# An improved exact CRR basis conversion algorithm for FHE without floating-point arithmetic

Hongyuan Qu, Guangwu Xu

**Abstract.** Fully homomorphic encryption (FHE) has attracted much attention recently. Chinese remainder representation (CRR) or RNS representation is one of the core technologies of FHE. CRR basis conversion is a key step of KeySwitching procedure. Bajard et al. proposed a fast basis conversion method for CRR basis conversion, but the elimination of error had to be ignored. Halevi et al. suggested a method using floating-point arithmetic to avoid errors, but floating-point arithmetic has its own issues such as low efficiency and complex chip design. In this work, we establish a more concise and efficient CRR basis conversion method by observing that each of the ciphertext modulus selected by the CRR CKKS scheme is very close to an integer that is a power of 2. Our conversion algorithm eliminates errors and involves only integer arithmetic and bit operations. The proof of correctness of our algorithm is given. Extensive experiments are conducted and comparisons between the method of Halevi et al. and ours are obtained, which show that our method has the same accuracy and a slightly better effeciency. Our method is also applicable to the CRR variant of BGV and BFV schemes, and can be used to simplify chip design.

**Keywords:** Fully homomorphic encryption· CRR basis conversion· Floating-point arithmetic· Error elimination.

## 1 Introduction

Since Gentry proposed the first homomorphic encryption scheme[12], the field of homomorphic encryption has been developed rapidly. Fully homomorphic encryption allows any secure computation of the encrypted ciphertexts without the need for decryption. At present, it has been extremely important in many fields.

At present, the mainstream wordbased fully homomorphic encryption schemes include BGV[6], BFV[5, 11, 4], CKKS[9, 7], etc.. These schemes all use packing technology to realize the component-wise homomorphic calculation of data vectors[20]. One of the main advantages of CKKS scheme is that it supports homomorphic calculation of complex vectors, so it has a wide range of applications. There are many open source homomorphic encryption libraries, such as Helib[15], SEAL[2], PALISADE[1], HEAAN[8], implementing one or more fully homomorphic encryption schemes, as well as their CRR variants. The CRR variants of these schemes make use of the Chinese Remainder Theorem to decompose a large integer into many small integers, and decompose the operation of large integers into many small integer operations, thus greatly speeding up the calculations. In order to improve the efficiency or reduce the error of fully homomorphic encryption, many optimization methods have been proposed. In 2012, in order to reduce the error of the KeySwitching procedure, Gentry et al. introduced an extra large module $P$, which replaced the previously used method of bit decomposition, thus reducing the complexity of the calculation. Their approach is called GHS optimization[13]. Han et al. developed a hybrid KeySwitching method in 2020, which combines the way of GHS optimization and the idea of CRR decomposition to reduce the bit length of $P$, thus allowing more homomorphic calculations[16]. In 2021, Kim et al. proposed an exact rescaling method to solve the problem of large error in CRR CKKS rescaling process. They proposed a new mode of selecting ciphertext modulus, which ensures that the error generated in rescaling process is greatly reduced[18].

In the CRR variant of the above schemes, the KeySwitching procedure differs significantly from the original scheme. When GHS or hybrid optimization is adopted, it involves the representation conversion of ciphertext polynomials between two coprime CRR basis, which is the core operation of KeySwitching. Barjard et al. proposed a method called fast basis conversion to convert representations of polynomials from one CRR basis to another. This method is also applied to the CRR variant of the CKKS scheme. However, this method cannot eliminate the errors in the conversion process[4]. In 2019, Halevi et al. proposed a method that uses floating-point arithmetic to eliminate the errors in the CRR basis conversion process, and it is the fastest accurate method currently[14]. However, floating-point arithmetic has many disadvantages such as long operation time and complex chip design[19, 10, 17].

### 1.1 Our contributions

We propose a method to eliminate errors in CRR basis conversion procedure without the need for floating-point arithmetic. Our approach is based on the observation that each small prime modulus is very close to an integer $q$ that is a power of 2. Using this observation we only need to use integer addition and subtraction and bit operations to calculate the error term, thus replacing the floating-point operations in Halevi et al. 's method. We prove the correctness of our method. By selecting the encryption parameters reasonably, the error probability is negligible. Even if there is an error, the error is reduced compared with the original scheme. We applied our method to the CRR CKKS scheme. After experimental verification, our method obtains the same accuracy as Halevi et al. 's method, with an improvement of $0 - 0.5$ bits compared with the original scheme, and the running time difference is less than 10ms from the original scheme. We mention that our method has good theoretical significance and can be used to simplify the design of homomorphic encryption chips. Our method is also applicable to CRR BGV and CRR BFV schemes if they use the same modular selection method.

### 1.2 organizations

Section 2 provides the necessary background about CRR basis conversion methods and CRR CKKS scheme. Section 3 describes our method for faster error calculation. Section 4 describes our modification of the CRR CKKS scheme and complexity analysis of our method, and section 5 describes our experimental results.

## 2 Preliminaries

All logarithmic operations are in base 2 unless otherwise specified. For an integer $Q$, we use $[-Q/2, Q/2) \cap \mathbb{Z}$ as a representation interval of $\mathbb{Z}_Q$, and use $[x]_Q$ to represent the reduction of the integer $x$ modulo $Q$ into the interval. For an integer $N$ that is a power of 2, we denote $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$, $\mathcal{S} = \mathbb{R}[X]/(X^N + 1)$, $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R}$. A finite ordered set $\mathcal{C} = \{q_0, q_1, \ldots, q_{\ell-1}\}$ is called a CRR basis if its elements are coprime to each other. We denote $Q = \prod_{i=0}^{\ell-1} q_i$, $\hat{q}_i = Q/q_i$, $\hat{q}_i^{-1} = 1/\hat{q}_i \pmod{q_i}$.

For a polynomial $a$, we use $a \leftarrow U(S)$ to denote that $a$ is sampled uniformly at random in the set $S$. We use $a \leftarrow \chi$ to indicate that $a$ is sampled according to the distribution $\chi$. We use $\chi_{key}, \chi_{err}, \chi_{enc}$ to represent the distribution used during private key generation, error generation and encryption, respectively. Ternary distribution is commonly used in $\chi_{key}$, which means that all the coefficients of $a$ are selected uniformly from $\{-1, 0, 1\}$. This distribution is the most efficient option recommended by homomorphic encryption standard[3]. Discrete Gaussian distribution is commonly used as $\chi_{err}$ to ensure security.

### 2.1 Chinese remainder representation (CRR)

For a CRR basis $\mathcal{C} = \{q_0, q_1, \ldots, q_{\ell-1}\}$, $Q = \prod_{i=0}^{\ell-1} q_i$, according to Chinese Remainder Theorem, for any $x \in \mathbb{Z}_Q$, $x$ can be uniquely represented by the so called Chinese remainder representation (CRR) or RNS representation in the basis $\mathcal{C}$, denoted as $[x]_\mathcal{C} = ([x]_{q_0}, [x]_{q_1}, \ldots, [x]_{q_{\ell-1}})$. And $x$ satisfies

$$x = \sum_{i=0}^{\ell-1} [x]_{q_i} \cdot \hat{q}_i \cdot \hat{q}_i^{-1} - v \cdot Q,$$

where $v \in \mathbb{Z}$. Or

$$x = \sum_{i=0}^{\ell-1} [[x]_{q_i} \cdot \hat{q}_i^{-1}]_{q_i} \cdot \hat{q}_i - e \cdot Q,$$

where $e \in [-\ell/2, \ell/2) \cap \mathbb{Z}$. For a polynomial $a \in \mathcal{R}_Q$, its Chinese remainder representation, denoted as $[a]_\mathcal{C}$, is $([a]_{q_0}, \ldots, [a]_{q_{\ell-1}})$, where $[a]_{q_i}$ denotes the polynomial obtained by $a$ modulo $q_i$ for each of its coefficients.

