A Comparison of the Homomorphic Encryption Libraries HElib, SEAL and FV-NFLlib

Carlos Aguilar Melchor[1], Marc-Olivier Kilijian[2], Cédric Lefebvre[3](✉), and Thomas Ricosset[4]

[1] ISAE SUPAERO, University of Toulouse, Toulouse, France
[2] CRM/UDeM and LATECE/UQAM, CNRS, Montréal, QC, Canada
[3] INP, ENSEEIHT, CNRS, IRIT, 31000 Toulouse, France
cedric_lefebvre@orange.fr
[4] Thales, 92230 Gennevilliers, France

Abstract. Fully homomorphic encryption has considerably evolved during the past 10 years. In particular, the discovery of more efficient schemes has brought the computational complexity down to acceptable levels for some applications. Several implementations of these schemes have been publicly released, enabling researchers and practitioners to better understand the performance properties of the schemes. This improved understanding of the performance has led to the discovery of new potential applications of homomorphic encryption, fuelling further research on all fronts.

In this work, we provide a comparative benchmark of the leading homomorphic encryption libraries HElib, FV-NFLlib, and SEAL for large plaintext moduli of up to 2048 bits, and analyze their relative performance.

Keywords: Homomorphic encryption · Benchmark · SEAL · FV-NFLlib · HElib

1 Introduction

Recent advances in *homomorphic cryptography* completely reshaped the possibilities to secure a computing system in untrusted environments. Regarded as cryptography's "Holy Grail", *Fully Homomorphic Encryption (FHE)* enables evaluating arbitrary functions on encrypted data [4,9]. Since the first fully homomorphic encryption schemes were invented, the landscape of FHE has undergone some great changes. Prototypes demonstrating private health diagnosis, signal processing, genomic statistics, and database queries spur hope on the practical deployment of FHE in the near future.

Most of the current applications only consider binary plaintext spaces, and construct binary circuits to compute the desired functions over encrypted data. Existing homomorphic encryption libraries like HElib [11–13] or SEAL [5] are well adapted to this setting. However, they also offer the possibility to choose a larger plaintext space, for situations where the function can be evaluated more efficiently when represented by a modular arithmetic circuit. Nevertheless, very large plaintext moduli do not seem to be the main target for these libraries, and what is feasible in these settings is unclear and remains to be evaluated.

Handling very large plaintext moduli can be useful for two types of applications. First, it becomes possible to compute fixed-point high-precision operations over real (truncated or rounded) data. In this case the modulus bit size is approximately the precision times the multiplicative depth of the circuit. A second set of applications concerns discrete logarithm and factorization-based cryptographic operations over encrypted data in an outsourced setting, with imposed moduli ranging from 256 to 2048 bits.

1.1 This Work

In this paper, we experiment the use of large plaintext moduli with three different FHE libraries in order to evaluate their respective capabilities and performance: SEAL, HElib, and FV-NFLlib. It is worth noting that, for this purpose, we had to modify HElib to be able to handle multi-precision moduli [16], and called this version HElib-MP. Regarding SEAL and FV-NFLlib, we simply used the plain versions of those libraries (SEAL v2.3 for plaintext moduli up to 60 bits and SEAL v2.1 for larger moduli).

Of course, it would be possible to do multi-precision plaintext computations with any of the libraries by using several instances of the encryption scheme with relatively prime plaintext moduli (and then using the Chinese Remainder Theorem to work modulo the product of the plaintext moduli). But using such a method, one cannot use a specific modulus such as those given for ECDSA or RSA, which would not factor appropriately.

We analyze the performance results and compare the impact on the overall performance of the different strategies used in these libraries to handle noise and representation changes.

All our experiments were conducted on a single core of an Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30 GHz; the SHE parameters were selected using the Albrecht-Player-Scott *sagemath* script [1] to ensure at least 128 bits of security (the default security target in SEAL v2.3). Note that, with classical heuristics, doubling security would only increase costs by a factor of two, enabling a generalization of our conclusions to higher security choices.

2 Benchmark Description

We benchmark the libraries for both 1 bit, 64 bits, 256 bits, and 2048 bits plaintexts. The one bit case is not the main focus of our work but is interesting for comparisons passing from one-bit to multi-bit plaintext moduli involves radical behaviour changes.

For each plaintext size, a set of tests is repeatedly run with increasing multiplicative depth until the cost of a test reaches 12+ h. Each test consists in squaring repeatedly a ciphertext until reaching a target multiplicative depth. The output for each test is its running time (averaged over three runs) divided by the depth, which gives the average cost of a multiplication.

As batching is very dependent on the exact value of p, and not just its size, we do not take it into account. The time is thus the average cost of a multiplication for a batch of plaintexts. The batch size will depend on the exact value of p and on the used library. The fact that HElib can provide better batch sizes is something that must be considered on top of plain performance results.

2.1 HElib-MP

In order to be able to run those tests we had to modify HElib. HElib-MP is basically an HElib extension for multi-precision that keeps using NTL for implementing the BGV scheme.

HElib, as is, only handles plaintext space moduli of the form p^r, p and r being single precision integers. As noted in the introduction, it would be possible to do multi-precision plaintext computations with HElib, either by using moduli of the form p^r, or by using several instances of HElib with relatively prime plaintext moduli (and then using the Chinese Remainder Theorem to work modulo the product of the plaintext moduli). But this method forbids the use of specific moduli such as those given for ECDSA or RSA, which would not factor appropriately.

We modified the key generation, encryption, decryption, addition and multiplication routines to allow working with arbitrary plaintext moduli. We therefore adapted the parameters accordingly, such as the parameters concerning the modulus switching.

2.2 Other Libraries Tested

The benchmark is run for HElib-MP, FV-NFLlib and two versions of SEAL: v2.1 [6] and v2.3 [5].

While SEAL v2.1 natively supports arbitrary sized plaintext moduli, it was not optimized to be used with p bigger than 64 bits. SEAL v2.1 stores all ciphertext in full coefficient representation instead of using a CRT or double-CRT representation (see the following performance analysis sections). While this gives a lot of freedom for the choice of the coefficient modulus and in that sense simplifies the parameter selection, performing the polynomial multiplications over $\mathbb{Z}[x]$ is asymptotically less efficient than using the CRT representation(s), and this under-performance increases with larger parameters.

Seal v2.3 makes a big leap forward by using both a CRT representation and the full-RNS[1] variant of FV proposed in [2]. We include both versions on the

[1] RNS or Residue Number Scheme is the proper name of the representation used when we use the Chinese Remainder Theorem.

benchmark to highlight the importance of CRT representations and to try to measure the impact of the modifications brought by [2].

