)$
ModRaise	$\bar{\ell}^2 \cdot \mathcal{O}(N)$	$\frac{\bar{\ell}}{2} \cdot \mathcal{O}(\frac{\bar{\ell}}{2} \cdot N)$
DotProduct	$\bar{\ell}^2 \cdot \mathcal{O}(N \log N)$	$\bar{\ell} \cdot \mathcal{O}(N \log N)$
ModDown	$\bar{\ell} \cdot \mathcal{O}(N)$	$\frac{\bar{\ell}}{2} \cdot \mathcal{O}(N)$
Key-Switch Error	$\approx$ fresh error	$\lesssim$ fresh error
Impl Complexity	same	same
Utility	many levels$\Rightarrow$ infrequent bootstrapping	few levels$\Rightarrow$ more frequent bootstrapping

[7] According to Craig. Technically, we should be looking at the norm of the error term under the canonical embedding, but Craig says this is an isometry, so we can get an idea of relative error by looking at the coefficients.

6.4 Key Switching Suggestions

6.4.1 Don't isolate q_0

Liberate is *always* putting the q_0 CRT component into its own state, regardless of the number of special primes.[8] When $k = 2$, this isn't a big deal: some states have one prime, and some states have two depending on the level of the input. When $k > 2$, this becomes more unbalanced. For example, if $k = 4$, most states will correspond to four primes, while one state will always correspond to (only) q_0. In the extreme case of $k = \ell$, (e.g., $k = \ell = 8$), there *should* only be one state corresponding to $q_0 \cdot \ldots \cdot q_7$, but Liberate insists on putting q_0 into its own state, so there are now two states: one with q_0 and one with $q_1 \cdot \ldots \cdot q_7$. This means in Liberate's implementation, there is no computational advantage of using $k = \ell$ over $k = \ell/2$. From a utility perspective, this is also wasteful: we only *need* seven special primes (or maybe even less, depending on Δ), but we're using eight, resulting in more work and a smaller-than-necessary key-switch error.

6.4.2 Only extend with $\max(Q_j)$ special primes

As noted above, states (corresponding to Q_j) don't always correspond to the same number of special primes. In general, we only need $P \gtrsim \max_j(Q_j)$. Again, this isn't such a big deal when $k = 2$ because unless we're at the highest level, some Q_j will correspond to two primes, and we always basis-extend with both special primes. However, when $k = \ell$ (independent of the issue above), we need seven p_js for level 0 key-switching, six for level-1, and so on. But Liberate is *always* basis-extending with all eight special primes. This is computationally wasteful, and results in noise that is much smaller than necessary.

References

[Bad+18] Ahmad Al Badawi et al. *Implementation and Performance Evaluation of RNS Variants of the BFV Homomorphic Encryption Scheme*. Cryptology ePrint Archive, Paper 2018/589. 2018. DOI: 10.1109/TETC.2019.2902799. URL: https://eprint.iacr.org/2018/589.

[Baj+16] Jean-Claude Bajard et al. *A Full RNS Variant of FV like Somewhat Homomorphic Encryption Schemes*. Cryptology ePrint Archive, Paper 2016/510. 2016. URL: https://eprint.iacr.org/2016/510.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. *Efficient Fully Homomorphic Encryption from (Standard) LWE*. Cryptology ePrint Archive, Paper 2011/344. 2011. URL: https://eprint.iacr.org/2011/344.

[CCS18] Hao Chen, Ilaria Chillotti, and Yongsoo Song. *Improved Bootstrapping for Approximate Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2018/1043. 2018. URL: https://eprint.iacr.org/2018/1043.

[Che+16] Jung Hee Cheon et al. *Homomorphic Encryption for Arithmetic of Approximate Numbers*. Cryptology ePrint Archive, Paper 2016/421. 2016. URL: https://eprint.iacr.org/2016/421.

[Che+18a] Jung Hee Cheon et al. *A Full RNS Variant of Approximate Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2018/931. 2018. URL: https://eprint.iacr.org/2018/931.

[Che+18b] Jung Hee Cheon et al. *Bootstrapping for Approximate Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2018/153. 2018. URL: https://eprint.iacr.org/2018/153.