### 2.2 CRR Basis Conversion

CRR basis conversion is a core operation of the KeySwitching procedure in CRR CKKS scheme. The original scheme uses the fast basis conversion method to convert the representation of a polynomial into a new basis that is coprime to the original basis. Specifically, for a CRR basis $\mathcal{D} = \{p_0, \dots, p_{k-1}, q_0, \dots, q_{\ell-1}\}$, let $\mathcal{B} = \{p_0, \dots, p_{k-1}\}$ and $\mathcal{C} = \{q_0, \dots, q_{\ell-1}\}$ be two sub bases of $\mathcal{D}$, and let $P = \prod_{i=0}^{k-1} p_i$, $Q = \prod_{j=0}^{\ell-1} q_j$. Then one can convert the CRR $[x]_\mathcal{C} = ([x]_{q_0}, \dots, [x]_{q_{\ell-1}}) \in \mathbb{Z}_{q_0} \times \cdots \times \mathbb{Z}_{q_{\ell-1}}$ of an integer $x \in \mathbb{Z}_Q$ into an element of $\mathbb{Z}_{p_0} \times \cdots \times \mathbb{Z}_{p_{k-1}}$ by computing

$$\mathrm{Conv}_{\mathcal{C}\to\mathcal{B}}([x]_\mathcal{C}) = \left(\sum_{j=0}^{\ell-1} [[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j} \cdot \hat{q}_j \pmod{p_i}\right)_{0\le i<k}.$$

We note that the result above is actually $\mathrm{Conv}_\mathcal{C} = [x + Q \cdot e]_\mathcal{B}$, where $e \in [-\ell/2, \ell/2) \cap \mathbb{Z}$.

### 2.3 Exact CRR basis conversion

Halevi et al. proposed a method to calculate the above $e$ using floating-point arithmetic, which can eliminate the error of CRR basis conversion[14]. Specifically,

$$e = \left\lceil \left( \sum_{j=0}^{\ell-1} [[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j} \cdot \hat{q}_j \right) / Q \right\rfloor = \left\lceil \sum_{j=0}^{\ell-1} [[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j} \cdot \frac{\hat{q}_j}{Q} \right\rfloor = \left\lceil \sum_{j=0}^{\ell-1} \frac{[[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j}}{q_j} \right\rfloor.$$

Therefore, we first calculate $y_j := [[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j},\ j = 0, \dots, \ell-1$, then we compute rational numbers $z_j := y_j/q_j,\ j = 0, \dots, \ell-1$. Then we sum up all the $z_j$'s and round it to get $e$. And finally we calculate $[x]_{p_i} = \left[\sum_{j=0}^{\ell-1} y_j \cdot [\hat{q}_j]_{p_i} - e \cdot [Q]_{p_i}\right]_{p_i}$ for $i = 0, \dots, k-1$.

### 2.4 CRR CKKS scheme

All operations of the CRR CKKS scheme are performed under CRR. Plaintext space is $\mathcal{R}$. We let $M = 2N$ and $\mathbb{Z}_M^* = \{x \in \mathbb{Z}_M : \gcd(x, M) = 1\}$ be the multiplication group composed of elements that are coprime with $M$. The canonical embedding $\sigma : \mathcal{S} \to \mathbb{C}^N$ is defined as $\sigma(a) = (a(\zeta^j))_{j\in\mathbb{Z}_M^*}$, where $\zeta = \exp(2\pi i/M)$. We also define natural projection $\tau : \mathbb{C}^N \to \mathbb{C}^{N/2}$ used in encoding and decoding procedure in the CRR CKKS scheme. The main processes of the scheme is as follows.

- **Setup**$(q, L, \eta, 1^\lambda)$**.** For a base integer $q = 2^m$ and an integer $L$, given the security parameter $\lambda$, choose a power-of-two $N$, and $\chi_{key}, \chi_{err}, \chi_{enc}$ for $\lambda$-bit of security. Then choose a basis $\mathcal{D} = \{q_0, \dots, q_{L-1}, p_0, \dots, p_{k-1}\}$ such that $q/q_j \in [1 - 2^{-\eta}, 1 + 2^{-\eta}]$ for $0 \le j < L$. Let $\mathcal{B} = \{p_0, \dots, p_{k-1}\}$,

$\mathcal{C}_\ell = \{q_0, \ldots, q_{\ell-1}\}$ and $\mathcal{D}_\ell = \{q_0, \ldots, q_{\ell-1}, p_0, \ldots, p_{k-1}\}$. Let $P = \prod_{i=0}^{k-1} p_i$ and $Q_\ell = \prod_{j=0}^{\ell-1} q_j$ for $1 \le \ell \le L$. Finally, perform some necessary precomputation.

- **KeyGen.** First sample secret $s \leftarrow \chi_{key}$, $(a^{(0)}, \ldots, a^{(L-1)}) \leftarrow U\left(\prod_{j=0}^{L-1} \mathcal{R}_{q_j}\right)$, $e \leftarrow \chi_{err}$. Set the secret key as $\boldsymbol{sk} \leftarrow (1, s)$ and public key as $\boldsymbol{pk} \leftarrow \left(\boldsymbol{pk}^{(j)} = (b^{(j)}, a^{(j)}) \in \mathcal{R}_{q_j}^2\right)_{0 \le j < L}$, where $b^{(j)} \leftarrow -a^{(j)} \cdot s + e \pmod{q_j}$.
- **KeySwitchGen$_{\boldsymbol{sk}}(s')$.** Sample $(a^{'(0)}, \ldots, a^{'(L+k-1)}) \leftarrow U\left(\prod_{j=0}^{L-1} \mathcal{R}_{q_j} \times \prod_{i=0}^{k-1} \mathcal{R}_{p_i}\right)$ and an error $e' \leftarrow \chi_{err}$, Set the switching key $\boldsymbol{swk}$ as

  $$\left(\boldsymbol{swk}^{(0)} = (b^{'(0)}, a^{'(0)}), \ldots, \boldsymbol{swk}^{(L+k-1)} = (b^{'(L+k-1)}, a^{'(L+k-1)})\right) \in \prod_{j=0}^{L-1} \mathcal{R}_{q_j}^2 \times \prod_{i=0}^{k-1} \mathcal{R}_{p_i}^2$$

  where $b^{'(j)} \leftarrow -a^{'(j)} \cdot s + [P]_{q_j} \cdot s' + e' \pmod{q_j}$ for $0 \le j < L$ and $b^{'(L+i)} \leftarrow -a^{'(L+i)} \cdot s + e' \pmod{p_i}$ for $0 \le i < k$.
- **Encode$(\boldsymbol{x})$.** For a vector $\boldsymbol{x} \in \mathbb{C}^{N/2}$, output $\lceil \sigma^{-1} \circ \tau^{-1}(q \cdot \boldsymbol{x}) \rfloor \in \mathcal{R}$.
- **Decode$(m)$.** For a plaintext $m \in \mathcal{R}$, output $\tau \circ \sigma(m) \in \mathbb{C}^{N/2}$.
- **Enc$_{\boldsymbol{pk}}(m)$.** For $m \in \mathcal{R}$, sample $r \leftarrow \chi_{enc}$, $e_0, e_1 \leftarrow \chi_{err}$, output ciphertext $\boldsymbol{ct} = (\boldsymbol{ct}^{(j)})_{0 \le j < L} \in \prod_{j=0}^{L-1} \mathcal{R}_{q_j}^2$ where $\boldsymbol{ct}^{(j)} \leftarrow r \cdot \boldsymbol{pk}^{(j)} + (m + e_0, e_1) \pmod{q_j}$ for $0 \le j < L$.
- **Dec$_{\boldsymbol{sk}}(\boldsymbol{ct})$.** For a ciphertext $\boldsymbol{ct} = (\boldsymbol{ct}^{(j)}_{0 \le j \le \ell-1})$, output $\langle \boldsymbol{ct}^{(0)}, \boldsymbol{sk} \rangle \pmod{q_0}$.
- **Add$(\boldsymbol{ct}, \boldsymbol{ct'})$.** For two ciphertexts $\boldsymbol{ct} = (\boldsymbol{ct}^{(j)})_{0 \le j < \ell}$, $\boldsymbol{ct'} = (\boldsymbol{ct'}^{(j)})_{0 \le j < \ell}$, output $\boldsymbol{ct}_{\text{add}} = (\boldsymbol{ct}_{\text{add}}^{(j)})_{0 \le j < \ell}$ where $\boldsymbol{ct}_{\text{add}}^{(j)} \leftarrow \boldsymbol{ct}^{(j)} + \boldsymbol{ct'}^{(j)} \pmod{q_j}$ for $0 \le j < \ell$.
- **KeySwitch$_{\boldsymbol{swk}}(\boldsymbol{ct})$.** The two core operations used in this stage are

