Secure and Efficient Bootstrapping for Approximate Homomorphic Encryption

Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza,
Jean-Pierre Hubaux

École polytechnique fédérale de Lausanne
first.last@epfl.ch

Abstract. We present a bootstrapping procedure for the full-RNS variant of the approximate homomorphic encryption scheme of Cheon et al., CKKS (Asiacrypt 17, SAC 18). Compared to the previously proposed procedures (Eurocrypt 18 & 19, CT-RSA 20), our bootstrapping is simultaneously more precise and more efficient in terms of CPU cost and number of consumed levels. Moreover, unlike the previous approaches, it does not require the use of sparse secret-keys. Hence, to the best of our knowledge, this is the first procedure that enables efficient bootstrapping for parameters that are 128-bit-secure under more recent attacks on sparse R-LWE secrets.

We achieve this by introducing two novel contributions, applicable to the CKKS scheme: (i) We propose a generic algorithm for homomorphic polynomial evaluation that is scale-invariant and optimal in level consumption. (ii) We optimize the key-switch procedure and propose a new technique to perform rotations (*double hoisting*) that significantly reduces the complexity of homomorphic matrix-vector products.

Our scheme improvements and bootstrapping procedure are implemented in the open source Lattigo library[1]. As an example, bootstrapping a plaintext in $\mathbb{C}^{32768}$ takes 17 seconds, with an output coefficient modulus of 505 bits and a mean precision of 19.2 bits. Hence, we achieve an order of magnitude improvement in bootstrapped throughput (plaintext-bit per second) with respect to the previous best results, while ensuring 128-bit of security.

Keywords: Fully Homomorphic Encryption, Bootstrapping, Implementation

1 Introduction

Homomorphic encryption (HE) is becoming increasingly popular as a solution for processing confidential data in untrustworthy environments, as it enables computing over encrypted data without decrypting them first. Since Gentry's introduction of the first fully-homomorphic encryption (FHE) scheme over ideal lattices [1], continuous efficiency improvements brought these techniques closer to practical application domains.

[1] https://github.com/ldsec/lattigo

As a result, lattice-based FHE schemes are being increasingly used in experimental systems [2]–[4] and some of them are now being proposed as an industry-standard [5].

Cheon et al. [6] introduced a *leveled* encryption scheme, CKKS, which enables the encryption of vectors of complex numbers and the evaluation of arbitrary polynomial functions over those vectors in the encrypted domain. Although the family of *leveled* cryptosystems only offers a finite multiplicative depth, each multiplication *consuming* one level, the CKKS scheme enables the homomorphic re-encryption of an exhausted ciphertext into an almost *fresh* one. This capability, which is commonly called *bootstrapping*, theoretically enables the evaluation of arbitrary-depth circuits. In practice, however, the bootstrapping procedure for CKKS is approximate, and its precision and performance determine the actual maximum depth of a circuit.

Improving the performance of the bootstrapping is crucial for the practicality of FHE, and can be achieved in two ways: (i) Adapting the circuit representation of the bootstrapping by using HE-friendly numerical methods. (ii) Optimizing the scheme operations themselves, which also impacts the scheme performance in general.

Since the initial CKKS bootstrapping procedure by Cheon et al. [7] until the most recent work by Han and Ki [8], the bootstrapping efficiency has improved by several orders of magnitude. However, it is not yet fully practical, and is hence rarely used in real-world applications, where workarounds (e.g., increasing the parameters to accommodate deeper circuits) can yield better performance.

Additionally, all bootstrapping approaches [7]–[9] rely on sparse secret-keys to reduce the depth of their circuit representation. Hence, none of them has so far supported parameters with an equivalent security of at least 128 bits under the recent attacks on sparse R-LWE secrets [10], [11]. The lack of stability in the security of sparse keys has lead the standardization initiatives to exclude sparse secrets and therefore the bootstrapping operation from the currently proposed standards [5]. This raises the question about the practicality of a bootstrapping procedure that would not require the use of sparse secret-keys. In this work, we present a constructive proof to Proposition 1:

Proposition 1. *There exists a practical CKKS bootstrapping procedure that does not require the use of sparse secret-keys.*

1.1 Our Results

We show that it is possible to further improve the CKKS scheme and its bootstrapping operation, up to a point where the constraints on secret-keys sparsity can be removed. Our two major contributions for achieving this goal are: (i) A scale-invariant procedure for the homomorphic evaluation of polynomials which prevents the precision loss due to the approximate rescale. (ii) A faster key-switch procedure specific to the matrix-vector product, that significantly reduces the cost of computing such linear transformations over ciphertexts.

We show that our bootstrapping with standard "dense" keys has a 4.4× larger throughput (bootstrapped plaintext bits per second) than the most recent work from Han and Ki [8] with sparse keys on equivalent hardware. Moreover, when using the sparse-key-adjusted parameters of Curtis and Player [12] for sparse-secrets, our throughput is 15× larger. We now briefly describe the high-level ideas behind our contributions.

Scale-Invariant Homomorphic Polynomial Evaluation The efficiency gains of the full-RNS variant of the CKKS scheme come at the cost of restricting the rescale operation only to the division by the factors q_i of the ciphertext modulus Q. As the moduli q_i are chosen as NTT-friendly primes, the rescale cannot be done by a power of two as in the original CKKS scheme with power-of-two moduli, and it introduces a small scale deviation in the process.

A partial solution, which solves this problem for linear circuits, is to scale plaintexts by a scaling factor Δ which is equal to the q_i by which the ciphertext is going to be divided. A ciphertext multiplied by such plaintexts will then enable exact division and will also keep its original scale. However, when more complex circuits are involved, such as polynomial evaluations, additions between ciphertexts of different scales will eventually happen and will introduce errors. In such a case, one could first scale both ciphertexts to a common scale before adding them together, but this would lead to a sub-optimal level-consumption. For this reason, current state-of-the-art approaches have to trade off between precision and optimal level-consumption.

In this work, we propose a generic algorithm to homomorphically evaluate degree-d polynomial functions, that consumes an optimal number of $\lceil \log(d+1) \rceil$ levels. We generalize the aforementioned partial solution, by observing that a polynomial can be computed by recursively evaluating a linear function. Starting from a user-defined output scale, the intermediate scales can be back-propagated in the recursion by ensuring that each rescale operation in the circuit is exact (thus, so are the homomorphic sums).