2.3 Parameter Selection

For the tests, $\log q$ is chosen heuristically as $\log p$ times the multiplicative depth, and then it is increased (using the library granularity as q is a product of fixed sizes moduli) until being able to decrypt correctly. Then n is chosen as the smallest power-of-two providing 128 bits of security. This is estimated using the Albrecht-Player-Scott *sagemath* script [1].[2] After n is chosen, correctness is tested again and we iterate (increasing q and n) until we obtain both correctness and an estimation of security above 128 bits.

Note that we cannot give bounds on the probability of the output correctness as we estimate it heuristically. However $\log q$ is large for almost all of our tests and thus the number of margin bits that one would need to include in q's size to ensure a very low (even cryptographically secure) decryption error is negligible.

For the smallest tests (e.g. $\log p = 1$ and low depth) this is not the case, but as the choice of a decryption error probability is application dependent we let this issue aside and just highlight that in order to have low decryption error probabilities the cost of the smallest tests may increase.

3 Benchmark Results

3.1 Ciphertext Modulus Size

We start by presenting the ciphertext modulus size evolution for two settings: $\log p = 1$ and $\log p = 64$. When the plaintext coefficient size is 256 or 2048 the results are equivalent, up to a scale factor to the case $\log p = 64$.

Figure 1 (left) describes the setting $\log p = 1$. Note that, as depth grows, all the libraries converge to quite similar ciphertext modulus sizes. During the relinearization phase, HElib-MP includes the so-called special primes and ciphertext size is a (roughly) constant multiplicative factor larger than without the special primes.

There is also a small asymptotic multiplicative factor that places FV-NFLlib above the other two libraries. This comes from the fact that FV-NFLlib takes larger noise distributions than the other two libraries.

Figure 1 (right) describes the setting $\log p = 64$. We observe the same small multiplicative gap between FV-NFLlib and SEAL due to noise width. On the other hand, as depth grows, a much larger multiplicative factor (of two) appears between HElib-MP and the other two libraries (this factor grows to three in the relinearization phase). The situation is the same for $\log p = 256$ and $\log p = 2048$. At first sight, the noise generated by the multiplications follows $2 \log p$ ($3 \log p$ with special primes) whereas for SEAL and FV-NFLlib it follows $\log p$.

[2] The script returns the security of best known attacks against cryptography based on LWE, we assume the results hold for R-LWE.

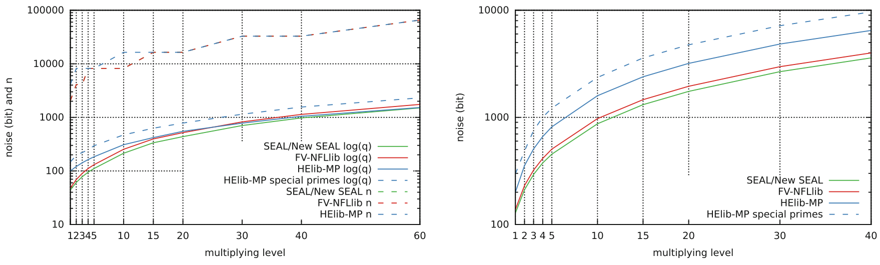

Fig. 1. Evolution of n and $\log q$ needed for correctness as a function of the multiplying depth when $\log p = 1$ (left) and $\log p = 64$ (right). For $\log p = 256$ and $\log p = 2048$, results are similar to the case $\log p = 64$. Results for HElib-MP are given twice, with and without the special primes used in the relinearization operation.

3.2 Computational Costs

Figure 2 presents the computational performance results when $\log p = 1$. Note that the costs for SEAL v2.1 grows much faster than for the other libraries, showing that the usage of CRT is essential for deep multiplications. Costs for SEAL v2.3 and FV-NFLlib cross themselves regularly, and even if SEAL outperforms FV-NFLlib most of the time there is no trend separating the two libraries.

The cost for HElib on the other hand seems to grow less quickly than for the other libraries, starting at depth 25 there is a gap that grows up to a factor 4 for depth 60. Asymptotically HElib seems to fundamentally outperform the other libraries.

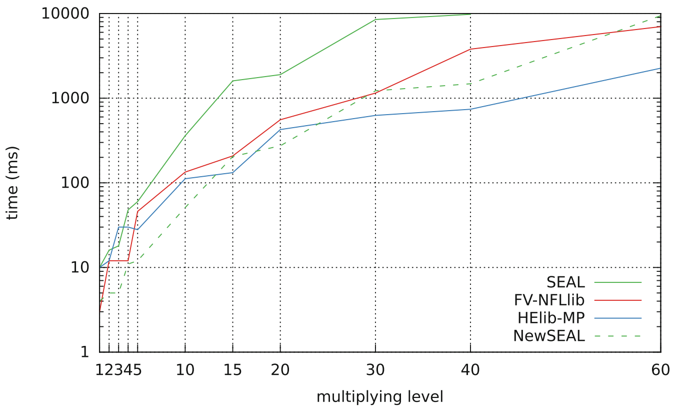

Fig. 2. Average time for one multiplication as a function of the multiplying depth when $\log p = 1$.

Figure 4 presents the computational performance results when $\log p$ is 64, 256, and 2048, except for SEAL where $\log p = 64$ is replaced by $\log p = 60$ to be able to use SEAL v2.3. The first main observation is that SEAL v2.1, as we already saw when $\log p = 1$, has a computational cost that grows much faster than the other libraries. The line corresponding to it stops before the end of the tests for $\log p$ equal 64 or 256 as the tests took more than 12 h after the stopping point. For 2048 SEAL v2.1 could not handle even a single multiplication.

The second main observation is that SEAL v2.3 and FV-NFLlib show almost exactly the same performance results (note however that the scale is logarithmic) for 64 and 256 bit plaintexts. This is remarkable as SEAL v2.3 implements the full-RNS variant of FV whereas FV-NFLlib doesn't. For 2048 bit plaintexts the test could not be run for SEAL v2.3 as the key generation step was stopped after a full day. As the key generation function is the same for both libraries, it is probably due to a minor implementation issue.

The third main observation is that the cost for HElib is higher (starting at a factor 2) when $\log p = 64$ up to a depth around 40 and when $\log p = 256$ up to a depth of 7. Above that it should be the opposite (tests for $\log p = 64$ and depth above 40 were too long). There is therefore a crossing point when $\log q$ is around 2500 bits after which HElib becomes better and better. For plaintexts of 2048 bits HElib is always better (up to a factor 2.5) (Fig. 5).

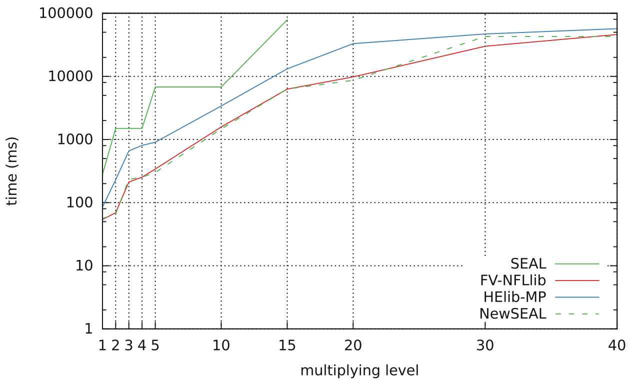

Fig. 3. Average time for one multiplication as a function of the multiplying depth when $\log p = 64$ ($\log p = 60$ for SEAL).