  $$\text{ModUp}_{\mathcal{C}_\ell \to \mathcal{D}_\ell}(\cdot) : \prod_{j=0}^{\ell-1} \mathcal{R}_{q_j} \to \prod_{j=0}^{\ell-1} \mathcal{R}_{q_j} \times \prod_{i=0}^{k-1} \mathcal{R}_{p_i},$$
  $$\text{ModDown}_{\mathcal{D}_\ell \to \mathcal{C}_\ell}(\cdot) : \prod_{j=0}^{\ell-1} \mathcal{R}_{q_j} \times \prod_{i=0}^{k-1} \mathcal{R}_{p_i} \to \prod_{j=0}^{\ell-1} \mathcal{R}_{q_j},$$

  where $\text{ModUp}_{\mathcal{C}_\ell \to \mathcal{D}_\ell}([a]_{\mathcal{C}_\ell})$ uses $\text{Conv}_{\mathcal{C}_\ell \to \mathcal{B}}$ coefficient-wisely to convert the CRR of the polynomial $a$ under basis $\mathcal{C}_\ell$ to the CRR under basis $\mathcal{D}_\ell$. The functionality of $\text{ModDown}_{\mathcal{D}_\ell \to \mathcal{C}_\ell}([a]_{\mathcal{D}_\ell})$ is to calculate $\lceil \frac{a}{P} \rfloor$, during which $\text{Conv}_{\mathcal{B} \to \mathcal{C}_\ell}$ is used coefficient-wisely to compute the CRR of $a \pmod{P}$ under basis $\mathcal{C}_\ell$.

  The process is as follows. For a ciphertext $\boldsymbol{ct} = (\boldsymbol{ct}^{(j)})_{0 \le j < \ell}$, where $\boldsymbol{ct}^{(j)} = (ct_0^{(j)}, ct_1^{(j)}) \in \mathcal{R}_{q_j}^2$, first compute

  $$\tilde{\boldsymbol{ct}}_1 \leftarrow \text{ModUp}_{\mathcal{C}_\ell \to \mathcal{D}_\ell}(ct_1^{(0)}, \ldots, ct_1^{(\ell-1)}).$$

  Then compute

  $$\tilde{\boldsymbol{ct}} = (\tilde{\boldsymbol{ct}}^{(0)} = (\tilde{c_0}^{(0)}, \tilde{c_1}^{(0)}), \ldots, \tilde{\boldsymbol{ct}}^{(\ell+k-1)} = (\tilde{c_0}^{(\ell+k-1)}, \tilde{c_1}^{(\ell+k-1)})) \in \prod_{j=0}^{\ell-1} \mathcal{R}_{q_j}^2 \times \prod_{i=0}^{k-1} \mathcal{R}_{p_i}^2,$$

where $\tilde{\boldsymbol{ct}}^{(j)} = \tilde{ct_1}^{(j)} \cdot \boldsymbol{swk}^{(j)} \pmod{q_j}$ and $\tilde{\boldsymbol{ct}}^{(\ell+i)} = \tilde{ct_1}^{(\ell+i)} \cdot \boldsymbol{swk}^{(\ell+i)} \pmod{p_i}$ for $0 \le j < \ell$, $0 \le i < k$. Then compute

$$(\hat{c}_0^{(0)}, \ldots, \hat{c}_0^{(\ell-1)}) \leftarrow \mathrm{ModDown}_{\mathcal{D}_\ell \to \mathcal{C}_\ell}(\tilde{c_0}^{(0)}, \ldots, \tilde{c_0}^{(\ell+k-1)})$$
$$(\hat{c}_1^{(0)}, \ldots, \hat{c}_1^{(\ell-1)}) \leftarrow \mathrm{ModDown}_{\mathcal{D}_\ell \to \mathcal{C}_\ell}(\tilde{c_1}^{(0)}, \ldots, \tilde{c_1}^{(\ell+k-1)}).$$

Finally output

$$\hat{\boldsymbol{ct}} = (\hat{\boldsymbol{ct}}^{(0)}, \ldots, \hat{\boldsymbol{ct}}^{(\ell-1)}) \in \prod_{j=0}^{\ell-1} \mathcal{R}_{q_j}^2$$

where $\hat{\boldsymbol{ct}}^{(j)} = (ct_0^{(j)}, 0) + (\hat{c}_0^{(j)}, \hat{c}_1^{(j)}) \pmod{q_j}$ for $0 \le j < \ell$.

- **Mult$_{\boldsymbol{evk}}$($\boldsymbol{ct}, \boldsymbol{ct'}$).** For two ciphertexts $\boldsymbol{ct} = \left(\boldsymbol{ct}^{(j)} = (c_0^{(j)}, c_1^{(j)})\right)_{0 \le j < \ell}$ and $\boldsymbol{ct'} = \left(\boldsymbol{ct}'^{(j)} = (c_0'^{(j)}, c_1'^{(j)})\right)_{0 \le j < \ell}$, for $0 \le j < \ell$, compute

$$d_0^{(j)} \leftarrow c_0^{(j)} \cdot c_0'^{(j)} \pmod{q_j},$$
$$d_1^{(j)} \leftarrow c_0^{(j)} \cdot c_1'^{(j)} + c_1^{(j)} \cdot c_0'^{(j)} \pmod{q_j},$$
$$d_2^{(j)} \leftarrow c_1^{(j)} \cdot c_1'^{(j)} \pmod{q_j}$$

Then compute $\hat{\boldsymbol{d}}_2 \leftarrow \mathrm{KeySwitch}_{\boldsymbol{swk}}\left(\left((0, d_2^{(j)})\right)_{0 \le j < \ell}\right)$. Output

$$\boldsymbol{ct}_{Mult} = \left((d_0^{(j)}, d_1^{(j)}) + \hat{\boldsymbol{d}}_2^{(j)}\right)_{0 \le j < \ell}.$$

- **Rot$_{\boldsymbol{rk}_\kappa}$($\boldsymbol{ct}, \kappa$).** For a ciphtertext $\boldsymbol{ct} = \left(\boldsymbol{ct}^{(j)} = (c_0^{(j)}, c_1^{(j)})\right)_{0 \le j < \ell}$ and a rotation index $\kappa$, first apply automorphism $\tau_\kappa$ to $\boldsymbol{ct}$ and get $\boldsymbol{ct}_\kappa = \left(\boldsymbol{ct}_\kappa^{(j)} = (c_{\kappa 0}^{(j)}, c_{\kappa 1}^{(j)})\right)_{0 \le j < \ell}$, then compute $\hat{\boldsymbol{c}}_\kappa \leftarrow \mathrm{KeySwitch}_{\boldsymbol{rk}_\kappa}\left(\left((0, c_{\kappa 1}^{(j)})\right)_{0 \le j < \ell}\right)$ and finally output

$$\boldsymbol{ct}_{rot} = \left((c_{\kappa 0}^{(j)}, 0) + \hat{\boldsymbol{c}}_\kappa^{(j)}\right)_{0 \le j < \ell}.$$

- **Rescaling($\boldsymbol{ct}$).** For a ciphertext $\boldsymbol{ct} = \left(\boldsymbol{ct}^{(j)} = (c_0^{(j)}, c_1^{(j)})\right)_{0 \le j < \ell} \in \prod_{j=0}^{\ell-1} \mathcal{R}_{q_j}^2$, compute $c_i'^{(j)} \leftarrow q_\ell^{-1} \cdot \left(c_i^{(j)} - c_i^{(\ell-1)}\right) \pmod{q_j}$ for $i = 0, 1$ and $0 \le j < \ell - 1$. Output $\boldsymbol{ct}' \leftarrow \left(\boldsymbol{ct}'^{(j)} = (c_0'^{(j)}, c_1'^{(j)})\right)_{0 \le j < \ell-1} \in \prod_{j=0}^{\ell-2} \mathcal{R}_{q_j}^2$.