[CP15] Eric Crockett and Chris Peikert. *$\Lambda \circ \lambda$: Functional Lattice Cryptography*. Cryptology ePrint Archive, Paper 2015/1134. 2015. DOI: 10.1145/2976749.2978402. URL: https://eprint.iacr.org/2015/1134.

[GCL92] K.O. Geddes, S.R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Springer US, 1992. ISBN: 9780792392590. URL: https://books.google.com/books?id=B9tC7DOX_oUC.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. *Homomorphic Evaluation of the AES Circuit*. Cryptology ePrint Archive, Paper 2012/099. 2012. URL: https://eprint.iacr.org/2012/099.

[HK19] Kyoohyung Han and Dohyeong Ki. *Better Bootstrapping for Approximate Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2019/688. 2019. URL: https://eprint.iacr.org/2019/688.

[HPS18] Shai Halevi, Yuriy Polyakov, and Victor Shoup. *An Improved RNS Variant of the BFV Homomorphic Encryption Scheme*. Cryptology ePrint Archive, Paper 2018/117. 2018. DOI: 10.1007/978-3-030-12612-4_5. URL: https://eprint.iacr.org/2018/117.

[KPP20] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. *Approximate Homomorphic Encryption with Reduced Approximation Error*. Cryptology ePrint Archive, Paper 2020/1118. 2020. URL: https://eprint.iacr.org/2020/1118.

[QX24] Hongyuan Qu and Guangwu Xu. *An improved exact CRR basis conversion algorithm for FHE without floating-point arithmetic*. Cryptology ePrint Archive, Paper 2024/417. 2024. URL: https://eprint.iacr.org/2024/417.

[8] Independent of the issue raised here, there doesn't seem to be any mathematical reason to do this.

A Montgomery Arithmetic

The method by which an implementation performs modular arithmetic is orthogonal to the details of FHE, but we explain the details here.

First, let N be an (odd) modulus, and $R = 2^k > 2N$. Since N is odd and R is a power of two, $\gcd(R, N) = 1$. We want $R > 2N$ because Montgomery reduction outputs a value in the range $[0, 2N)$.

We will need to calculate $N' = -N^{-1} \bmod R$. While this can be calculated directly, you may see a different formula for N', namely

$$N' = \frac{R \cdot (R^{-1} \bmod N) - 1}{N}$$

Lemma A.1. $-N^{-1} \bmod R = \frac{R \cdot (R^{-1} \bmod N) - 1}{N}$

Proof. By definition, $R \cdot (R^{-1} \bmod N) \equiv 1 \bmod N$, hence $R \cdot (R^{-1} \bmod N) = 1 + k \cdot N$ over the integers. Solving for k, we find $k = \frac{R \cdot (R^{-1} \bmod N) - 1}{N}$. Hence we aim to show that $k = -N^{-1} \bmod R$.

Look at $R \cdot (R^{-1} \bmod N) = 1 + k \cdot N$. Mod R, this becomes $0 \equiv 1 + k \cdot N \bmod R$, hence $k \equiv -N^{-1} \bmod R$, as desired. □

A.1 Limits

The following section assumes a word size of 64 bits, but generalizes to any word size.

We perform multiplication of two 64-bit numbers a and b by splitting them into two 32-bit pieces: $a = 2^{32} \cdot a_h + a_\ell$ and $b = 2^{32} \cdot b_h + b_\ell$. We represent all four of these pieces as 64-bit numbers, then multiply the pieces appropriately to get the three 64-bit components of the result: $c = 2^{64} \cdot a_h \cdot b_h + 2^{32} \cdot (a_h \cdot b_\ell + a_\ell \cdot b_h) + a_\ell \cdot b_\ell$. Note that all four products are smaller than 64 bits, however the middle word needs an extra bit since we sum two 64-bit values. This means R cannot be larger than 2^{63}, otherwise the computation of this middle term could overflow.

Since we need to be able to subtract numbers in Montgomery form, it is useful to use a signed data type. This further restricts $R \leq 2^{62}$.