4 Analysis: Noise Estimation

In order to understand the evolution of $\log q$ in the benchmark, noise evolution must be considered for each library. The analysis done in the design document of HElib [12] provides noise variance estimations, whereas in the original paper of FV [8] (and on subsequent implementations) the analysis done provides upper-bounds on the noise (supposing that there is a tail cut on used gaussians).

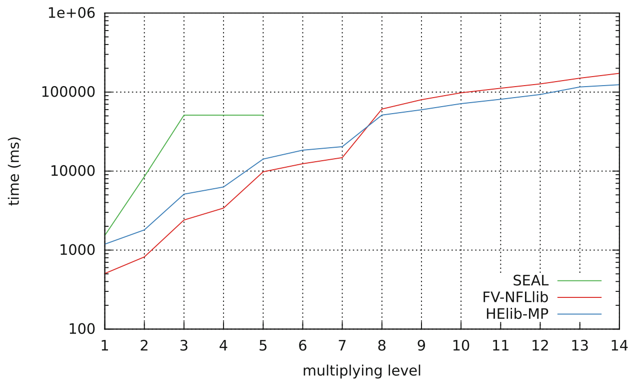

Fig. 4. Average time for one multiplication as a function of the multiplying depth when $\log p = 256$.

The goal of this section is not to provide fine-tuning strategies for parameter derivation but to formalize and justify the observations of the previous section. We therefore approximate the noise estimates using upper-bounds.

Besides the notations of Sect. A we will also use $\delta = \mathsf{sup}(\|\mathbf{a}\cdot\mathbf{b}\|_\infty / \|\mathbf{a}\|_\infty \|\mathbf{b}\|_\infty : a, b \in R)$, the ring expansion factor. We will also note B_{key} and B_{err} the upper-bounds associated to χ_{key} and χ_{err}.

4.1 Noise Growth in FV-NFLlib and SEAL

Following the analysis done in [15], a freshly encrypted ciphertext output by FV.Encrypt has an initial noise term that can be bounded by $V = B_{\mathsf{err}}(1+2\delta B_{\mathsf{key}})$.

After L levels of multiplications, each followed by a relinearization operation, the resulting noise term is bounded by[3]

$$U_L = C_1^L V + L C_1^{L-1} C_2 = L C_1^L (C_2/C_1 + V/L),$$

where

$$C_1 = 2\delta^2 p B_{\mathsf{key}}, \quad C_2 = \delta^2 B_{\mathsf{key}}(B_{\mathsf{key}} + p^2) + \delta\omega \log_\omega(q) B_{\mathsf{err}}.$$

In SEAL we have $B_{\mathsf{key}} = 1$. Using this, and replacing the constants in the expression we get

$$U_L = L\left(2\delta^2 p\right)^L \left(\frac{1+p^2}{2p} + \frac{\omega \log_\omega(q) B_{\mathsf{err}}}{2\delta p} + o_L(1)\right),$$

where $o_L(1)$ is a vanishing term in L. The first fraction is always smaller than (but close to) p and the second fraction is smaller than 1 for all the considered values of ω (word used for relinearization). This upper bound can thus be slightly loosened to get the much simpler expression: $U_L \simeq Lp(2\delta^2 p)^L$. For SEAL, we can therefore expect $\log q$ to be close to $L \log(2\delta^2 p)$ as L grows.

[3] Note that we give here a slightly looser bound to get a simpler expression for C_1.

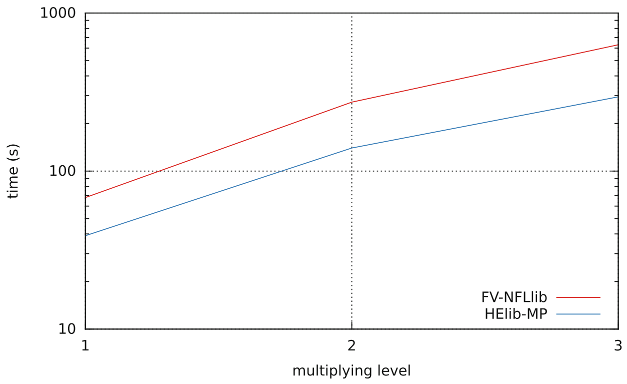

Fig. 5. Average time for one multiplication as a function of the multiplying depth when $\log p = 2048$.

For FV-NFLlib we considered that $B_{\mathsf{key}} = B_{\mathsf{err}}$ (which is the case in all the examples given in the website of the library). Using the same approach we get $U_L \simeq LpB_{\mathsf{err}}(2\delta^2 pB_{\mathsf{err}})$ and thus we can expect $\log q$ to be close to $L \log(2\delta^2 pB_{\mathsf{err}})$ as L grows.

This explains why there is a small multiplicative gap on the cost between FV-NFLlib and SEAL for $\log p = 1$ which vanishes for larger $\log p$ as $\log(2\delta^2 p)$ and $\log(2\delta^2 pB_{\mathsf{err}})$ get are very close for $\log p \in \{64, 256, 2048\}$. Of course, choosing $B_{\mathsf{key}} = 1$ for FV-NFLlib by modifying the code and adapting the security parameters to take the smaller noise into account makes the difference in noise size disappear between FV-NFLlib and SEAL for $\log p = 1$.

4.2 Noise Growth in HElib

Any HElib ciphertext ct carries an inherent noise term, which is an element $\mathbf{v} \in R$ of minimal norm $\|\mathbf{v}\|_\infty$, such that $\mathbf{c_0} + \mathbf{c_1} \cdot \mathbf{s} = p\mathbf{v} + [q_i \boldsymbol{\mu}]_p$. If $\|\mathbf{v}\|_\infty < q_i/p$ decryption works correctly, which means that it returns the plaintext $\boldsymbol{\mu}$. A freshly encrypted ciphertext generated by $\mathsf{HElib.Encrypt}$ has an inherent noise term $\mathbf{v}_{\mathsf{enc}} = \mathbf{u}\cdot\mathbf{e}+\mathbf{e_0}+\mathbf{s}\cdot\mathbf{e_1}$ that can be bounded by $\|\mathbf{v}_{\mathsf{enc}}\|_\infty < V = (1+2\delta)B_{\mathsf{err}}$.

A detailed study of the noise evolution of HElib is beyond the scope of this paper, especially as it has already been done in [12] using the canonical embedding norm, and studying the noise variance, which is probably the best approach to get precise bounds. However, the resulting formulas are quite involved and thus hard to use. We are therefore going to provide a simple intuition on what the upper-bound for the noise resulting from multiplication (and modulus switching, and relinearization) should be.