## 3 Exact CRR basis conversion algorithm without floats

The fast basis conversion procedure is one of the core technologies in CRR CKKS, but it brings additional errors, which is generally an integer multiple of module $Q_\ell$. Halevi et al. proposed a universal method for eliminating errors, but at the cost of introducing additional floating-point operations. We propose a new method for eliminating errors during CRR basis conversion according to the modulus selection method of the CRR CKKS scheme. Our method converts floating-point operations into very low-cost integer addition and bit operations, thereby improving the computational efficiency. Moreover, our method is also applicable to the CRR BFV and CRR BGV schemes if they use the same modulus selection method as the CRR CKKS scheme.

### 3.1 Algorithm to calculate the error term

According to 2.3, calculating error term $e$ is the core operation for eliminating errors. We notice that $q$ and $q_j$ satisfy the following relation when selecting small prime modules: $q/q_j \in [1-2^{-\eta}, 1+2^{-\eta}]$, where $q = 2^m$ for some $m \in \mathbb{Z}^+$. This means that $q_j$ is very close to $q$. According to 2.3, we know that the calculation formula of the error term is

$$e = \left\lceil \sum_{j=0}^{\ell-1} \frac{[[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j}}{q_j} \right\rfloor .$$

Because $q_j \approx q, j = 0, \ldots, \ell-1$, the above equation can be approximated as

$$e = \left\lceil \sum_{j=0}^{\ell-1} \frac{[[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j}}{q_j} \right\rfloor \approx \left\lceil \sum_{j=0}^{\ell-1} \frac{[[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j}}{q} \right\rfloor = \left\lceil \frac{\sum_{j=0}^{\ell-1}[[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j}}{q} \right\rfloor .$$

So we get the following fast algorithm for calculating $e$:

**Algorithm 1** Fast algorithm for calculating $e$

**Input:** $[x]_{q_j}, \hat{q}_j^{-1}, q, q_j$
**Output:** $e$
Compute $y_0 := [[x]_{q_0} \cdot \hat{q}_0^{-1}]_{q_0}, \ldots, y_{\ell-1} := [[x]_{q_{\ell-1}} \cdot \hat{q}_{\ell-1}^{-1}]_{q_{\ell-1}}$.
Let $e = 0, temp = y_0$.
**for** $j = 1 \to \ell - 1$ **do**
    $temp = temp + y_j$
    **if** $temp > \frac{q}{2}$ **then**
        $e = e + 1$
        $temp = temp - q$
    **else if** $temp < -\frac{q}{2}$ **then**
        $e = e - 1$
        $temp = temp + q$
    **end if**
**end for**
**return** $e$

At this point, we only need to perform up to $3(\ell-1)$ additions and subtractions to obtain $e$, which replaces $\ell$ floating-point division and $\ell$ floating-point addition as well as the final rounding operation in Halevi et al.'s method.

### 3.2 The correctness of the algorithm

We note that the above algorithm obtains an approximate value of $e$. In order to ensure that the result obtained by our algorithm is equal to the true value of $e$, we need to analyze the relation between $\ell$ and $\eta$. We first prove the following lemma:

**Lemma 1.** *For the number $x$ that is uniformly distributed in the real number field, $r \in (0,1)$ is a constant, then the probability that $\lceil x \rfloor$ is equal to $\lceil x+r \rfloor$ is $1-r$. That is*

$$P[\lceil x \rfloor = \lceil x+r \rfloor] = 1-r.$$

*Further more, if $\lceil x \rfloor \neq \lceil x+r \rfloor$, we have $\lceil x+r \rfloor - \lceil x \rfloor = 1$.*

*Proof.* We can choose $x$ in the following way: first choose an integer $a$ uniformly at random, and then choose a real number $b$ between $[0,1)$ independently and uniformly at random, and let $x = a+b$. Then $P[b \in [0, \frac{1}{2})] = P[b \in [\frac{1}{2}, 1)] = \frac{1}{2}$. This is equivalent to $P[\lceil x \rfloor = a] = P[\lceil x \rfloor = a+1] = \frac{1}{2}$. And $x+r = a+b+r$, $0 < b+r < 2$, so $\lceil x+r \rfloor$ depends on the size of $b+r$. In the following, we classify and discuss $b$ and $r$ on the intervals $[0, \frac{1}{2})$ and $[\frac{1}{2}, 1)$, respectively.

1. $b \in [0, \frac{1}{2}), r \in [0, \frac{1}{2})$. In this case, $\lceil x \rfloor = a, b+r \in [0,1)$, so we have $\lceil x+r \rfloor = a$ or $\lceil x+r \rfloor = a+1$. In order to make $\lceil x+r \rfloor = a$, there needs to be $b+r \in [0, \frac{1}{2})$. Because $r$ is a constant, we have

$$P[b+r \in [0, \frac{1}{2}) | b \in [0, \frac{1}{2})] = P[b \in [0, \frac{1}{2} - r) | b \in [0, \frac{1}{2})] = \frac{\frac{1}{2} - r}{\frac{1}{2}} = 1 - 2r.$$

   And when $b+r \in [\frac{1}{2}, 1)$, we have $\lceil x+r \rfloor = a+1 = \lceil x \rfloor + 1$.
2. $b \in [\frac{1}{2}, 1), r \in [0, \frac{1}{2})$. In this case, $\lceil x \rfloor = a+1$, $b+r \in [\frac{1}{2}, \frac{3}{2})$, and we have $\lceil x+r \rfloor = a+1 = \lceil x \rfloor$.
3. $b \in [0, \frac{1}{2}), r \in [\frac{1}{2}, 1)$. In this case, $\lceil x \rfloor = a$, while $b+r \in [\frac{1}{2}, \frac{3}{2})$, so $\lceil x+r \rfloor = a+1$, and the probability that $\lceil x \rfloor$ equals $\lceil x+r \rfloor$ is 0, and $\lceil x+r \rfloor - \lceil x \rfloor = 1$.
4. $b \in [\frac{1}{2}, 1), r \in [\frac{1}{2}, 1)$. In this case, $\lceil x \rfloor = a+1$, $b+r \in [1,2)$, so we have $\lceil x+r \rfloor = a+1$ or $\lceil x+r \rfloor = a+2$. In order to make $\lceil x+r \rfloor = \lceil x \rfloor = a+1$, there needs to be $b+r \in [\frac{1}{2}, \frac{3}{2})$ , so we have

$$P[b+r \in [\frac{1}{2}, \frac{3}{2}) | b \in [\frac{1}{2}, 1)] = P[b \in [\frac{1}{2}, \frac{3}{2} - r) | b \in [\frac{1}{2}, 1)] = \frac{\frac{3}{2} - r - \frac{1}{2}}{1 - \frac{1}{2}} = 2 - 2r.$$

   And when $b+r \in [\frac{3}{2}, 2)$, we have $\lceil x+r \rfloor = a+2 = \lceil x \rfloor + 1$.