Algorithm 2 Montgomery Reduction in Liberate

```
import numpy as np
nbits ← 62
R ← 1 ≪ nbits
half_nbits ← 31
fb_mask ← np.int64((1 ≪ nbits) - 1)
lb_mask ← np.int64((1 ≪ half_nbits) - 1)
k ← −inverse_mod(N, R)
procedure PARTITIONINT64(x)
    return x & lb_mask, x ≫ half_nbits
N_ℓ, N_h ← partitionInt64(N)
k_ℓ, k_h ← partitionInt64(k)
procedure MONTREDC(a, b)
                                              ▷ Compute T = a · b
    a_ℓ, a_h ← partitionInt64(a)
    b_ℓ, b_h ← partitionInt64(b)
    α ← ah · bh
    β ← ah · bl + al · bh
    γ ← al · bl
                                              ▷ Compute m = T · k mod R
    γ_ℓ, γ_h ← partitionInt64(γ)
    β_ℓ, β_h ← partitionInt64(β)
    u ← γ_ℓ · k_h + (γ_h + β_ℓ) · k_ℓ
    m ← ((u ≪ half_nbits) + γ_ℓ · k_ℓ) & fb_mask
                                              ▷ Compute t = (T + m · N)/R
    t_ℓ, t_h ← partitionInt64(t)
    tNb ← t_h · N_ℓ + s_ℓ · N_h               ▷ Middle word of t · N
    tNb_ℓ, tNb_h ← partitionInt64(tNb)
    carry ← (γ + t_ℓ · N_ℓ) ≫ half_nbits
    carry ← (carry + β_ℓ + tNb_ℓ) ≫ half_nbits
    t ← α + β_h + tNb_h + carry + t_h · N_h
    return t
```

First, let's look at the math. Montgomery reduction computes

$$\begin{aligned} m &= (T \bmod R) \cdot k \bmod R \\ t &= (T + m \cdot N)/R \end{aligned}$$

We assume that $0 \leq T < R \cdot N$. Then $t \in [0, 2N)$. This is easy to see by looking at the numerator of t: $T \leq R \cdot N$ by assumption, and $0 \leq m < R$, hence $T + m \cdot N < 2R \cdot N$, hence $t \leq 2N$.

We note that $N < R$, so if we multiply two mod-N integers, their product is $< N^2 < R \cdot N$. If $2N < R$ and one input to the product is in the range $[0, 2N)$, the product is still $< N \cdot R$ so reduction still has the

same output range. Similarly, if $4N < R$, then both multiplicands can be in $[0, 2N)$. This is the case with Liberate parameters: they chose their large primes to be a hair under 60 bits, and $R = 2^{62}$, hence reduction to $[0, 2N)$ suffices for the multiplication inputs.

We are also interested in what happens on negative inputs. First, we sketch a proof that the algorithm is correct on negative inputs (though the output range is different). First, we assume that the implementation language uses sign extension for right shifts. This means that the high word output of partitionInt64 will have 33 1s in the top bits, followed by 31 bits of signal in the low bits. However, the invariant for partitionInt64(x) still holds: $x = 2^{31} \cdot x_h + x_\ell$. This means that the product of a and b also holds: $a \cdot b = 2^{62} \cdot \alpha + 2^{31} \cdot \beta + \gamma$. Next, we note that the two's-complement representation of integers means that $T \bmod R = T \;\&$ fb_mask, whether T is positive or negative. In short, this means that Algorithm 2 does something meaningful on negative inputs. For instance, if both multiplicands are in the range $[-N, N)$, T is in the range $[-N^2, N^2) \subseteq [-R \cdot N, R \cdot N)$, hence $(T + m \cdot N)/R \in [-N, 2N)$. This means we could maintain a centered representation (with double range) with one conditional subtraction. Note this is slightly *worse* than the standard $[0, 2N)$ range, since we can maintain that expanded range with *no* conditional subtractions.

Overall, Liberate's use of Montgomery arithmetic is inconsistent. For example, rescale uses PyTorch tensor subtraction, while mod_down uses Montgomery subtraction (even though they first do a modular reduction in mod_down, so there's actually *more* reason do use Montgomery subtraction in rescale than mod_down).

Keeping track of output ranges is tedious in the extreme; I strongly recommend that an implementation consistently reduces to the standard representative unless you can are absolutely sure that not doing so will still produce a valid result.