Our algorithm enables a depth-optimal and scale-invariant evaluation of arbitrary polynomials and is, to the best of our knowledge, the first general solution for the problem of the approximate rescale arising from the full-RNS variant of the CKKS scheme. We detail this approach and the rescale algorithm in Section 3.

Faster Matrix × Ciphertext Products The most expensive homomorphic operation of the CKKS scheme, regardless of its variant, is the key-switch. This operation is an integral building block of the homomorphic multiplication, slots rotations and conjugation. Moreover, the CKKS bootstrapping requires two linear transformations that involve a large number of rotations (key-switch operations), so minimizing the number of key-switch and/or their complexity will have a significant impact on bootstrapping.

Given an $n \times n$ plaintext matrix M and an encrypted vector $\mathbf{v}$, all recent previous works use a baby-step giant-step (BSGS) algorithm to compute the encrypted product $M\mathbf{v}$ [7]–[9] in $\mathcal{O}(\sqrt{n})$ rotations. These works treat the key-switch procedure as a black-box and focus on reducing the number of times it is executed.

We improve this BSGS algorithm by proposing a new format for rotation keys and a modified key-switch procedure (Section 4), that enables us to add different levels of *hoisting*[2]. This strategy is generic, ant it reduces the theoretical minimum complexity (in terms of modular products) of any linear transformation over ciphertexts. In our bootstrapping procedure, it reduces the cost of the CoeffsToSlots and SlotsToCoeffs steps (Section 5.3) by roughly a factor of 3.

Improved Bootstrapping Procedure We integrate our proposed improvements in the bootstrapping circuit proposed by Cheon et al. [7], Chen et al. [9], Cheon et al. [14] and Han and Ki [8], and propose a new scale-invariant, high-precision and faster bootstrapping circuit with updated parameters which are 128-bit secure, even

[2] The general concept of *hoisting* [13] refers to the ability to take advantage of the commutativity, distributivity and redundancy of some operations to factor them out and reduce the number of times they need to be computed.

considering the most recent attacks on sparse keys [10], [11]. We detail our approach for each step of the improved bootstrapping in Section 5.

Parameterization and Evaluation We discuss the parametrization of the CKKS scheme and its bootstrapping circuit, and propose a procedure to choose and fine-tune the parameters for a given use-case. Our contributions, as well as our bootstrapping, are implemented in the open source library Lattigo: `https://github.com/ldsec/lattigo`. To the best of our knowledge, this is the first public and open source implementation of the bootstrapping for the full-RNS variant of the CKKS scheme.

2 Background and Related Work

We now recall the full-RNS variant of the CKKS encryption scheme and review its previously proposed bootstrapping procedures.

2.1 The Full-RNS CKKS Scheme

We consider the CKKS encryption scheme [6] in its full-RNS variant [15], in which the polynomial coefficients are always represented in the Residue Number System (RNS) and Number Theoretic Transform (NTT) domains, enabling fast integer and polynomial arithmetic. The conversion between the RNS and positional domain is costly due to multi-precision arithmetic, but is only required during the encoding and decoding of plaintext values. This yields an overall speedup across all operations of one order of magnitude compared to the original scheme with equivalent parameters.

Notation For a fixed power-of-two N and $L+1$ distinct primes $q_0, \dots, q_L$, we define $Q_L = \prod_{i=0}^{L} q_i$ and $R_{Q_L} = \mathbb{Z}_{Q_L}[X]/(X^N + 1)$ the cyclotomic polynomial ring over the integers. Unless otherwise stated, we consider elements of R_{Q_L} as their unique representative in the RNS domain: $R_{q_0} \times R_{q_1} \times ... \times R_{q_L} \cong R_{Q_L}$. A polynomial in R_{Q_L} is therefore represented by a $(L+1) \times N$ matrix of coefficients. We denote single elements (polynomials or numbers) in italic, e.g. a, and vectors of such elements in bold, e.g. $\mathbf{a}$, with $\mathbf{a}||\mathbf{b}$ the concatenation of two vectors. We denote $a^{(i)}$ the i-th element of the vector $\mathbf{a}$ or the degree-i coefficient of the polynomial a. We denote $||a||$ the infinity norm of the polynomial (or vector) a and $\mathsf{hw}(a)$ the hamming weight of the polynomial (or vector) a. Unless otherwise stated, all logarithms are base 2.

We denote $[x]_Q$ as the reduction of x modulo Q and $\lfloor x \rfloor$, $\lceil x \rceil$, $\lfloor x \rceil$ as the rounding of x to the next, the previous, and the closest integer, respectively (if x is a polynomial, the operation is applied coefficient-wise).

Plaintext and Ciphertext Space A plaintext is a polynomial $\mathsf{pt} = m(Y) \in \mathbb{Z}_{Q_L}[Y]/(Y^{2n} + 1)$ with $Y = X^{N/2n}$ and n a power-of-two smaller than N. We define the following packings: (i) The *coefficient* packing for which the message $\mathbf{m} \in R^{2n}$ is encoded as the coefficients of a polynomial in Y. (ii) The *slots* packing for which the message $\mathbf{m} \in \mathbb{C}^n$ is first subjected to a discrete Fourier transform and then taken as a polynomial in Y, as in (i). A negacyclic convolution in the coefficient domain results in a Hadamard product in the slot domain, which enables slot-wise arithmetic.

Plaintexts and ciphertexts are respectively represented by the tuples $\{\mathsf{pt}, Q_\ell, \Delta\}$ and $\{\mathsf{ct}, Q_\ell, \Delta\}$, where, for a secret $s \in R_{Q_L}$, pt is a degree-zero polynomial in s, i.e. of R_{Q_ℓ}, and ct is a degree-one polynomial in s, i.e. of $R^2_{Q_\ell}$. We define $Q_\ell = \prod_{i=0}^{\ell} q_i$ as the modulus at level ℓ and Δ as a scaling factor. We denote L as the maximum level (the default level of the keys) and use $0 \leq \ell \leq L$ to represent an intermediate level between the smallest level 0 and the highest level L. We refer to the depth of a circuit as the number of levels required for the evaluation of the circuit.