In summary, when $r \in [0, \frac{1}{2})$, according to 1 and 2 above, we have

$$\begin{aligned} P[\lceil x \rfloor = \lceil x+r \rfloor] &= P[b+r \in [0, \frac{1}{2}) | b \in [0, \frac{1}{2})] \cdot P[b \in [0, \frac{1}{2})] + 1 \cdot P[b \in [\frac{1}{2}, 1)] \\ &= (1-2r) \cdot \frac{1}{2} + \frac{1}{2} \\ &= 1 - r. \end{aligned}$$

When $r \in [\frac{1}{2}, 1)$, according to 3 and 4 above, we have

$$\begin{aligned} P[\lceil x \rfloor = \lceil x + r \rfloor] &= 0 \cdot P[b \in [0, \frac{1}{2})] + P[b + r \in [\frac{1}{2}, \frac{3}{2}) | b \in [\frac{1}{2}, 1)] \cdot P[b \in [\frac{1}{2}, 1)] \\ &= 0 + (2 - 2r) \cdot \frac{1}{2} \\ &= 1 - r. \end{aligned}$$

Thus when $r \in [0, 1)$ is a constant,

$$P[\lceil x \rfloor = \lceil x + r \rfloor] = 1 - r,$$

and

$$\lceil x + r \rfloor - \lceil x \rfloor \leq 1.$$

With the above lemma, we can prove the following theorem about the correctness of Algorithm 1 :

**Theorem 1.** *Let $r \in (0, 1)$ be a constant. Assume that the output of Algorithm 1 is $e'$. Then when $\eta > \log \frac{r+\ell}{r}$, we have $Pr[e' = e] > 1 - r$. Even if $e' \neq e$, $|e' - e|$ must be 1.*

*Proof.* We notice that

$$\sum_{j=0}^{\ell-1} \frac{[[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j}}{q_j} = \sum_{j=0}^{\ell-1} \frac{[[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j}}{q} \cdot \frac{q}{q_j}.$$

According to $\frac{q}{q_j} \in [1 - 2^{-\eta}, 1 + 2^{-\eta}]$ for $0 \leq j < \ell$, we have

$$\sum_{j=0}^{\ell-1} \frac{[[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j}}{q} \cdot (1 - 2^{-\eta}) \leq \sum_{j=0}^{\ell-1} \frac{[[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j}}{q_j} \leq \sum_{j=0}^{\ell-1} \frac{[[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j}}{q} \cdot (1 + 2^{-\eta}). \tag{1}$$

Let $A = \sum_{j=0}^{\ell-1} \frac{[[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j}}{q}$, $B = \sum_{j=0}^{\ell-1} \frac{[[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j}}{q_j}$, then $e = \lceil B \rfloor$, and the output of Algorithm 1 is $\lceil A \rfloor$. According to (1) we have

$$\lceil A \cdot (1 - 2^{-\eta}) \rfloor \leq e \leq \lceil A \cdot (1 + 2^{-\eta}) \rfloor.$$

We want $e = \lceil A \rfloor$ to be true. All we have to do is make $\lceil A \cdot (1 - 2^{-\eta}) \rfloor = \lceil A \cdot (1 + 2^{-\eta}) \rfloor$, and we can't know the size of $A$ in advance, but we can get the upper and lower bounds of $A$. Specifically, we have the following formula

$$A = \sum_{j=0}^{\ell-1} \frac{[[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j}}{q} \leq \frac{\frac{q_0}{2} + \cdots + \frac{q_{\ell-1}}{2}}{q} \leq \frac{\ell}{2(1 - 2^{-\eta})},$$

$$A = \sum_{j=0}^{\ell-1} \frac{[[x]_{q_j} \cdot \hat{q}_j^{-1}]_{q_j}}{q} \geq \frac{-\frac{q_0}{2} - \cdots - \frac{q_{\ell-1}}{2}}{q} \geq -\frac{\ell}{2(1 - 2^{-\eta})}.$$

Thus $|A| \le \frac{\ell}{2(1-2^{-\eta})}$. Let

$$\alpha = |A \cdot (1 + 2^{-\eta}) - A \cdot (1 - 2^{-\eta})| = |A \cdot 2^{1-\eta}|,$$

and $\alpha \le \frac{\ell \cdot 2^{1-\eta}}{2(1-2^{-\eta})} = \frac{\ell \cdot 2^{-\eta}}{1-2^{-\eta}}$. Observe that the upper bound of $\alpha$ is only determined by $\eta$ and $\ell$, so we can control the upper bound of $\alpha$ by selecting $\eta$ and $\ell$. Let $r \in (0, 1)$ be a constant, and assume that $A$ is uniformly distributed. Let $\alpha < r$, then according to lemma 1, the probability that $\lceil A \cdot (1 - 2^{-\eta}) \rfloor$ equals $\lceil A \cdot (1 + 2^{-\eta}) \rfloor$ is $1 - \alpha > 1 - r$, so the probability of $e = \lceil A \rfloor$ is greater than $1 - r$. Now we turn to analyze the relation between $\eta$ and $\ell$.

Let $\alpha < r$, it is enough to make $\frac{\ell \cdot 2^{-\eta}}{1-2^{-\eta}} < r$. Solve the inequality and we get $2^{-\eta} < \frac{r}{r+\ell}$. Take the logarithm of both sides and multiply by $-1$ to get $\eta > \log \frac{r+\ell}{r}$.

In summary, when $\eta > \log \frac{r+\ell}{r}$, the probability that the output of Algorithm 1 is equal to $e$ is greater than $1 - r$, and even if they are not equal, the difference between the two is $\pm 1$.

According to the theorem 1, given $r$ and $\ell$, we can determine the lower bound of $\eta$. We observe that the lower bound of $\eta$ is a logarithmic function of $\ell$. As $\ell$ increases, so does the lower bound of $\eta$. Therefore, when the scheme is initialized, we can determine the values of $L$ and $\eta$, which naturally satisfy the conditions of the remaining layers. In addition, the logarithmic function grows very slowly, which enables us to select a small enough $r$ to ensure the correctness of Algorithm 1 without causing the lower bound of $\eta$ to be too large. For example, Fig. 1 shows how the lower bound of $\eta$ changes with $\ell$ when taking $r = 0.001$.

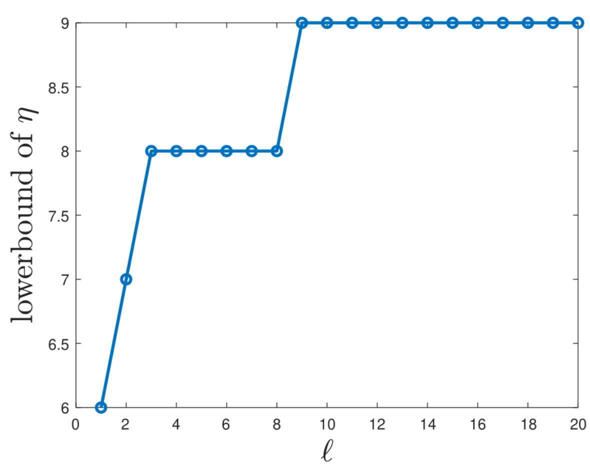



**Fig. 1.** The relation between the lower bound of $\eta$ and $\ell$ for $r = 0.001$.

### 3.3 Optimal parameter selection

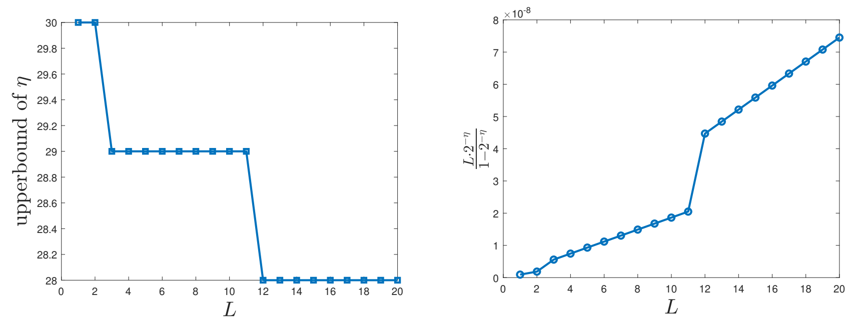
We note that the lower bound of $\eta$ above is loose. In practice, we want $\eta$ to be as big as possible, because the bigger $\eta$ is, the smaller $\frac{\ell \cdot 2^{-\eta}}{1-2^{-\eta}}$ is, the greater the probability that Algorithm 1 is correct. But $\eta$ has an upper bound. This is because when the CRR CKKS scheme is initialized, we are given $N, q, L$, and then select $L$ prime numbers in the interval $[\frac{q}{1+2^{-\eta}}, \frac{q}{1-2^{-\eta}}]$ all of which are congruent to 1 modulo $2N$. The selection of $\eta$ should ensure that the number of prime numbers in the interval $[\frac{q}{1+2^{-\eta}}, \frac{q}{1-2^{-\eta}}]$ that are congruent to 1 modulo $2N$ is greater than or equal to $L$. The larger the $\eta$, the smaller the interval length, and the fewer primes that meet the condition. Given $N, q, L$, we use the binary search to find the maximum value of $\eta$ that we can get.