Multiplication. First, let's consider multiplying $\mathsf{ct} := (\mathbf{c_0}, \mathbf{c_1})$ by itself[4] with

$$\mathbf{c_0} := \mathbf{u} \cdot \mathbf{b} + p\mathbf{e_0} + [q_i \boldsymbol{\mu}]_p \text{ and } \mathbf{c_1} := \mathbf{u} \cdot \mathbf{a} + p\mathbf{e_1}.$$

Note that the first term of the resulting ciphertext we compute is $[q_i^{-1}]_p \mathbf{c_0} \cdot \mathbf{c_0}$. This is the ciphertext part that will contain information about the plaintext $\boldsymbol{\mu}$. We must compute $\left[q_i^{-1}\right]_p \mathbf{c_0} \cdot \mathbf{c_0}$ instead of $\mathbf{c_0} \cdot \mathbf{c_0}$, in order to preserve the invariant which says that the plaintext is encoded, multiplied (mod p) by the current ciphertext modulus (which ensures decryption correctness). The associated term resulting from $\mathbf{c_0} \cdot \mathbf{c_0}$ would be $[q_i^2 \boldsymbol{\mu}^2]_p$ instead of $[q_i \boldsymbol{\mu}]_p$.

Now that the importance of the $[q_i^{-1}]_p$ term is clear let's focus on its impact. When multiplying the noise term by itself we get $p^2 \mathbf{e_0} \cdot \mathbf{e_0}$. As we also multiply all the terms by $[q_i^{-1}]_p$, which can (and probably will) be of the same size as p, we get a bound on this term that is p^3 times the previous noise squared. So if we note U a bound on the previous noise we get after squaring at least a bound of $\delta p^3 U^2$, just considering this term.

Modulus Switching. During a modulus switching operation from the current modulus q_i to the smaller modulus q, the inherent noise term is scaled down by a $\Delta = q_i/q$ factor, then increased by the additive term due to the rounding error. The resulting noise term upper bound is at least $\|\mathbf{v}_{\mathsf{mod}}\|_\infty < \|\mathbf{v}\|_\infty/\Delta + B_{\mathsf{roundin}}$, B_{err} being at least p due to the noise introduced in step 2 to get an element divisible by p. Note that this implies that modulus switching cannot lower the noise below p.

***L*-depth Multiplications.** Let's consider now L-depth multiplications focusing only on the noise of the first ciphertext element (ignoring for now relinearizations). A fresh ciphertext has a noise upper-bound that is $V \sim 2\delta B_{\mathsf{err}}$. After a multiplication the bound on the noise becomes (at least) $4p^3\delta^2 B_{\mathsf{err}}^2$. When p is large, in order to avoid a geometric explosion of the noise, the factor Δ used in the modulus switching step will have to be at least $\Delta \sim p^2$. Thus, in order to handle L multiplications, $\log q$ will have to grow in $2 \log p$ which justifies the observation that $\log q$ grows twice faster for HElib-MP with respect to FV-NFLlib and SEAL.

Relinearization. Finally, if we consider a multiplication followed by a relinearization operation, when $\|\mathbf{v_0}\|_\infty\|, \|\mathbf{v_1}\|_\infty\| < V$, the resulting noise term is bounded by $\|\mathbf{v}_{\mathsf{mul}}\|_\infty < U = tV^2 + \delta B_{\mathsf{err}} \sum_{i=1}^{\ell} B_i/2q'$. As in HElib the size of the B_i is fixed to $1/3 \log q_i$, the analysis done for L-depth multiplications remains correct as long as $\log q' \sim 1/3 \log q_i$. This validates both the previous simplified reasoning and the size observed for HElib-MP with special primes.

5 Analysis: Representation and Computation

In order to analyze the performance results we must first consider the representations used for polynomials in R_p and R_q, the costs associated to the

[4] We consider squaring to reduce the number of variables, as squaring does not change the noise size with respect to multiplying two different ciphertexts.

transformation between representations, and the cost of the different operations associated to the possible representations.

5.1 Transformation Costs

Switching between a DoubleCRT representation (a vector of polynomial values in CRT form) and a (simple) CRT representation (a vector of polynomial coefficients in CRT form) is quite inexpensive as it only requires an NTT or an inverse-NTT transform, whose costs are in $O(n \log n \log q)$.

Lifting the coefficients from a CRT form, or projecting coefficients into a CRT vector, on the other hand, can become quite expensive as the cost for each coefficient is in $O(n \log^2 q)$ and $\log q$ can become quite large when working with large plaintext moduli.

As a rule of thumb, in order to ensure security in lattice based cryptography, n has to grow linearly in $\log q$ and thus the ratio $\log q / \log n$ (which is the ratio between the asymptotic costs of the two transformations) will grow in $O(\log q / \log \log q)$.

In order to optimize performance the best would therefore be to swap between DoubleCRT and CRT representations but to avoid coefficient lifting/projections as much as possible.

5.2 CRT-Compatible Operations

Some polynomial operations can be done directly in DoubleCRT representation:

- Polynomial addition
  DoubleCRT$(\mathbf{a} + \mathbf{b})$ = DoubleCRT$(\mathbf{a})$+ DoubleCRT$(\mathbf{b})$
- Polynomial multiplication
  DoubleCRT$(\mathbf{a} \cdot \mathbf{b})$ = DoubleCRT$(\mathbf{a})\cdot$ DoubleCRT$(\mathbf{b})$
- Scalar multiplication
  DoubleCRT$(p \cdot \mathbf{a}) = p\cdot$DoubleCRT$(\mathbf{a})$
- Uniform sampling
  DoubleCRT$(\mathbf{a})$ for $\mathbf{a} \overset{U}{\leftarrow} R_q \sim \mathbf{a} \overset{U}{\leftarrow} R_q$

The cost of these operations in R_q is in $O(n \log q)$. If the polynomials are in (simple) CRT form the costs are asymptotically the same except for the polynomial multiplication which would lead to a $O(n \log n \log q)$ cost (NTT, then DoubleCRT multiplication, then inverse-NTT).

Coefficient Operations. Note also that subtracting multiples of a constant and checking divisibility by an integer for the polynomial coefficients (operations required by HElib.ModSwitch) also avoid the cost in $O(n \log^2 q)$ that the CRT transforms have. To subtract the constant, it is enough to project it into CRT form (once for the entire polynomial, with a cost in $O(\log^2 q)$, and then subtract the CRT representation from each coefficient as many times as required. To check divisibility by p (in HElib.ModSwitch), it is enough to extend the CRT representation of the coefficients to $q \times p$ and check whether the coordinate associated to p is zero. This extension has a cost $O(n \log q \log p)$, which does not grow quadratically (in $\log q$) as we try to compute more and more homomorphic multiplications.