Scheme RNS-CKKS – Basic Operations

- $\mathsf{Setup}(N, h, b, \sigma)$: For a power of two ring degree N, a secret distribution hamming-weight h, standard deviation σ and a modulus bit-size b: Select the moduli chains $\{q_0, \ldots, q_L\}$ and $\{p_0, \ldots, p_{\alpha-1}\}$ composed of pairwise different NTT-friendly primes close to powers of two such that $\log(\prod_{i=0}^{L} q_i \times \prod_{j=0}^{\alpha-1} p_j) \leq b$. That is, primes of the form $q_i = 2^{s_i} + k_i 2N + 1$ (hence, $q_i \equiv 1 \mod 2N$) for s_i a positive integer size and k_i a small integer such that q_i is a prime. Set $Q_L = \prod_{i=0}^{L} q_i$, $P = \prod_{j=0}^{\alpha-1} p_j$. Define the following distributions over R: χ_{key} where the coefficients are uniformly distributed over $\{-1, 0, 1\}$ and where exactly h coefficients are non-zero. χ_{pkenc} where the coefficients are distributed over $\{-1, 0, 1\}$ with respective probabilities $\{1/4, 1/2, 1/4\}$. χ_{err} where the coefficients are distributed according to a discrete Gaussian distribution with standard deviation σ and truncated to $\lfloor 6\sigma \rfloor$.

- Encode $(\mathbf{m}, \Delta, n, \ell)$ (*coefficients→slots*): For a message $\mathbf{m} \in \mathbb{C}^n$ with $1 \leq n < N$, where n divides N, apply the canonical map $\mathbb{C}^n \rightarrow \mathbb{Z}[Y]/(Y^{2n}+1) \rightarrow R_{Q_\ell}$ with $Y = X^{N/2n}$. Compute $\mathbf{m}' = \lfloor \Delta \cdot \mathrm{FFT}_n^{-1}(\mathbf{m}) \rceil$ and set $\mathbf{m}'_0 || \mathbf{m}'_1 \in \mathbb{Z}^{2n}$, with $\mathbf{m}'_0 = \frac{1}{2}(\mathbf{m}' + \overline{\mathbf{m}'})$ and $\mathbf{m}'_1 = \frac{-i}{2}(\mathbf{m}' - \overline{\mathbf{m}'})$, as a polynomial in Y. Finally, apply the change of variable $Y \rightarrow X$ and return $\{\mathsf{pt}, Q_\ell, \Delta\}$.

- $\mathsf{Decode}(\{\mathsf{pt}, Q_\ell, \Delta\}, n)$ (*slots→coefficients*): For $1 \leq n < N$, where n divides N, apply the inverse of the canonical map $R_{Q_\ell} \rightarrow \mathbb{Z}[Y]/(Y^{2n}+1) \rightarrow \mathbb{C}^n$, with $Y = X^{N/2n}$. Map the input plaintext pt to the vector $\mathbf{m}'_0 || \mathbf{m}'_1 \in \mathbb{Z}^{2n}$ and return $\mathbf{m} = \mathrm{FFT}_n(\Delta^{-1} \cdot (\mathbf{m}'_0 + i \cdot \mathbf{m}'_1))$.

- $\mathsf{SecKeyGen}(\cdot)$: Sample $s \leftarrow \chi_{\mathsf{key}}$ and return the secret key s.

- $\mathsf{SwitchKeyGen}(s, s', \mathbf{w} = (w^{(0)}, \ldots, w^{(\beta-1)}))$: For $\mathbf{w}$ an integer decomposition basis of β elements, sample $a_i \in R_{Q_L P}$ and $e_i \leftarrow \chi_{\mathsf{err}}$ and return the switching key $\mathsf{swk}_{(s \rightarrow s')} = (\mathsf{swk}^{(0)}_{(s \rightarrow s')}, \ldots, \mathsf{swk}^{(\beta-1)}_{(s \rightarrow s')})$, where $\mathsf{swk}^{(i)}_{(s \rightarrow s')} = (-a_i s' + s w^{(i)} P + e_i, a_i)$.

- $\mathsf{PubKeyGen}(s)$: Set the public encryption key $\mathsf{pk} \leftarrow \mathsf{SwitchKeyGen}(0, s, (1))$, the relinearization key $\mathsf{rlk} \leftarrow \mathsf{SwitchKeyGen}(s^2, s, \mathbf{w})$, the rotation keys $\mathsf{rot}_k \leftarrow \mathsf{SwitchKeyGen}(s^{5^k}, s, \mathbf{w})$ (a different key has to be generated for each different k), and the conjugation key $\mathsf{conj} \leftarrow \mathsf{SwitchKeyGen}(s^{-1}, s, \mathbf{w})$ and return: $(\mathsf{pk}, \mathsf{rlk}, \{\mathsf{rot}_k\}_k, \mathsf{conj})$.

- $\mathsf{Enc}(\{\mathsf{pt}, Q_\ell, \Delta\}, s)$: Sample $a \in_u R_{Q_\ell}$ and $e \leftarrow \chi_{\mathsf{err}}$, set $\mathsf{ct} = (-as + e, a) + (\mathsf{pt}, 0)$ and return $\{\mathsf{ct}, Q_\ell, \Delta\}$.

- $\mathsf{PubEnc}(\{\mathsf{pt}, Q_\ell, \Delta\}, \mathsf{pk})$: Sample $u \leftarrow \chi_{\mathsf{pkenc}}$ and $e_0, e_1 \leftarrow \chi_{\mathsf{err}}$, set $\mathsf{ct} = \mathsf{SwitchKey}(u, \mathsf{pk}) +$ $(\mathsf{pt} + e_0, e_1)$ and return $\{\mathsf{ct}, Q_\ell, \Delta\}$.

- $\mathsf{SwitchKey}(d, \mathsf{swk}_{s\to s'})$: For $d \in R_{Q_\ell}$ a polynomial[3], decompose d base $\mathbf{w}$ such that $d = \sum b_{\mathbf{w}}^{(i)} w^{(i)}$ and return $(d_0, d_1) = \lfloor P^{-1} \cdot \sum b_{\mathbf{w}}^{(i)} \mathsf{swk}_{s\to s'}^{(i)} \rceil \bmod Q_\ell$, for $P^{-1} \in \mathbb{R}$.

- $\mathsf{Dec}(\{\mathsf{ct}, Q_\ell, \Delta\}, s)$: For $\mathsf{ct} = (c_0, c_1)$, return $\{\mathsf{pt} = c_0 + c_1 s, Q_\ell, \Delta\}$.