**Algorithm 2** Binary search algorithm to compute the maximum value of $\eta$.

**Input:** $N, q, L$
**Output:** The maximum value of $\eta$.
Set $\eta_l = 1$.
**while** $\eta_l < \eta_r$ **do**
  Let $\eta_m = \lfloor \frac{\eta_l + \eta_r}{2} \rfloor$. Calculate the number of primes that are congruent to 1 modulo $2N$ in the interval $[\frac{q}{1+2^{-\eta_m}}, \frac{q}{1-2^{-\eta_m}}]$ and $[\frac{q}{1+2^{-(\eta_m+1)}}, \frac{q}{1-2^{-(\eta_m+1)}}]$ respectively, and denote them as $c_m, c_{m+1}$.
  **if** $c_m \geq L$ and $c_{m+1} < L$ **then**
    **return** $\eta_m$.
  **else if** $c_m < L$ **then**
    $\eta_r = \eta_m$
  **else**
    $\eta_l = \eta_m + 1$
  **end if**
**end while**

According to Algorithm 2, we can get the optimal value of $\eta$ when $N$, $q$ and $L$ are set to different values. For example, Fig. 2 shows how the upperbound of $\eta$ and $\frac{L \cdot 2^{-\eta}}{1-2^{-\eta}}$ change with $L$ when $N = 2^{14}, q = 2^{50}$. We observe that the upper bound of $\eta$ decreases very slowly as $L$ increases, and that $\frac{L \cdot 2^{-\eta}}{1-2^{-\eta}}$ does not exceed $8 \times 10^{-8}$. This shows that the probability of Algorithm 1 getting the correct result with this parameter setting is greater than $1 - 8 \times 10^{-8}$, which is very close to 1.

We notice that the image of $\frac{L \cdot 2^{-\eta}}{1-2^{-\eta}}$ changes with $L$ has a large increase from $L = 2$ to $L = 3$ and from $L = 11$ to $L = 12$ respectively, roughly doubling. This is because the upper bound of $\eta$ decreases by 1 from $L = 2$ to $L = 3$ and from $L = 11$ to $L = 12$. The upper bound of $\eta$ is not only related to $L$, but also to the bit length of $N$ and $q$. Specifically, when $q$ increases 1 bit, or $N$ decreases 1 bit, the number of numbers in the interval $[\frac{q}{1+2^{-\eta}}, \frac{q}{1-2^{-\eta}}]$ that are congruent to 1 modulo $2N$ is twice as large, so the number of prime numbers in these numbers is roughly twice as large. Thus the upper bound of $\eta$ increases roughly by 1. In




(a) Relation between upperbound of $\eta$ and $L$. (b) Relation between $\frac{L\cdot 2^{-\eta}}{1-2^{-\eta}}$ and $L$.

**Fig. 2.** When $N = 2^{14}, q = 2^{50}$, the images of how the upper bound of $\eta$ and $\frac{L\cdot 2^{-\eta}}{1-2^{-\eta}}$ change respectively as a function of $L$.

order to ensure the amortization efficiency of the scheme, a relatively large value of $N$ is usually taken in practice. Therefore, in order to maximize the probability of obtaining the correct value of Algorithm 1, we suggest that the bit length of $q$ should be increased appropriately.

## 4 Modifications and comparisons to the original scheme

We have implemented our method on the PALISADE homomorphic encryption open source library. We have modified the process of CKKS scheme in the following aspects: First, in the initialization phase of the scheme, we input three parameters, namely ring dimension $N$, the size of ciphertext module $q$ and the number of prime modulus $L$, where $q$ is a power of 2. We first use Algorithm 2 to calculate the maximum value that $\eta$ can take, and use Algorithm 3 below to obtain $L$ ciphertext modulus. Observe that Algorithm 2 guarantees that there are at least $L$ prime numbers in the interval $[\frac{q}{1+2^{-\eta}}, \frac{q}{1-2^{-\eta}}]$ that meet the condition, which guarantees the correctness of Algorithm 3. We also use Algorithm 2 followed by Algorithm 3 in turn to obtain $k$ KeySwitching modulus $p_0, \ldots, p_{k-1}$.

We note that the PALISADE open source library uses a residual class representation range of $\{0, 1, \ldots, q_j - 1\}$ for ciphertext computation, which allows us to further simplify Algorithm 1, avoid conditional branching, and replace addition and subtraction with bit operations. Specifically, we use Algorithm 4 below to calculate $e$. Similarly, for the basis $\mathcal{B} = \{p_0, \ldots, p_{k-1}\}$ we use the same algorithm to calculate $e_p$.

### 4.1 Precomputation

Notice that when we use exact basis conversion, we need to subtract $e \cdot Q_\ell \pmod{p_i}$ for $0 \le i < k$ or $e_p \cdot P \pmod{q_j}$ for $0 \le j < \ell$. According to Algorithm 4, we know the range of $e$ or $e_p$, i.e. $e \in \{0, \ldots, \ell - 1\}$ and $e_p \in \{0, \ldots, k - 1\}$.

**Algorithm 3** Algorithm for computing $L$ prime modulus

**Input:** $N, q, L, \eta$
**Output:** $L$ prime modulus

Compute $upperbound = \frac{q}{1-2^{-\eta}}, lowerbound = \frac{q}{1+2^{-\eta}}$.
Compute $qFirst = 2N \cdot \lfloor \frac{upperbound}{2N} \rfloor + 1$.
Construct an empty array $result$.
**while** $L \geq 1$ **do**
  **if** $MillerRabinTest(qFirst) = True$ **then**
    $result.append(qFirst)$.
    $L = L - 1$.
  **else**
    $qFirst = qFirst - 2N$.
  **end if**
**end while**
**return** $result$.

**Algorithm 4** A faster algorithm for calculating $e$.

**Input:** $[x]_{q_j}, \hat{q}_j^{-1}, q, q_j$
**Output:** $e$

Compute $y_0 := [[x]_{q_0} \cdot \hat{q}_0^{-1}]_{q_0}, \ldots, y_{\ell-1} := [[x]_{q_{\ell-1}} \cdot \hat{q}_{\ell-1}^{-1}]_{q_{\ell-1}}$.
Set $e = 0, temp = y_0$.
**for** $j = 1 \to \ell - 1$ **do**
  $temp = temp + y_j$
  $e = e + temp \& q$
  $temp = temp \& (q-1)$
**end for**
**return** $e$

Thus we can precompute these values during the initialization of the scheme. Specifically, we compute and store $\alpha \cdot Q_\ell \pmod{p_i}$ for $1 \leq \ell \leq L, 0 \leq \alpha < \ell, 0 \leq i < k$ and $\beta \cdot P \pmod{q_j}$ for $0 \leq \beta < k, 0 \leq j < L$ during the initialization phase of the scheme.

### 4.2 Complexity analysis

We first analyze the complexity of Algorithm 4. Algorithm 4 first evaluates $y_0, \ldots, y_{\ell-1}$, which requires $\ell$ modular multiplication. Next, it loops $\ell - 1$ times, each time performing 3 integer additions and subtractions and 2 bit operations, for a total of $3(\ell - 1)$ integer additions and subtractions and $2(\ell - 1)$ bit operations. Finally we compute

$$\sum_{j=0}^{\ell-1} y_j \cdot [\hat{q}_j]_{p_i} - e \cdot Q_\ell \pmod{p_i} \text{ for } 0 \leq i < k,$$

Since $e \cdot Q_\ell \pmod{p_i}$ for $0 \leq i < k$ has been calculated in the initialization phase of the scheme, a total of $\ell \cdot k$ modular multiplications and $\ell \cdot k$ modular additions are needed. In summary, in the whole exact CRR basis conversion procedure, we perform a total of $\ell \cdot (k+1)$ integral modular multiplications, $\ell \cdot k + 3(\ell - 1)$ integral modular additions or subtractions and $2(\ell - 1)$ bit operations.