Finally, if an integer d_1 in CRT representation is divisible by a constant d_2 (i.e. if $d_1 \mod d_2 = 0$), and d_2 is part of the CRT basis in which d_1 is represented, then it is possible to compute d_1/d_2 without leaving the CRT base. Indeed, it is enough to suppress the coordinate associated to d_2 and multiply the other coordinates by the inverse of d_2 with respect to the coordinates moduli.

HElib. Note that the operations described in this section include all that is required for the algorithms HElib.Add, HElib.Mul, HElib.ModSwitch, HElib.Relin. HElib only needs to project/lift integers during key generation, encryption and decryption.

5.3 CRT-Incompatible Operations

Most operations that are incompatible with the CRT representation (e.g. sampling from a gaussian) are only required in key generation, encryption and decryption algorithms.

There is however a key operation in FV, when computing with ciphertexts, that is a priori incompatible with the CRT representation. Indeed, homomorphic multiplication in FV involves computing $[\lfloor p(\mathbf{a} \cdot \mathbf{b})/q \rceil]_q$ with $\mathbf{a}$, $\mathbf{b}$ elements of R_q.

The key issue here is that the multiplication $p(\mathbf{a} \cdot \mathbf{b})$ must be done *over the integers* and not modulo q so we have to lift the CRT representations. Moreover, the division by q and rounding must be done over the coefficients, but of course we do not want to do the polynomial multiplication $\mathbf{a} \cdot \mathbf{b}$ on a coefficient representation which would have a cost quadratic in n.

The trivial approach would require a FastMul algorithm to multiply the polynomials with a coefficient representation for the polynomials and a baseline (i.e. non-CRT) representation for the coefficients. Such an approach would have a cost in $K_c n^{1+c} \log^2 q$ with K_c a constant that can be quite large when c is below 0.58 (for Karatsuba multiplication).

In FV-NFLlib, the idea is to extend the CRT basis to a modulus $q' > q^2 n$, then do the multiplication modulo q'. As this modulus is beyond the infinity norm for the result polynomial, there is no reduction $\mod q'$ and therefore the result is the same as if we did the multiplication over $\mathbf{Z}$.

Note however that the polynomial's coefficients must be lifted and projected multiple times with a cost each time in $n \log^2 q$. Taking $\log q = O(L \log p)$ we get a complexity in $O(nL^2 \log^2 p)$ for FV-NFLlib and in $O(nL \log^2 p)$ for HElib which explains the asymptotic results observed.

SEAL and Full-RNS FV. As noted before, Seal v2.3 implements the full-RNS variant of FV proposed by Bajard et al. at SAC 2016 [2]. The objective of this paper is precisely to deal with FV's operations over $\mathbb{Z}$ that are required both during the homomorphic multiplication algorithm and during the decryption algorithm.

The idea is to stay in CRT representation using CRT-basis pivots to extend basis instead of doing it naively as in FV-FNFLlib. Bajard et al. have many interesting ideas to reduce costs and circumvent the many issues that arise when trying to do this. Unfortunately, as they state in Sect. 4.7 of their paper, they were not able to get rid of the $O(n \log^2 q)$ complexity in the homomorphic multiplication algorithm. This seems to be the reason why SEAL faces the same asymptotic issues as FV-NFLlib when compared to HElib-MP.

6 Conclusions

6.1 Library Choice

The first conclusion we can draw is that there are simple criteria to decide which library should be used (based on computational performance of multiplicative homomorphic operations).

For $\log p = 1$ SEAL v2.3 is the best choice up to a depth of around 12. then SEAL v2.3 and HElib have similar performance until a depth of around 25 and above HElib clearly outperforms all the other libraries.

For $\log p = 60$ FV-NFLlib and SEAL v2.3 both outperform HElib for all practical values (up to a depth slightly above 40). Given that FV-NFLlib and SEAL result in similar performance and that SEAL is more actively developed and more user friendly, in practice the natural choice is SEAL v2.3.

For $\log p$ above 60 two situations arise. If the plaintext modulus can be factorized into sub-moduli of 60 bits then the previous conclusion still applies by using a CRT approach on the plaintext space. If not (for example if it is a cryptographic prime or a hard-to-factor modulus), then SEAL v2.3 cannot be used and FV-NFLlib gives the best results for $\log q < 2000$, above that HElib-MP is the best choice.

6.2 Implementation Recommendations

The first conclusion that arises from our performance analysis is that non-CRT representation of integers, even with Karatsuba multiplication, do not seem to be an option for somewhat homomorphic encryption schemes.

The second is that BGV seems to be plainly superior to FV for very large moduli, even considering the FullRNS variant of Bajard et al. It would be thus very interesting to see a simpler, more optimized, implementation of BGV, for example based on NFLlib, to see if it can match FV's performance for smaller ciphertext moduli.

Finally, both highly specialized libraries such as NFLlib and the FullRNS approach of Bajard et al. seem to provide important performance benefits to FV. An implementation combining both seems feasible and a natural step to go for in order to fully exploit the potential of this scheme.

A Homomorphic Schemes and Libraries

In this section, we recall the variant of the Brakerski-Gentry-Vaikuntanathan (BGV) homomorphic encryption scheme [4], implemented in the software library HElib [11,13]. This description is mostly taken from [10]. We also briefly describe the Fan-Vercauteren (FV) scheme [8], implemented in both SEAL [14] and FV-NFLlib.

A.1 A BGV Scheme Variant

The variant of the BGV scheme implemented in HElib is defined over polynomial rings of the form $R = \mathbb{Z}[x]/\Phi_m(x)$ where m is a parameter and Φ_m is the m-th cyclotomic polynomial. In order to make comparisons simpler we will consider that $m = 2 * n$ with n a power of two, which implies that $\Phi_m(x) = X^n + 1$. The plaintext space is usually the ring $R_p = R/pR$ for an integer p. A plaintext polynomial $\mathbf{a} \in R_p$ is encrypted as a vector over $R_q = R/qR$, where q is an odd public modulus. More specifically, BGV contains a chain of moduli of decreasing size $q_0 > q_1 > \cdots > q_L$ and freshly encrypted ciphertexts are defined modulo q_0. During homomorphic evaluation, we keep switching to smaller moduli after each multiplication until we get ciphertexts modulo q_L, which cannot be multiplied anymore. L is therefore an upper bound on the multiplicative depth of the circuit. We note q_i to indicate that the current modulus we are working with may be any of $\{q_0, \ldots, q_L\}$. We also define another set of primes whose product is noted q'. These are called the *special primes* and they are used to limit the error introduced in relinearization. Noise is drawn from a gaussian distribution $D_{\mathbb{Z}^n,\sigma}$ with standard deviation σ over the integer lattice $\mathbb{Z}^n$. Besides the usual input parameters for key generation used in lattice-based homomorphic schemes (degree n, deviation σ, plaintext modulus p, ciphertext modulus q), it is also possible to choose a relinearization parameter ℓ which enables different trade-offs between noise growth and computational costs. In practice, in HElib this parameter is always set to three.