Given the vectors $\mathbf{m}, \mathbf{a} = f(\mathbf{m})$ and $\mathbf{b} = \mathsf{Decode}(\mathsf{Decrypt}(f(\mathsf{Encrypt}(\mathsf{Encode}(\mathbf{m}))))) \in \mathbb{C}^n$, with $f(\cdot)$ a polynomial function, we denote $\log(1/\epsilon)$ the precision of $\mathbf{b}$ relative to $\mathbf{a}$ (the negative log of the average of their absolute difference), i.e. $\log(1/\epsilon_{\text{real}})$ and $\log(1/\epsilon_{\text{imag}})$ where $\epsilon = \frac{1}{n}\sum_{i=0}^{n-1} |a^{(i)} - b^{(i)}|$. Appendix A details the homomorphic operations of CKKS.

2.2 CKKS Bootstrapping

Let $\mathsf{ct} = (c_0, c_1)$ be a ciphertext at level $\ell = 0$, and s a secret key of Hamming weight h, such that $\mathsf{Decrypt}(\mathsf{ct}, s) = [c_0 + c_1 s]_{Q_0} = \mathsf{pt}$. The goal of the bootstrapping operation is to compute a ciphertext ct' at level $L - k > 0$ (where k is the depth of the bootstrapping circuit) such that $Q_{L-k} \gg Q_0$ and $[c_0' + c_1' s]_{Q_{L-k}} \approx \mathsf{pt}$. Note that $[c_0 + c_1 s]_{Q_L} = \mathsf{pt} + Q_0 \cdot I$, where I is an integer polynomial [7]; thus, bootstrapping is equivalent to homomorphically reducing each coefficient of $\mathsf{pt} + Q_0 \cdot I$ modulo Q_0.

Cheon et al. proposed the first procedure [7] to compute this modular reduction, by: (i) homomorphically applying the encoding algorithm, to enable the parallel (slot-wise) evaluation, (ii) computing a modular reduction approximated by a scaled sine function on each slot and (iii) applying the decoding algorithm to retrieve a close approximation of pt without the polynomial I:

$$\underbrace{\mathrm{FFT}^{-1}(\mathsf{pt} + Q_0 \cdot I) = \mathsf{pt}'}_{\text{(i) } \mathsf{Encode}(\mathsf{pt}+Q_0\cdot I)} \;\Rightarrow\; \underbrace{\frac{Q_0}{2\pi}\sin\Big(\frac{2\pi \mathsf{pt}'}{Q_0}\Big) = \mathsf{pt}''}_{\text{(ii) } \mathsf{EvalSine}(\mathsf{pt}')} \;\Rightarrow\; \underbrace{\mathrm{FFT}(\mathsf{pt}'') \approx \mathsf{pt}}_{\text{(iii) } \mathsf{Decode}(\mathsf{pt}'')}.$$

The complexity of the resulting bootstrapping circuit is influenced by two parameters: The first one is the secret-key hamming weight h, which directly impacts the depth of the bootstrapping circuit. Indeed, Cheon et al. show that $||I|| \leq \mathcal{O}(\sqrt{h})$ with very high probability. A denser key will therefore require evaluating a larger-degree polynomial, with a larger depth. The second parameter is the number of plaintext slots n, which has a direct impact on the complexity of the circuit (but not on its depth). By scaling down the values to compress them closer to the origin, Cheon et al. are able to evaluate the sine function using a low degree Taylor polynomial approximation of the complex exponential, and then use repeated squaring (the double angle formula) to obtain the correct result. This approach enables a fast evaluation, but it consumes a large number of levels, so this step dominates the depth of the circuit. Also, the evaluation of the encoding algorithms is low depth but requires a large amount of operations. In fact, the linear transformations are the main bottleneck of the bootstrapping because the number of operations grows linearly with the number of plaintext slots. For example, bootstrapping a ciphertext that packs $n = 2^{15}$ plaintext slots would take more than a day, with less than a minute devoted to the evaluation of the scaled sine function.

This issue was later addressed by Chen et al., who proposed an improved procedure to compute the homomorphic evaluation of the encoding and decoding transformations

[3] SwitchKey does not act directly in a ciphertext; instead, we define it as a generalized intermediate function used as a building block that takes a polynomial as input.

[9]. They observe that these transformations are direct/inverse discrete Fourier transforms (DFTs) that can be efficiently computed by using the Cooley-Tukey algorithm. Chen et al. show that the corresponding homomorphic circuit has depth $\log(n)$, i.e. the number of iterations in the Cooley-Tukey algorithm, because each iteration of the algorithm has a multiplicative depth of one, with only two key-switch operations (see Section 5.3). Chen et al. merge several layers of the Cooley-Tukey algorithm to reduce the number of iterations (and the circuit depth) at the cost of additional complexity per iteration, and they discuss the introduced trade-off. In a concurrent work, Cheon et al. [14] explored techniques to efficiently evaluate DFTs on ciphertexts. They show how to factorize the encoding matrices into a series of $\log_r(n)$ sparse matrices where r is a power-of-two radix. The contributions in [9] and [14], that we review in Section 5.3, enabled a speed up in the homomorphic evaluation of the encoding algorithms by two orders of magnitude. Additionally, Chen et al. [9] improved the approximation of the scaled sine function with a near optimal depth method using a Chebyshev interpolant.

In a more recent work, Han and Ki ported the bootstrapping procedure to the full-RNS variant of CKKS and proposed several improvements to the bootstrapping circuit and to the CKKS scheme [8]. They propose a generalization of its key-switch procedure, by using an intermediate RNS decomposition that enables a finer-grained selection of the scheme parameters and trade-offs between the complexity of the key-switch and the homomorphic capacity of a fresh ciphertext. They also provide an alternative way of computing the scaled sine function of the bootstrapping circuit that takes the magnitude of the underlying plaintext into account and uses the cosine function along with the double angle formula. Their technique requires a smaller amount of non-scalar multiplications than Chen et al.'s [9]. They show that, combined, these changes lead to a speedup factor of 2.5 to 3 compared to the work of Chen et al. [9].

On the implementation side, both [14] and [9] were implemented with HEAAN [16], but only the code of the former was published. The work of [8] was implemented using SEAL [17], but the code was not published.