We compare the complexity of our method with the original scheme and the method of Halevi et al. in the CRR basis conversion process, respectively. The following table shows the numbers of different operations in the three methods.

| | Modular multiplications | Modular additions or subtractions | Floating-point multiplications | Floating-point additions or subtractions | Bit operations | Floating-point roundings | Whether to eliminate error |
|---|---|---|---|---|---|---|---|
| Our method | $\ell \cdot (k+1)$ | $\ell \cdot k + 3(\ell - 1)$ | 0 | 0 | $2(\ell - 1)$ | 0 | Yes |
| Halevi et al. 's method[14] | $\ell \cdot (k+1)$ | $\ell \cdot k$ | $\ell$ | $\ell - 1$ | 0 | 1 | Yes |
| Fast basis conversion[7, 4] | $\ell \cdot (k+1)$ | $(\ell - 1) \cdot k$ | 0 | 0 | 0 | 0 | No |

**Table 1.** Complexity comparison of CRR basis conversion for ciphertext at $\ell$ layer.

It is a well-known fact that the costs of computational tasks such as integer or floating-point multiplications, division and modular operation are significantly higher than that for integer modular addition and bit operations. The bit operations are even trivial compared to operations such as multiplications, so our method is more efficient than the method of Halevi et al..Our method eliminates the error of CRR basis conversion used in the original fast method, but the performance is not much affected.

## 5 Experimental results

We implemented our method and Helavi et al. 's method in the CKKS scheme of PALISADE library, and combined them with hybrid method[16] and exactRescale method[18]. Our experiments were run on a laptop with AMD Ryzen 7

4800U with Radeon Graphics 1.80 GHz CPU with 16 GB RAM, running Ubuntu 20.04. All experiments were conducted in single-thread mode.

Our experiments were carried out in full packing mode, that is, we encrypted a vector $\boldsymbol{x} \in \mathbb{C}^{N/2}$ each time. Every elements in the vector were randomly selected in the interval $[0, 1]$. In order to measure the precision of the decryption result $\tilde{\boldsymbol{x}}$, we calculate $\frac{2}{N}\sum_{i=1}^{N/2} -\log(|x_i - \tilde{x_i}|)$ to represent the precision.

We use the following notation to indicate the different technologies used in the CRR CKKS scheme:

- *Fast* represents the CRR basis conversion method of the original scheme[7], *ExactHPS* denotes the method proposed by Helevi et al for eliminating errors using floating-point arithmetic[14], and *Exact* denotes our method for eliminating errors using integer arithmetic.
- *ApproxRescaling* denotes the rescaling method of the original scheme[7], and *ExactRescaling* denotes the exact rescaling method proposed by Kim et al[18].
- *GHS* represents the KeySwitching method proposed by Gentry et al[13]., and *Hybrid* represents the hybrid KeySwitching method proposed by Han et al[16].

We compared the efficiency and error of homomorphic multiplication and homomorphic rotation using *Fast*, *ExactHPS* and *Exact* methods under different parameter settings and different techniques, respectively. Table 2,3,4,5 shows the results.

| | | | *Fast* | | | | *Exact* | | | | *ExactHPS* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $q$ | $\log N$ | $K$ | Mult. | | Rot. | | Mult. | | Rot. | | Mult. | | Rot. | |
| | | | prec. | time | prec. | time | prec. | time | prec. | time | prec. | time | prec. | time |
| $2^{40}$ | 13 | 80 | 18.33 | 2.896 | 18.06 | 2.081 | 18.33 | 2.998 | 18.09 | 2.145 | 18.34 | 3.201 | 18.07 | 2.245 |
| | 14 | 200 | 17.598 | 8.51 | 17.32 | 7.549 | 17.613 | 9.053 | 17.372 | 8.487 | 17.604 | 9.889 | 17.382 | 8.691 |
| | 15 | 400 | 16.911 | 34.396 | 16.482 | 31.474 | 16.925 | 37.032 | 16.68 | 32.374 | 16.913 | 33.873 | 16.687 | 32.275 |
| | 16 | 800 | 16.219 | 140.467 | 15.544 | 135.643 | 16.241 | 144.343 | 15.992 | 135.428 | 16.232 | 141.314 | 15.997 | 133.13 |
| $2^{50}$ | 13 | 100 | 25.238 | 3.653 | 24.980 | 2.07 | 25.274 | 3.437 | 25.008 | 2.106 | 25.259 | 2.844 | 25.001 | 2.207 |
| | 14 | 200 | 24.523 | 6.808 | 24.225 | 6.063 | 24.509 | 7.228 | 24.297 | 7.005 | 24.526 | 7.718 | 24.311 | 7.405 |
| | 15 | 400 | 23.822 | 28.233 | 23.547 | 26.263 | 23.843 | 27.374 | 23.605 | 23.258 | 23.826 | 27.713 | 23.628 | 23.967 |
| | 16 | 1000 | 23.141 | 163.081 | 22.418 | 144.776 | 23.162 | 163.795 | 22.923 | 146.913 | 23.146 | 162.335 | 22.912 | 147.347 |

**Table 2.** Comparison of homomorphic multiplication and homomorphic rotation operations between the three methods in *ApproxRescaling* and *GHS* modes, where $K = \lceil \log Q_L \rceil$ and $\lambda \geq 128$ bits. Precision and time are measured in bits and milliseconds, respectively.

We observe that our method and Helevi et al. 's method have an precision improvement of $0 - 0.5$ bits compared to the original scheme, in which the precision improvement of homomorphic multiplication is small, and the precision improvement of homomorphic rotation is large. And the running time of the three methods is about the same. We find that our method do not improve significantly over the original method because the CRR basis conversion operation is not the

| | | | *Fast* | | | | *Exact* | | | | *ExactHPS* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $q$ | $\log N$ | $K$ | Mult. | | Rot. | | Mult. | | Rot. | | Mult. | | Rot. | |
| | | | prec. | time | prec. | time | prec. | time | prec. | time | prec. | time | prec. | time |
| $2^{40}$ | 14 | 200 | 17.610 | 14.035 | 17.298 | 12.364 | 17.618 | 14.763 | 17.392 | 13.07 | 17.62 | 15.106 | 17.377 | 13.912 |
| | 14 | 320 | 17.605 | 20.304 | 17.335 | 18.054 | 17.591 | 20.688 | 17.363 | 17.869 | 17.601 | 23.156 | 17.377 | 18.526 |
| | 15 | 640 | 16.903 | 87.146 | 16.580 | 82.141 | 16.912 | 82.08 | 16.695 | 73.642 | 16.921 | 79.74 | 16.669 | 79.708 |
| | 16 | 880 | 16.219 | 247.18 | 15.923 | 198.613 | 16.235 | 220.947 | 15.989 | 205.13 | 16.215 | 221.865 | 15.982 | 202.583 |
| $2^{50}$ | 13 | 150 | 25.244 | 4.135 | 24.987 | 3.898 | 25.246 | 4.372 | 25.013 | 4.195 | 25.237 | 4.506 | 24.968 | 4.003 |
| | 14 | 300 | 24.508 | 15.451 | 24.3 | 14.211 | 24.532 | 15.662 | 24.312 | 13.907 | 24.528 | 15.352 | 24.318 | 14.016 |
| | 15 | 600 | 23.835 | 65.389 | 23.54 | 57.243 | 23.846 | 66.145 | 23.623 | 58.928 | 23.842 | 62.922 | 23.631 | 59.394 |
| | 16 | 1000 | 23.156 | 218.138 | 22.898 | 188.795 | 23.145 | 221.221 | 22.924 | 185.967 | 23.158 | 221.512 | 22.924 | 186.801 |