- $\mathsf{HElib.KeyGen}$:
  - sk Sample a random secret key $\mathsf{sk} := \mathbf{s} \leftarrow R_{q_0}$ with coefficients in $\{-1, 0, 1\}$, where exactly h of them are non-zero.
  - pk Generate a public key $\mathsf{pk} := (\mathbf{b}, \mathbf{a}) \in R_{q_0}^2$, with $\mathbf{a} \leftarrow R_{q_0}$ drawn uniformly at random, and $\mathbf{b} := p\mathbf{e} - \mathbf{a} \cdot \mathbf{s}$, where $\mathbf{e} \leftarrow R_{q_0}$ follows $D_{\mathbb{Z}^n,\sigma}$.
  - rk Generate a relinearization key in $R_{q_0 \cdot q'}^{2\times\ell}$. Split q in ℓ evenly-sized factors $B_1, \ldots, B_\ell$. Define $\mathsf{rk}_i := (\mathbf{a_i}, \mathbf{b_i})^t \in R_{q_0 q'}^2$, with $\mathbf{a_i} \leftarrow R_{q_0 q'}$ drawn uniformly and $\mathbf{b_i} := \left(\prod_{j=0}^{i-1} B_j\right) \mathbf{s}^2 + p\mathbf{e_i} - \mathbf{a_i} \cdot \mathbf{s}$, where $\mathbf{e_i} \leftarrow R_{q_0 q'}$ follows $D_{\mathbb{Z}^n,\sigma}$. Output $\mathsf{rk} := (\mathsf{rk}_1, \ldots, \mathsf{rk}_\ell)$.
- $\mathsf{HElib.Encrypt}(\mathsf{pk}, \boldsymbol{\mu})$: Generate a fresh ciphertext $\mathsf{ct} \in R_{q_0}^2$ from a plaintext $\boldsymbol{\mu} \in R_p$, encrypted using the public key $\mathsf{pk} := (\mathbf{b}, \mathbf{a}) \in R_{q_0}^2$. We have $\mathsf{ct} := (\mathbf{c_0}, \mathbf{c_1})$ with $\mathbf{c_0} := \mathbf{u} \cdot \mathbf{b} + p\mathbf{e_0} + [q_0 \boldsymbol{\mu}]_p$ and $\mathbf{c_1} := \mathbf{u} \cdot \mathbf{a} + p\mathbf{e_1}$, where $\mathbf{u} \leftarrow R_{q_0}$ is drawn uniformly in $\{-1, 0, 1\}^n$ and $\mathbf{e_0}, \mathbf{e_1}$ follow $D_{\mathbb{Z}^n,\sigma}$.

- $\mathsf{HElib.Add}(\mathsf{ct}_0, \mathsf{ct}_1)$: Add two ciphertexts $\mathsf{ct}_0 := (\mathbf{c_{00}}, \mathbf{c_{01}}) \in R_{q_i}^2$ and $\mathsf{ct}_1 := (\mathbf{c_{10}}, \mathbf{c_{11}}) \in R_{q_i}^2$ into a ciphertext $\mathsf{ct}_+ := (\mathbf{c_0}, \mathbf{c_1}) \in R_{q_i}^2$, with $\mathbf{c_0} = \mathbf{c_{00}} + \mathbf{c_{10}}$ and $\mathbf{c_1} = \mathbf{c_{01}} + \mathbf{c_{11}}$.
- $\mathsf{HElib.Mul}(\mathsf{ct}_0, \mathsf{ct}_1)$: Multiply two ciphertexts $\mathsf{ct}_0 := (\mathbf{c_{00}}, \mathbf{c_{01}}) \in R_{q_i}^2$ and $\mathsf{ct}_1 := (\mathbf{c_{10}}, \mathbf{c_{11}}) \in R_{q_i}^2$ into a ciphertext $\widetilde{\mathsf{ct}}_\times := (\mathbf{c_0}, \mathbf{c_1}, \mathbf{c_2}) \in R_{q_i}^3$, with $\mathbf{c_0} = \left[q_i^{-1}\right]_p \mathbf{c_{00}} \cdot \mathbf{c_{10}}$, $\mathbf{c_1} = \left[q_i^{-1}\right]_p (\mathbf{c_{00}} \cdot \mathbf{c_{11}} + \mathbf{c_{01}} \cdot \mathbf{c_{10}})$ and $\mathbf{c_2} = \left[q_i^{-1}\right]_p \mathbf{c_{01}} \cdot \mathbf{c_{11}}$.
- $\mathsf{HElib.ModSwitch}(\mathsf{ct}, q)$: Remove primes from the current modulus to obtain a new target modulus q and scale the ciphertext ct down by a factor of Δ (equal to the current modulus divided by the target modulus) using the following optimized procedure described in [10]:
  1. Reduce the coefficients of ct to obtain $\mathsf{ct}' = \mathsf{ct} \bmod \Delta$,
  2. Add or subtract multiples of Δ from each coefficient in ct' until it is divisible by p,
  3. Set $\mathsf{ct}^* = \mathsf{ct} - \mathsf{ct}'$, // ct^* divisible by Δ, and $\mathsf{ct}^* \equiv \mathsf{ct} \pmod p$
  4. Output ct^*/Δ.
- $\mathsf{HElib.Relin}(\mathsf{rk}, \widetilde{\mathsf{ct}}_\times)$: Relinearize a ciphertext $\widetilde{\mathsf{ct}}_\times := (\mathbf{c_0}, \mathbf{c_1}, \mathbf{c_2}) \in R_{q_i}^3$ into a ciphertext $\mathsf{ct}_\times \in R_{q_i}^2$ using the relinearizing key $\mathsf{rk} := W \in R_{q_0 q'}^{2\times\ell}$. First we break $\mathbf{c_2}$ into a collection of ℓ lower-norm polynomials $\mathbf{c_2}^{(i)}$:
```
1. $\mathbf{d_1} \leftarrow \mathbf{c_2}$
2. For $i \leftarrow 1, \ldots, \ell$:
3.     $\mathbf{c_2}^{(i)} \leftarrow \mathbf{d_i} \bmod B_i$
4.     $\mathbf{d_{i+1}} \leftarrow \left(\mathbf{d_i} - \mathbf{c_2}^{(i)}\right)/B_i$
```
  We then reduce the relinearization key matrix modulo $q_i q'$, and add the small primes corresponding to $q_i q'$ to all the $\mathbf{c_2}^{(i)}$'s, then compute the ciphertext
$$\overline{\mathsf{ct}}_\times := \left(\mathbf{c_0} + \sum_{i=1}^{\ell} W_{i,0} \cdot \mathbf{c_2}^{(i)},\ \mathbf{c_1} + \sum_{i=1}^{\ell} W_{i,1} \cdot \mathbf{c_2}^{(i)}\right) \in R_{q_i q'}^2$$
  Finally, using the modulus switching function defined above, we output $\mathsf{ct}_\times := \mathsf{HElib.ModSwitch}(\overline{\mathsf{ct}}_\times, q_i) \in R_{q_i}^2$.
- $\mathsf{HElib.Decrypt}(\mathsf{sk}, \mathsf{ct})$: Decrypt a ciphertext $\mathsf{ct} := (\mathbf{c_0}, \mathbf{c_1}) \in R_{q_i}^2$ into a plaintext $\boldsymbol{\mu} := \left[\left[q_i^{-1}\right]_p [\mathbf{c_0} + \mathbf{c_1} \cdot \mathbf{s}]_{q_i}\right]_p \in R_p$.