2.3 Security of Sparse Keys

One commonality between all the aforementioned works is the use of a sparse secret-keys with a hamming weight $h = 64$. A key with a small number of non-zero coefficients is convenient, as it enables a low-depth bootstrapping circuit and is therefore essential for its practicality. However, recent advances in the cryptanalysis of the R-LWE problem have demonstrated that hybrid attacks specifically targeting such sparse keys can severely impact its security [10], [11]. Curtis and Player [12] discussed the practicality and usability of sparse keys. In light of the most recent attacks, they estimate that for a sparse key with $h = 64$ and a ring degree $N = 2^{16}$, a modulus of at most 990 bits is needed to achieve a security of 128 bits. This is much smaller than the modulus size of 1450 bits previously deemed secure (according to Albrecht's estimator [18]).

In their initial bootstrapping proposal, Cheon et al. [7] use the parameters $\{N = 2^{16}, \log(Q) = 2480, h = 64, \sigma = 3.2\}$ and estimated the security of these parameters to 80 bits. Han and Ki [8] proposed new parameter sets, one of which had 128-bit of security: $\{N = 2^{16}, \log(Q) = 1450, h = 64, \sigma = 3.2\}$. However, these estimates are based on results obtained using Albrecht's estimator [18] which, at the time, did not take the most recent attacks on sparse keys into account. In more recent work from Son and Cheon, the security of the parameter set $\{N = 2^{16}, \log(Q) = 1250, h = 64, \sigma = 3.2\}$ was estimated at 113 bits. This sets a loose upper bound to the parameters proposed by Han and Ki, which have a 1450 bits modulus. Therefore, the bootstrapping parameters must be updated to comply with the most recent security recommendations, as none of the current works achieves a security of 128 bits.

3 Scale-Invariant Polynomial Evaluation

In theory, any function that can be approximated by a polynomial can also be evaluated using the full-RNS CKKS scheme. In practice, however, managing the scale throughout a polynomial evaluation in the scheme is not straightforward, and is typically left to the scheme user. Addressing this issue in a generic and practical way is crucial for the adoption of CKKS. As a significant step toward this goal, we introduce a homomorphic polynomial evaluation algorithm that preserves the scale of a ciphertext.

Approximate Rescale The main disadvantage of the full-RNS variant of the CKKS scheme stems from its rescale operation, which does not divide the scale by a power-of-two, as in the original scheme, but by one of the moduli. Those moduli are chosen, for efficiency purposes, as distinct NTT-friendly primes of the form $2^s + k2N + 1$ where s is a positive integer, k a non-zero integer and N the ring degree. Under this constraint, the power-of-two rescale of the original CKKS scheme can only be approximated. The exact ciphertext scale needs to be tracked and the rescale treated as an exact division in order to avoid introducing errors. Thus, ciphertexts at the same level can have slightly different scales (depending on the previous homomorphic operations and the level at which they were previously rescaled). The exact scale resulting from the homomorphic sum between two ciphertexts at different scales is undefined, forcing the user to reconcile these scales.

The Scale Reconciliation Problem Let ct_1, ct_2, ct_3 be three ciphertexts with respective scales $\Delta_1 \approx \Delta_2 \approx \Delta_3$ and at respective levels ℓ_1, ℓ_2, ℓ_3. When computing the monomial $\mathsf{ct}_1 \cdot \mathsf{ct}_2 + \mathsf{ct}_3$, there are three options to apply the rescale:

- *Early rescale*: Rescale the result of $\mathsf{ct}_1 \cdot \mathsf{ct}_2$, then add ct_3. As the additions between two ciphertexts of different scales is not defined, we choose to set the scale to the maximum of the two different scales (assuming that their difference is less than 1). This option ensures an optimal level consumption, since the resulting level will be $\min(\min(\ell_1, \ell_2) - 1, \ell_3)$. However, it introduces an error proportional to the difference between the scale of $\mathsf{rescale}(\mathsf{ct}_1 \cdot \mathsf{ct}_2)$ and ct_3.
- *Late rescale*: Scale ct_3 upward to (exactly) match the scale of $\mathsf{ct}_1 \cdot \mathsf{ct}_2$, add ct_3, then rescale the result. This option results in no error, as the two scales match, but it will not ensure an optimal level consumption, as the resulting level will be $\min(\min(\ell_1, \ell_2) - 1, \ell_3 - 1)$.
- *Dynamic rescale*: Apply *Early rescale* if $\min(\ell_1, \ell_2) > \ell_3$, and apply *Late rescale* otherwise. This option ensures an optimal level consumption while minimizing the number of additions between ciphertexts at different scales.

The choice of the scale-matching strategy significantly impacts the output precision and level-consumption when homomorphically evaluating a polynomial of large degree with the baby-step-giant-step approach (described below). This is highly relevant for our bootstrapping procedure, where relaxing the constraint on sparse keys requires a polynomial evaluation of degree ≈ 450.

Algorithm 1: BSGS algorithm for degree-d polynomials in Chebyshev basis

```
  Input: $p(u) = \sum_{i=0}^{d} c_i T_i(u)$, a degree-$d$ polynomial and a point $t$.
  Output: The evaluation of $p(u)$ at the point $t$.
1 $\mathsf{m} \leftarrow \lceil \log(d+1) \rceil$
2 $\mathsf{l} \leftarrow \lfloor \mathsf{m}/2 \rfloor$
3 $T_0(t) = 1$, $T_1(t) = t$
4 Evaluate $T_2(t), T_3(t), \ldots, T_{2^{\mathsf{l}}}(t)$ and $T_{2^{\mathsf{l}+1}}(t), \ldots, T_{2^{\mathsf{m}-1}}(t)$ using
    $T_{i=a+b}(t) \leftarrow 2T_a(t)T_b(t) - T_{|a-b|}(t)$.
5 Find $q(u)$ and $r(u)$ such that $p(u) = q(u) \cdot T_{2^{\mathsf{m}-1}}(u) + r(u)$.
6 Recurse on step 5 by replacing $p(u)$ by $q(u)$ and $r(u)$ and $\mathsf{m}$ by $\mathsf{m}-1$, until
    the degree of $q(u)$ and $r(u)$ is smaller than $2^{\mathsf{l}}$.
7 Evaluate $q(u)$ and $r(u)$ at the point $t$ using $T_{2^i}(t)$ for $0 \leq i < \mathsf{l}$.
8 Evaluate $p(u)$ at the point $t$ using $q(t)$, $r(t)$ and $T_{2^{\mathsf{m}-1}}(t)$ according to the
    chosen scale-matching strategy.
9 return $p(t)$
```

Baby Step Giant Step (BSGS) algorithm In their bootstrapping circuit, Han and Ki [8] adapt a generic baby-step giant-step polynomial evaluation algorithm to encrypted polynomials in Chebyshev basis, to minimize the number of ciphertext-ciphertext multiplications. Algorithm 1 gives a high-level description of the procedure.