B Unification of Rescaling Notions

Let $x \in \mathbb{Z}_{Q_\ell}$. Section 5.1 gives three different definitions of rescaling x:

1. $\mathrm{RS}(x) = \lfloor q_\ell^{-1} \cdot x \rceil \bmod Q_{\ell-1}$ from [Che+16; Che+18b; CCS18; KPP20]
2. $\mathrm{RS}(x) = q_\ell^{-1} \cdot (x - x_\ell) \bmod Q_{\ell-1}$ from [Che+18a; HK19]
3. $\mathrm{RS}(x) = \left[\left\lfloor \sum_{i=1}^{k} x_i \cdot \left(\tilde{q}_i \cdot \frac{t}{q_i}\right)\right\rceil\right]_t$ from Section 5.1.2, which is adapted from [HPS18]

In this section, we show that these three definitions are equivalent.

B.1 Equivalence of (1) and (2)

[CP15] describes why this process works. In short, we want to compute $\left[\left\lfloor \frac{1}{q_\ell} \cdot x \right\rceil\right]_{Q_{\ell-1}}$ where $x \in \mathbb{Z}_{Q_\ell}$. Forget the final mod for a moment and just look at $\frac{1}{q_\ell} \cdot x$. Write $\frac{1}{q_\ell} \cdot x = \omega + \theta$ where $\omega \in \mathbb{Z}$ and $\theta \in [-1/2, 1/2)$. $x \equiv x_\ell \mod q_\ell$, so $\omega = \frac{x - x_\ell}{q_\ell}$ and $\theta = \frac{x_\ell}{q_\ell}$. Note that since the canonical representative $x_\ell = [x]_{q_\ell}$ is symmetric around 0, $\omega = \left\lfloor \frac{1}{q_\ell} \cdot x \right\rceil \in \mathbb{Z}$. Finally,

$$\left[\left\lfloor \frac{1}{q_\ell} \cdot x \right\rceil\right]_{Q_{\ell-1}} = \left[\frac{x - x_\ell}{q_\ell}\right]_{Q_{\ell-q}} = [q_\ell^{-1} \cdot (x - x_\ell)]_{Q_{\ell-q}}$$

Now we extend this to $x = (x_1, x_2, \ldots, x_\ell)$ in RNS form:

$$\left[\left\lfloor \frac{1}{q_\ell} \cdot (x_1, x_2, \ldots, x_\ell) \right\rceil\right]_{Q_{\ell-1}} = \left[\frac{(x_1 - x_\ell, x_2 - x_\ell, \ldots, x_\ell - x_\ell)}{q_\ell}\right]_{Q_{\ell-q}} \tag{1}$$

$$= (q_\ell^{-1} \cdot (x_1 - x_\ell), q_\ell^{-1} \cdot (x_x - x_\ell), \ldots, q_\ell^{-1} \cdot (x_{\ell-1} - x_\ell)) \in Z_{Q_{\ell-q}} \tag{2}$$

B.2 Equivalence of (2) and (3)

In the context of rescaling, $q = Q_\ell$ and $t = Q_{\ell-1}$, so $t/q = 1/q_\ell$. We rewrite the equation with these substitutions:

$$\mathrm{RS}(x) = \left[\left\lfloor \sum_{i=1}^{\ell} x_i \cdot \left(\tilde{q}_i \cdot \frac{Q_{\ell-1}}{q_i}\right)\right\rceil\right]_{Q_{\ell-1}}$$

For simplicity, we will compute the output mod $Q_{\ell-1}$ in RNS form, so we really care about the rounded value mod q_i, $1 \le i < \ell$. The j^{th} component of $\mathrm{RS}(x)$ is

$$\left[\left\lfloor \sum_{i=1}^{\ell} x_i \cdot \left(\tilde{q}_i \cdot \frac{Q_{\ell-1}}{q_i}\right)\right\rceil\right]_{q_j}$$

First, note that we can pull the first $\ell - 1$ terms out of the rounding function because $q_i | Q_{\ell-1}$ for $1 \le i \le \ell - 1$. Thus we only need to compute the integral part of last term.