A.2 Fan and Vercauteren's Scheme

The Fan-Vercauteren (FV) scheme is closely related to BGV, but instead of modulus switching it uses the *scale-invariant approach* of Brakerski [3] to control noise growth, and encodes the plaintext in the high-order bits of the coefficients of the ciphertext polynomials, instead of the low-order bits.

It is possible to choose a relinearization parameter, noted in this case ω. This value is a basis in which the relinearization key is decomposed and can be modified to tune performance. Two different distributions are used: χ_{key} and χ_{err}. FV-NFLlib defines $\chi_{\mathsf{key}} = D_{\mathbb{Z}^n, \sigma_{\mathsf{key}}}$ and $\chi_{\mathsf{err}} = D_{\mathbb{Z}^n, \sigma_{\mathsf{err}}}$, for given σ_{err} and σ_{err}. SEAL defines χ_{key} as the uniform distribution over $\{-1, 0, 1\}$ and χ_{err} as FV-NFLlib does.

- $\mathsf{FV.KeyGen}$:
  sk Output $\mathsf{sk} := \mathbf{s} \leftarrow \chi_{\mathsf{key}}$
  pk Let $\mathbf{a} \leftarrow R_q$ drawn uniformly and $\mathbf{e} \leftarrow \chi_{\mathsf{err}}$.
  Define $\mathsf{pk} := ([-\mathbf{a} \cdot \mathbf{s} - \mathbf{e}]_q, \mathbf{a})$.
  rk Generate a relinearization key $\mathsf{rk} := (\mathsf{rk_1}, \ldots, \mathsf{rk}_\ell)$ with $\ell = \lfloor \log_\omega(q) \rfloor + 1$ and $\mathsf{rk_i} := (\mathbf{s^2}\omega^i - (\mathbf{a_i} \cdot \mathbf{s} + \mathbf{e_i}), \mathbf{a_i})$, with $\mathbf{a_i} \leftarrow R_q$ uniformly at random and $\mathbf{e_i} \leftarrow \chi_{\mathsf{err}}$.
- $\mathsf{FV.Encrypt}(\mathsf{pk}, \mu)$: Generate a fresh ciphertext $\mathsf{ct} \in R_q^2$ from a plaintext $\boldsymbol{\mu} \in R_p$, encrypted using the public key $\mathsf{pk} := (\mathbf{b}, \mathbf{a}) \in R_q^2$. We have $\mathsf{ct} := (\mathbf{c_0}, \mathbf{c_1})$ with $\mathbf{c_0} := \mathbf{u} \cdot \mathbf{b} + \mathbf{e_0} + \Delta[\mu]_p$ and $\mathbf{c_1} := \mathbf{u} \cdot \mathbf{a} + \mathbf{e_1}$, where $\mathbf{u}$ follows χ_{key} and $\mathbf{e_0}, \mathbf{e_1}$ follow χ_{err} and $\Delta = q/p$.
- $\mathsf{FV.Add}(\mathsf{ct}_0, \mathsf{ct}_1)$: Add two ciphertexts $\mathsf{ct}_0 := (\mathbf{c_{00}}, \mathbf{c_{01}}) \in R_q^2$ and $\mathsf{ct}_1 := (\mathbf{c_{10}}, \mathbf{c_{11}}) \in R_q^2$ into a ciphertext $\widetilde{\mathsf{ct}}_+ := (\mathbf{c_0}, \mathbf{c_1}) \in R_q^2$, with $\mathbf{c_0} = \mathbf{c_{00}} + \mathbf{c_{10}}$ and $\mathbf{c_1} = \mathbf{c_{01}} + \mathbf{c_{11}}$.
- $\mathsf{FV.Mul}(\mathsf{ct}_0, \mathsf{ct}_1)$: Multiply two ciphertexts $\mathsf{ct}_0 := (\mathbf{c_{00}}, \mathbf{c_{01}}) \in R_q^2$ and $\mathsf{ct}_1 := (\mathbf{c_{10}}, \mathbf{c_{11}}) \in R_q^2$ into a ciphertext $\widetilde{\mathsf{ct}}_\times := (\mathbf{c_0}, \mathbf{c_1}, \mathbf{c_2}) \in R_q^3$, with $\mathbf{c_0} = \lfloor p/q(\mathbf{c_{00}} \cdot \mathbf{c_{10}}) \rceil$, $\mathbf{c_1} = \lfloor p/q(\mathbf{c_{00}} \cdot \mathbf{c_{11}} + \mathbf{c_{01}} \cdot \mathbf{c_{10}}) \rceil)$ and $\mathbf{c_2} = \lfloor p/q(\mathbf{c_{01}} \cdot \mathbf{c_{11}}) \rceil$.
- $\mathsf{FV.Relin}(\mathsf{rk}, \widetilde{\mathsf{ct}}_\times)$: Re-linearize a ciphertext $\widetilde{\mathsf{ct}}_\times := (\mathbf{c_0}, \mathbf{c_1}, \mathbf{c_2}) \in R_q^3$ into a ciphertext $\mathsf{ct}_\times \in R_q^2$. Noting the elements of the relinearization key $\mathsf{rk_i} = (\mathsf{rk_{i,0}}, \mathsf{rk_{i,1}}) \in R_q^2$, and $\mathbf{c_2}^{(i)}$ the decomposition of $\mathbf{c_2}$ in digits in base ω, return

$$\mathsf{ct}_\times := \left(\mathbf{c_0} + \sum_{i=0}^{\ell} \mathsf{rk}_{i,1} \cdot \mathbf{c_2}^{(i)}, \mathbf{c_1} + \sum_{i=1}^{\ell} \mathsf{rk_{i,1}} \cdot \mathbf{c_2}^{(i)} \right)$$

- $\mathsf{FV.Decrypt}(\mathsf{sk}, \mathsf{ct})$: Decrypt a ciphertext $\mathsf{ct} := (\mathbf{c_0}, \mathbf{c_1}) \in R_q^2$ into a plaintext $\boldsymbol{\mu} := [\lfloor p/q(\mathbf{c_0} + \mathbf{c_1} \cdot \mathbf{s}) \rceil]_p$

A.3 Homomorphic Operations

In the schemes presented, the plaintext space is R_p, and homomorphic additions correspond to additions over the ring R_q.[5]

In order to realize other operations (encryption, homomorphic multiplication, modulus switching, relinearization, decryption), we also need to compute multiplications over R_q and, for some of these operations, to manipulate directly the coefficients of the polynomials in R_q.