For a polynomial $p(t)$ of degree d, with $\mathsf{m} = \lceil \log(d+1) \rceil$ and $\mathsf{l} = \mathsf{m}/2$, the algorithm first decomposes $p(t)$ into a linear combination of $u_i(t) = \sum_{j=0}^{2^{\mathsf{l}}-1} c_{i,j} T_j(t)$, with $c_{i,j} \in \mathbb{C}$ and $T_{0 \leq j \leq 2^{\mathsf{l}}}$ a pre-computed power basis, such that $p(t) = \sum_{i=0}^{\lfloor d/\mathsf{l} \rfloor} u_i(t) T_{i \cdot \mathsf{l}}(t)$. We denote $u_{\lfloor d/\mathsf{l} \rfloor}(t)$ as u_{max}. The BSGS algorithm then recursively combines the monomials $u'_i(t) = u_{i+1}(t) \cdot T(t) + u_i(t)$ in a tree-like manner using a pre-computed power basis $T_{2^{\mathsf{l}} \leq i \leq \mathsf{m}}(t)$ to minimize the number of non-scalar multiplications. The algorithm requires $2^{\mathsf{l}} + \mathsf{m} - \mathsf{l} - 3 + \lceil (d+1)/2^{\mathsf{l}} \rceil$ non-scalar products and has, in the best case, depth m.

Table 1 reports empirical results on the impact of the different rescale approaches for the homomorphic evaluation of the function $f(x) = \cos(2\pi(x - 0.25)/2^r)$ followed by r evaluations of the double angle formula $\cos(2x) = 2\cos^2(x) - 1$ (each consuming one level). This function plays a central role in the bootstrapping and is therefore an ideal candidate to evaluate the impact of the proposed approaches (see Section 5.4). We observe that the dynamic approach successfully guarantees optimal level consumption and provides a close-optimal precision, but it cannot prevent a scale deviation of $\approx 2^{-30}$ (when normalized). This deviation is significant with respect to the output precision of 2^{-37}, and represents a loss of ≈ 7 bits in subsequent additions.

In Section 3.1, we show how to make this evaluation scale-invariant at no extra cost. This achieves the two-fold effect of preventing error due to scale deviation and making the polynomial evaluation easier to use as a black-box. We also observe that, in practice, Algorithm 1 consumes more than m levels when $d > 2^{\mathsf{m}} - 2^{\mathsf{l}-1}$. In Section 3.2, we show how to achieve an optimal level consumption in this case.

3.1 Scale-Invariant Polynomial Evaluation

Our solution to the problem of the rescale error for the evaluation of a polynomial $p(t)$ is to scale each of the coefficients of the $u_i(t) = \sum_{j=0}^{j<2^{\mathsf{l}}} c_{i,j} T_j(t)$ terms of the

			$\log(1/\epsilon)$ for (K, d, r)				
	Rescale	$\varDelta_\epsilon$	(12, 34, 2)	(15, 40, 2)	(17, 44, 2)	(21, 52, 2)	(257, 250, 3)
Alg. 1 ([8])	*Early*	$2^{-31.44}$	30.36	30.05	29.73	29.19	25.00
	Late	$2^{-28.48}$	37.52	37.80	37.40	37.05	29.41
	Dynamic	$2^{-28.84}$	37.52	37.75	36.24	31.06	29.43
Alg. 2 (ours)	*Early*	0	37.37	37.16	37.15	37.04	29.46
	Dynamic	0	37.63	37.75	37.50	37.05	29.43

Table 1: Comparison of the homomorphic evaluation of a Chebyshev interpolant of degree d of $\cos(2\pi(x-0.25)/2^r)$ in the interval $(-K/2^r, K/2^r)$ followed by r evaluations of $\cos(2x) = 2\cos^2(x) - 1$. The scheme parameters are $N = 2^{16}$, $n = 2^{15}$ slots, $h = 196$, 55-bit moduli and an initial scale $\varDelta = 2^{55}$ (similar to the bootstrapping parameters). $\varDelta_\epsilon$ is the normalized scale difference between the input and output scales, i.e. $|\varDelta_{\text{in}} - \varDelta_{\text{out}}| \cdot \varDelta_{in}^{-1}$.

decomposed polynomial $p(t)$ by some value $\varDelta_{i,j}$ such that, during all the steps of the polynomial evaluation, the rescales are exact and the additions are done between ciphertexts with matching scales. At a high level, this is achieved by starting from the desired output scale, back-propagating it through each multiplication and addition in the algorithm, and computing what scale each ciphertext should have during those operations. Eventually, the back-propagation reaches the coefficients $c_{i,j}$ of the $u_i(t)$ terms and we obtain the appropriate values by which they must be scaled. We exemplify our approach with a toy example $p(t)$ of degree $d = 15$ in Figure 1.

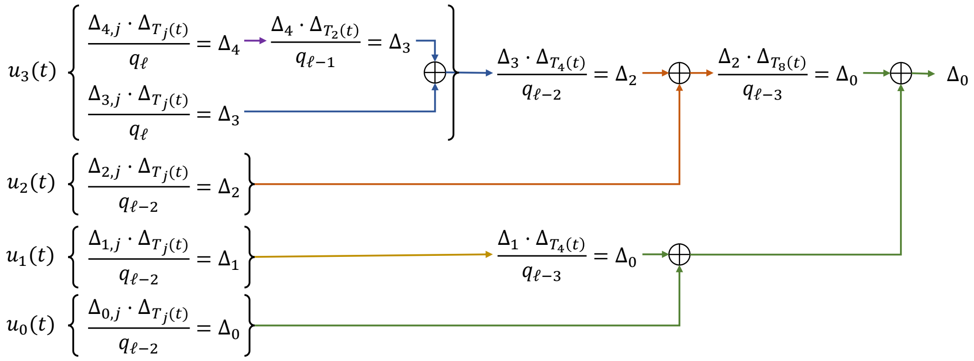
Algorithm 2 is our proposed algorithm for polynomial evaluation, that integrates our scale-propagation for the computation of the coefficient scales. It recursively computes the scale of $q(t)$ to match the scale of $r(t)$ after being multiplied by $T_{2^{\mathsf{m}-1}}$ and rescaled. Conversely, the scale of $r(t)$ is also recursively computed, such that the final scale of the ciphertext, after the polynomial evaluation, either remains unchanged or matches a desired scale; this is the reason why we denote this method as *scale-invariant.*