**Table 3.** Comparison of homomorphic multiplication and homomorphic rotation operations between the three methods in *ApproxRescaling* and *Hybrid* modes, where $K = \lceil \log Q_L \rceil$ and $\lambda \geq 128$ bits. Precision and time are measured in bits and milliseconds, respectively.

| | | | *Fast* | | | | *Exact* | | | | *ExactHPS* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $q$ | $\log N$ | $K$ | Mult. | | Rot. | | Mult. | | Rot. | | Mult. | | Rot. | |
| | | | prec. | time | prec. | time | prec. | time | prec. | time | prec. | time | prec. | time |
| $2^{40}$ | 13 | 80 | 18.312 | 2.524 | 18.027 | 1.928 | 18.33 | 3.241 | 18.081 | 1.922 | 18.312 | 2.278 | 18.048 | 2.348 |
| | 14 | 200 | 17.61 | 9.919 | 17.346 | 7.583 | 17.623 | 9.389 | 17.374 | 8.839 | 17.613 | 9.279 | 17.37 | 9.058 |
| | 15 | 400 | 16.913 | 39.138 | 16.535 | 31.682 | 16.934 | 39.214 | 16.96 | 31.955 | 16.927 | 38.436 | 16.683 | 33.343 |
| | 16 | 800 | 16.208 | 155.385 | 15.647 | 132.868 | 16.235 | 157.638 | 15.996 | 135.163 | 16.215 | 154.668 | 15.988 | 136.18 |
| $2^{50}$ | 13 | 100 | 25.239 | 2.384 | 24.972 | 3.689 | 25.265 | 2.975 | 24.981 | 2.813 | 25.231 | 3.164 | 25.023 | 2.091 |
| | 14 | 200 | 24.530 | 7.049 | 24.254 | 6.256 | 24.558 | 6.701 | 24.309 | 7.145 | 24.547 | 7.935 | 24.322 | 7.731 |
| | 15 | 400 | 23.82 | 29.438 | 23.44 | 26.137 | 23.85 | 30.288 | 23.625 | 27.936 | 23.823 | 29.171 | 23.613 | 25.921 |
| | 16 | 1000 | 23.15 | 167.717 | 22.433 | 144.642 | 23.153 | 177.682 | 22.913 | 144.525 | 23.145 | 166.603 | 22.92 | 147.33 |

**Table 4.** Comparison of homomorphic multiplication and homomorphic rotation operations between the three methods in *ExactRescaling* and *GHS* modes, where $K = \lceil \log Q_L \rceil$ and $\lambda \geq 128$ bits. Precision and time are measured in bits and milliseconds, respectively.

| | | | *Fast* | | | | *Exact* | | | | *ExactHPS* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $q$ | $\log N$ | $K$ | Mult. | | Rot. | | Mult. | | Rot. | | Mult. | | Rot. | |
| | | | prec. | time | prec. | time | prec. | time | prec. | time | prec. | time | prec. | time |
| $2^{40}$ | 14 | 200 | 17.588 | 15.961 | 17.362 | 12.246 | 17.622 | 15.186 | 17.398 | 12.885 | 17.633 | 16.662 | 17.387 | 13.792 |
| | 14 | 320 | 17.598 | 20.201 | 17.369 | 17.026 | 17.597 | 23.898 | 17.391 | 17.581 | 17.599 | 21.979 | 17.382 | 18.454 |
| | 15 | 640 | 16.917 | 86.446 | 16.583 | 74.926 | 16.925 | 88.25 | 16.672 | 78.572 | 16.932 | 87.347 | 16.682 | 75.827 |
| | 16 | 880 | 16.219 | 250.123 | 15.929 | 198.537 | 16.231 | 273.638 | 15.995 | 208.834 | 16.224 | 279.17 | 15.99 | 212.747 |
| $2^{50}$ | 13 | 150 | 25.219 | 4.317 | 25.005 | 4.04 | 25.271 | 4.667 | 24.996 | 4.109 | 25.245 | 4.396 | 24.976 | 4.136 |
| | 14 | 300 | 24.556 | 17.944 | 24.309 | 14.63 | 24.541 | 17.012 | 24.288 | 14.524 | 24.55 | 18.994 | 24.297 | 16.495 |
| | 15 | 600 | 23.845 | 64.447 | 23.578 | 56.934 | 23.848 | 68.938 | 23.616 | 58.57 | 23.837 | 68.135 | 23.613 | 58.507 |
| | 16 | 1000 | 23.14 | 217.689 | 22.72 | 183.25 | 23.166 | 217.891 | 22.918 | 185.668 | 23.152 | 216.967 | 22.92 | 189.188 |

**Table 5.** Comparison of homomorphic multiplication and homomorphic rotation operations between the three methods in *ExactRescaling* and *Hybrid* modes, where $K = \lceil \log Q_L \rceil$ and $\lambda \geq 128$ bits. Precision and time are measured in bits and milliseconds, respectively.

main source of error, nor is it a major part of the cost time. However, our method has a good theoretical significance. We only use the low cost modular addition and bit operation to eliminate the error of CRR basis conversion procedure, replacing the floating-point operations in the method of Helevi et al., which has a positive significance in simplifying chip design.

## 6 Declarations

The authors have no conflicts of interest to declare that are relevant to the content of this article.

## References


1. PALISADE lattice cryptography library(release v1.11.6). https://palisade-crypto.org/ (2022)
2. Microsoft SEAL. https://github.com/Microsoft/SEAL (2023)
3. Albrecht, M., Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., Halevi, S., Hoffstein, J., Laine, K., Lauter, K., et al.: Homomorphic encryption standard. Protecting privacy through homomorphic encryption pp. 31–62 (2021)
4. Bajard, J.C., Eynard, J., Hasan, M.A., Zucca, V.: 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. pp. 423–442. Springer (2017)
5. Brakerski, Z.: 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. pp. 868–886. Springer (2012)
6. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT) **6**(3), 1–36 (2014)
7. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: 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 25. pp. 347–368. Springer (2019)
8. Cheon, J.H., Kim, A., Kim, M., Song, Y.: HEAAN. https://github.com/snucrypto/HEAAN (2016)
9. Cheon, J.H., Kim, A., Kim, M., Song, Y.: 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 23. pp. 409–437. Springer (2017)
10. Ducas, L., Galbraith, S., Prest, T., Yu, Y.: Integral matrix gram root and lattice Gaussian sampling without floats. In: Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part II 30. pp. 608–637. Springer (2020)
11. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive (2012)

12. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the forty-first annual ACM symposium on Theory of computing. pp. 169–178 (2009)
13. Gentry, C., Halevi, S., Smart, N.P.: 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. pp. 850–867. Springer (2012)
14. Halevi, S., Polyakov, Y., Shoup, V.: 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. pp. 83–105. Springer (2019)
15. Halevi, S., Shoup, V.: Algorithms in helib. In: Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I 34. pp. 554–571. Springer (2014)
16. Han, K., Ki, D.: 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. pp. 364–390. Springer (2020)
17. Hettiarachchi, D.L.N., Davuluru, V.S.P., Balster, E.J.: Integer vs. Floating-Point Processing on Modern FPGA Technology. In: 2020 10th Annual Computing and Communication Workshop and Conference (CCWC). pp. 0606–0612 (2020). https://doi.org/10.1109/CCWC47524.2020.9031118
18. Kim, A., Papadimitriou, A., Polyakov, Y.: 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. pp. 120–144. Springer (2022)
19. Louca, Cook, Johnson: Implementation of IEEE single precision floating point addition and multiplication on FPGAs. In: 1996 Proceedings IEEE Symposium on FPGAs for Custom Computing Machines. pp. 107–116 (1996). https://doi.org/10.1109/FPGA.1996.564761
20. Smart, N.P., Vercauteren, F.: Fully homomorphic SIMD operations. Designs, codes and cryptography **71**, 57–81 (2014)