$$\left\lfloor x_\ell \cdot \left(\tilde{q}_\ell \cdot \frac{Q_{\ell-1}}{q_\ell}\right)\right\rceil = x_\ell \cdot (\tilde{q}_\ell \cdot Q_{\ell-1} - [\tilde{q}_\ell \cdot Q_{\ell-1}]_{q_\ell})/q_\ell \tag{3}$$

$$= x_\ell \cdot (\tilde{q}_\ell \cdot Q_{\ell-1} - 1)/q_\ell \tag{4}$$

Therefore, the j^{th} component of $\text{RS}(x)$ is

$$\left[\left(\sum_{i=1}^{\ell-1} x_i \cdot \tilde{q}_i \cdot \frac{Q_{\ell-1}}{q_i}\right) + x_\ell \cdot (\tilde{q}_\ell \cdot Q_{\ell-1} - 1)/q_\ell\right]_{q_j} = \left[x_j \cdot \tilde{q}_j \cdot \frac{Q_{\ell-1}}{q_j} + x_\ell \cdot (\tilde{q}_\ell \cdot Q_{\ell-1} - 1)/q_\ell\right]_{q_j} \tag{5}$$

$$= \left[x_j \cdot (q_\ell^{-1} \bmod q_j) + x_\ell \cdot (\tilde{q}_\ell \cdot Q_{\ell-1} - 1)/q_\ell\right]_{q_j} \tag{6}$$

$$= \left[x_j \cdot (q_\ell^{-1} \bmod q_j) + x_\ell \cdot (-q_\ell^{-1} \bmod q_j)\right]_{q_j} \tag{7}$$

$$= \left[(x_j - x_\ell) \cdot (q_\ell^{-1} \bmod q_j)\right]_{q_j} \tag{8}$$

where the Eq. (7) follows from the fact that $\tilde{q}_\ell \cdot Q_{\ell-1}$ is a multiple of q_j.

B.3 Comparison

We emphasize that Section 5.1.2 is actually more general; in particular it works when $t \nmid q$.

Formulation 2 requires one modular addition and one (scalar) modular multiplication per output RNS component. If using standard representatives, there is an addition comparison and addition.

Assuming we precompute $\omega_{i,j}$ and $\theta_{i,j}$ for each output component t_j, Formulation 3 requires ℓ (scalar) modular multiplications and additions (ignoring the shared cost of computing v). Thus we should prefer formulation 2 when possible.

C Key Switching Notes

C.1 Generating Keys

To generate a key-switch key for switching from key s_1 to key s_2:

1. Generate $a^{(i)} \leftarrow U(R_{PQ})$ for $0 \le i < \beta$
2. Generate $e^{(i)} \leftarrow \chi$ for $0 \le i < \beta$
3. Compute $\text{swk}^{(i)} = (b^{(i)}, a^{(i)}) \in R_{PQ}^2$, where $b^{(i)} = -a^{(i)} \cdot s_2 + P \cdot [\hat{Q}_i]_{q_j} \cdot s_1 + e^{(i)}$ Note that when we view this in RNS form, the term $P \cdot [\hat{Q}_i]_{q_j} \cdot s_1 \cong 0 \bmod p_i$. Note: we can
4. Output $\{\text{swk}^{(i)}\}_{0 \le i < \beta}$, which we can view as a $\beta \times 2$ matrix K, where each element is in R_{PQ}.

In the non-RNS version, key switch output $P \cdot s_1$ plus an encryption of zero; here we (roughly) output P times a component of $\text{RNS}\text{-}\text{Power}_{\mathcal{C}'}(s_1)$, plus a (fresh) encryption of zero.

C.2 Other Approaches

[HK19] does not appear to be the final word on CKKS key switching; this section describes differences in later papers. Eric: TODO

Eric!

C.2.1 Approximate Homomorphic Encryption with Reduced Approximation Error

[KPP20] describes a different approach for $\text{RNS}\text{-}\text{Decomp}_{\mathcal{C}}$ and $\text{RNS}\text{-}\text{Power}_{\mathcal{C}}$ based on a base-ω decomposition. However, in section 2.3 it says that when using RNS, we should use RNS-digit-decomposition instead of base-ω decomposition. I confirmed that the decompositions are identical.

KSGen uses slightly different notation, but is identical.