Table 1 reports the empirical precision of Algorithm 2. We observe that our approach successfully prevents the scale discrepancies and achieves near-exact additions between ciphertexts (up to the inherent and unavoidable rounding errors), achieving an optimal depth, along with a scale-invariant evaluation that does not impact the precision. For experimental purposes, we also evaluated a modification of Algorithm 2 using the early rescale strategy. There is no significant loss in precision, as Algorithm 2 prevents additions between ciphertexts of different scales.

3.2 Optimal Level Consumption

In practice, Algorithm 1 can consume more than m levels because of how the rescale and level management work in the full-RNS variant of the CKKS scheme. In particular, for a polynomial $p(t)$ of degree d with $\mathsf{m} = \lceil \log(d+1) \rceil$ and $\mathsf{l} = \lfloor \mathsf{m}/2 \rfloor$, if $d > 2^{\mathsf{m}} - 2^{\mathsf{l}-1}$, the depth will increase to $\mathsf{m}+1$. Given $p(t) = \sum_{i=0}^{\lfloor d/\mathsf{l} \rfloor} u_i(t) T_{i\cdot\mathsf{l}}(t)$ with $u_i(t) = \sum_{j=0}^{2^{\mathsf{l}}-1} c_{i,j} T_j(t)$, the depth to evaluate $p(t)$ is determined by the depth of $u_{\mathsf{max}}(t) T_{\mathsf{max}\cdot\mathsf{l}}(t)$ ($u_{\mathsf{max}} = u_{\lfloor d/\mathsf{l} \rfloor}$), which are evaluated sequentially as follows:

$$\left(\left(\left(\left(\sum_{j=0}^{2^{\mathsf{l}}-1} c_{\mathsf{max},j} T_j(t)\right) \cdot T_{2^{\mathsf{l}}}(t)\right) \cdot T_{2^{\mathsf{l}+1}}(t)\right) \cdots\right) \cdot T_{2^{\mathsf{m}-1}}(t).$$



$u_3(t)$
$\frac{\Delta_{4,j} \cdot \Delta_{T_j(t)}}{q_\ell} = \Delta_4$
$\frac{\Delta_4 \cdot \Delta_{T_2(t)}}{q_{\ell-1}} = \Delta_3$
$\frac{\Delta_{3,j} \cdot \Delta_{T_j(t)}}{q_\ell} = \Delta_3$
$\frac{\Delta_3 \cdot \Delta_{T_4(t)}}{q_{\ell-2}} = \Delta_2$
$\frac{\Delta_2 \cdot \Delta_{T_8(t)}}{q_{\ell-3}} = \Delta_0$
Δ_0
$u_2(t)$
$\frac{\Delta_{2,j} \cdot \Delta_{T_j(t)}}{q_{\ell-2}} = \Delta_2$
$u_1(t)$
$\frac{\Delta_{1,j} \cdot \Delta_{T_j(t)}}{q_{\ell-2}} = \Delta_1$
$\frac{\Delta_1 \cdot \Delta_{T_4(t)}}{q_{\ell-3}} = \Delta_0$
$u_0(t)$
$\frac{\Delta_{0,j} \cdot \Delta_{T_j(t)}}{q_{\ell-2}} = \Delta_0$

Fig. 1: Scale propagation for the evaluation of $p(t) = u_0(t) + u_1(t) \cdot T_4(t) + (u_2(t) + u_3(t) \cdot T_4(t)) \cdot T_8(t)$. The coefficient scale factors $\Delta_{i,j}$ of the $u_i(t)$ terms are computed as $\Delta_i q_{\ell_i} / \Delta_{T_j}$ where j is the degree of $u_i(t)$. The evaluation of $u_3(t)$ is different, to ensure an optimal depth consumption (see Section 3.2). The symbol $\oplus$ denotes an addition between two (matching) scales.

Due to the multiplications with $c_{i,j}$, the depth to evaluate $u_{\mathsf{max}}(t)$ is $\mathsf{l} + 1$, which brings it one level below the first power-of-two basis $T_{2^{\mathsf{l}}}$, i.e. one additional level is consumed. However, as long as the multiplication with the successive power-of-two basis T_{2^i} has at least one "gap" (one of the T_{2^i} is not used), this additional level can be "absorbed by the gap" and the resulting depth remains m. This is however not the case when $d > 2^{\mathsf{m}} - 2^{\mathsf{l}-1}$, because all the T_{2^i} are used, and the depth becomes $\mathsf{m} + 1$.

Our solution to this problem is to optimally evaluate $u_{\lfloor d/\mathsf{l} \rfloor}(t)$, i.e. its depth is at most l. To do so, we recursively call again the algorithm on $u_{\lfloor d/\mathsf{l} \rfloor}(t)$ if $d \leq 2^{\mathsf{m}} - 2^{\mathsf{l}-1}$ and until $\mathsf{l} = 1$. This ensures the lowest possible decomposition level: $\mathsf{l} = 1$, for which the case $d \leq 2^{\mathsf{m}} - 2^{\mathsf{l}-1}$ does not happen. E.g., ax^3 could be evaluated as $a \cdot x^3$, which would consume 1+2 levels, but by decomposing it as $(a \cdot x) \cdot x^2$ we instead consume 1+1 levels (the constant multiplication is absorbed). These additional recursions add $\lceil \log(d - 2^{\mathsf{m}} + 2^{\mathsf{l}-1} + 1) \rceil$ non-scalar multiplications, but enable the systematic evaluation of any polynomial using exactly m levels, as illustrated in the toy example of Figure 1.