Batching. Using the Chinese Remainder Theorem it is possible to encrypt a vector of elements of $\mathbb{Z}_p$ [7,17,18] so that homomorphic operations are applied component-wise between the plaintext vectors. This feature is called *batching*, and when used efficiently can massively improve the performance of homomorphic computations, e.g. by allowing thousands of instances of a function to be evaluated simultaneously on different inputs.

[5] One can easily obtain an HE scheme with a plaintext space $\mathbb{Z}_p$ by embedding $\mathbb{Z}_p$ into R_p via $a \in \mathbb{Z}_p \mapsto \mathbf{a} \in R_p$, and homomorphic operations then correspond to arithmetic operations over the ring $\mathbb{Z}_p$.

To understand how batching works, suppose the cyclotomic polynomial $\Phi_m(x)$ factors modulo the plaintext modulus p into a product of irreducible factors $\Phi_m(x) = \prod_{j=0}^{\ell-1} F_j(x) \pmod p$, then a plaintext polynomial $\mathbf{a} \in R_p$ can be viewed as encoding ℓ different small polynomials $a_j = a \bmod F_j$, and each constant coefficient of the a_j can be set to an element of $\mathbb{Z}_p$.

Note that the factorization of the cyclotomic polynomial, $\Phi_m(x) = \prod_{j=0}^{\ell-1} F_j(x) \pmod p$, depends on m and p. If p is fixed for a given application, then the batching capacity will be different for each m depending on how $\Phi_m(x)$ factors modulo p.

HElib gives complete freedom when choosing m but both FV-NFLlib and SEAL use optimized NTT transforms that require power-of-two cyclotomic polynomials. In practice this means that HElib is able, in some cases, to provide a better batching than FV-NFLlib and SEAL.

A.4 DoubleCRT Representation

In this section we recall the *DoubleCRT* representation used in HElib and NFLlib to represent elements of the cyclotomic ring R_q (it is also used with R_p, but for conciseness we will focus on R_q). This representation is composed of two sub-representations, the Chinese remainder theorem (CRT) and the number theoretic transform (NTT).

CRT. We consider the CRT basis as a l-uple of co-prime numbers $(q_0, \dots, q_{l-1})$, we note $q = \prod_{i=0}^{l-1} p_i$. An integer $a \in \mathbb{Z}_q$ is represented in CRT as:

$$\mathrm{CRT}(a) = (a \bmod q_i)_{0 \le i < l}$$

This representation is unique for all $a \in \mathbb{Z}_q$ by the Chinese Remainder Theorem.

NTT. Let q be an integer such that $\mathbb{Z}_q$ contains a primitive m-th root of unity ζ_i. A polynomial $\mathbf{a} \in R_q$ is represented in NTT as:

$$\mathrm{NTT}(\mathbf{a}) = \left(\mathbf{a}(\zeta^j)\right)_{j \in \mathbb{Z}_m^*}$$

DoubleCRT. To use at the same time the CRT representation for coefficients/values and the NTT representation, we need a CRT basis $(q_0, \dots, q_{l-1})$ such that every q_i is chosen so that $\mathbb{Z}/q_i\mathbb{Z}$ contains a primitive m-th root of unity ζ_i. A polynomial $\mathbf{a} \in R_q$ is represented in DoubleCRT as:

$$\mathrm{DoubleCRT}(\mathbf{a}) = \left(\mathbf{a}(\zeta_i^j) \bmod p_i\right)_{0 \le i < l,\ j \in \mathbb{Z}_m^*}$$

References

1. Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. Cryptology ePrint Archive, Report 2015/046 (2015). http://eprint.iacr.org/2015/046
2. Bajard, J.-C., Eynard, J., Hasan, M.A., Zucca, V.: A full RNS variant of FV like somewhat homomorphic encryption schemes. In: Avanzi, R., Heys, H. (eds.) SAC 2016. LNCS, vol. 10532, pp. 423–442. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69453-5_23
3. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical GapSVP. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 868–886. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_50
4. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed.) ITCS 2012, pp. 309–325. ACM, January 2012
5. Chen, H., Han, K., Huang, Z., Jalali, A.: Simple encrypted arithmetic library - seal (v2.3). Technical report, Microsoft, December 2017. https://www.microsoft.com/en-us/research/project/simple-encrypted-arithmetic-library/
6. Chen, H., Laine, K., Player, R.: Simple encrypted arithmetic library - SEAL v2.1. Cryptology ePrint Archive, Report 2017/224 (2017). http://eprint.iacr.org/2017/224
7. Cheon, J.H., et al.: Batch fully homomorphic encryption over the integers. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 315–335. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_20
8. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144 (2012). http://eprint.iacr.org/2012/144
9. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) 41st ACM STOC, pp. 169–178. ACM Press, May/June 2009
10. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 850–867. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_49
11. Halevi, S., Shoup, V.: Algorithms in HElib. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part I. LNCS, vol. 8616, pp. 554–571. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44371-2_31
12. Halevi, S., Shoup, V.: Design and implementation of a homomorphic-encryption library. Technical report, MIT (2014)
13. Halevi, S., Shoup, V.: Bootstrapping for HElib. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part I. LNCS, vol. 9056, pp. 641–670. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46800-5_25
14. Laine, K., Chen, H., Player, R.: Simple encrypted arithmetic library - seal (v2.1). Technical report, Microsoft, September 2016. https://www.microsoft.com/en-us/research/publication/simple-encrypted-arithmetic-library-seal-v2-1/
15. Lepoint, T., Naehrig, M.: A comparison of the homomorphic encryption schemes FV and YASHE. In: Pointcheval, D., Vergnaud, D. (eds.) AFRICACRYPT 2014. LNCS, vol. 8469, pp. 318–335. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06734-6_20
16. Ricosset, T.: Helib-multiprecision (2017). https://github.com/tricosset/HElib-MP

17. Smart, N.P., Vercauteren, F.: Fully homomorphic encryption with relatively small key and ciphertext sizes. In: Nguyen, P.Q., Pointcheval, D. (eds.) PKC 2010. LNCS, vol. 6056, pp. 420–443. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13013-7_25
18. Smart, N., Vercauteren, F.: Fully homomorphic SIMD operations. Cryptology ePrint Archive, Report 2011/133 (2011). http://eprint.iacr.org/2011/133