3.3 Conclusions

For an extra cost of $\lceil \log(d - 2^{\mathsf{m}} + 2^{\mathsf{l}-1} + 1) \rceil$ ciphertext-ciphertext products, our algorithm guarantees an optimal depth, thus an optimal level consumption. This extra cost is negligible, compared to the base cost of Algorithm 1, which is $2^{\mathsf{l}} + \mathsf{m} - \mathsf{l} - 3 + \lceil (d+1)/2^{\mathsf{l}} \rceil$.

Also, our proposed algorithm offers the possibility to choose the output scale and to guarantee that rescales and additions throughout the entire polynomial evaluation are exact, hence preventing the precision loss related to the scale deviation, and making the procedure easier to use. In Section 5, we show that these features are highly relevant for an efficient bootstrapping procedure. As linear transforms and constant multiplications can already be made scale-invariant, our scale-invariant polynomial evaluation is the remaining building block to enable scale-invariant circuits of arbitrary depth.

Algorithm 2: EvalRecurse

```
Input: A target scale $\Delta$, an upper-bound $\mathsf{m}$, a stop factor $\mathsf{l}$, a degree-$d$
       polynomial $p(t) = \{c_0, \dots, c_d\}$, and a power basis
       $T = \{T_0, \dots, T_{2^{\mathsf{l}}}, T_{2^{\mathsf{l}+1}}, \dots, T_{2^{\mathsf{m}}}\}$, pre-computed for a ciphertext $\mathsf{ct}$.
Output: A ciphertext encrypting the evaluation of $p(\mathsf{ct})$.
if $d < 2^{l}$ then
    if $p(t) = u_{max}(t)$ and $l > 2^{m}$ - $2^{l-1}$ and $l > 1$ then
        $\mathsf{m} \leftarrow \lceil \log(d+1) \rceil$
        $\mathsf{l} \leftarrow \lfloor \mathsf{m}/2 \rfloor$
        return $\mathsf{EvalRecurse}(\Delta, p(t), \mathsf{m}, \mathsf{l}, T)$
    else
        $\mathsf{ct} \leftarrow \lfloor c_0 \cdot \Delta \cdot q_{T_d} \rceil$
        for $i = d; i > 0; i = i - 1$ do
            $\mathsf{ct} \leftarrow \mathsf{Add}(\mathsf{ct}, \mathsf{MultConst}(T_i, \lfloor (c_i \cdot \Delta \cdot q_{T_d})/\Delta_{T_i} \rceil))$
        end
        return $\mathsf{Rescale}(\mathsf{ct})$
    end
end
$\mathsf{m} \leftarrow \mathsf{m} - 1$
Express $p(t)$ as $q(t) \cdot T_{2^{\mathsf{m}}} + r(t)$
$\mathsf{ct}_0 \leftarrow \mathsf{EvalRecurse}((\Delta \cdot q_{T_{\mathsf{m}-1}})/\Delta_{T_{\mathsf{m}}}, q(t), \mathsf{m}, \mathsf{l}, T)$
$\mathsf{ct}_1 \leftarrow \mathsf{EvalRecurse}(\Delta, r(t), \mathsf{m}, \mathsf{l}, T)$
$\mathsf{ct}_0 \leftarrow \mathsf{Mul}(\mathsf{ct}_0, T_{2^{\mathsf{m}}})$
if $level(ct_0) > level(ct_1)$ then
    $\mathsf{ct}_0 \leftarrow \mathsf{Add}(\mathsf{Rescale}(\mathsf{ct}_0), \mathsf{ct}_1)$
else
    $\mathsf{ct}_0 \leftarrow \mathsf{Rescale}(\mathsf{Add}(\mathsf{ct}_0, \mathsf{ct}_1))$
end
return $\mathsf{ct}_0$
```

4 Key-switch and Improved Matrix-Vector Product

The key-switch procedure is the generic public-key operation in the CKKS scheme. By generating specific public *switching-keys* derived from secret keys s' and s, it is possible to enable public re-encryption of ciphertexts from key s' to s. Beyond the public encryption procedure (switching from $s' = 0$ to s), key-switch is required by most homomorphic operations to cancel the effect of encrypted arithmetic on the decryption circuit, ensuring the compactness of the scheme. More specifically, homomorphic multiplication requires re-encryption from key s^2 back to s, whereas slot-rotations require re-encryption from the equivalent rotation of s back to s. The cost associated with the key-switch dominates the cost of these operations by one to two orders of magnitude (because it requires several NTTs and CRT reconstructions). Hence, optimizations of the key-switch algorithm have a high impact on the overall efficiency of the scheme.

We propose an optimized switching-key format and associated key-switch algorithm (Section 4.1). We then apply them to the specific case of rotation-keys, and further improve the hoisted rotation technique (Section 4.2), as introduced by Halevi et al. [13]. Finally, we propose a modified procedure for matrix-vector multiplications over

```
Algorithm 3: key-switch
  Input: $c \in R_{Q_\ell}$, the switching key $\mathsf{swk}_{s \to s'}$ (both in the NTT domain).
  Output: $(a', b') \in R^2_{Q_\ell}$.
  $\mathbf{d} \leftarrow \mathsf{Decompose}(c)$ (Algorithm 9 in Appendix D)
  $(a, b) \leftarrow \sum d^{(i)} \cdot \mathsf{swk}_{q_{\alpha_i}}$ // $d^{(i)} = \big[[c]_{q_{\alpha_i}}\big]_{Q_\ell P}$
  $a' \leftarrow \mathsf{ModDownNTT}(a)$ (Algorithm 10 in Appendix D) // $a' = \lfloor P^{-1} \cdot a \rceil$
  $b' \leftarrow \mathsf{ModDownNTT}(b)$ // $b' = \lfloor P^{-1} \cdot b \rceil$
  return $(a', b')$
```

packed ciphertexts (Section 4.3) that features a new *double-hoisting* optimization. For each procedure, we provide mathematical notes to enable an efficient implementation.

4.1 Improved Key-switch Keys and Procedure

Given a ciphertext modulus $Q_L = \prod_{j=0}^{L} q_j$, we use a basis $\mathbf{w}$ composed of products among the q_i, as described by Han and Ki [8]. In addition, we include the entire base $\mathbf{w}$ in the keys, as done by Bajard et al. and Halevi et al. [19], [20]; this saves one constant multiplication during the key-switch and enables a simpler key-switch keys generation. Hence, we propose a simpler and more efficient hybrid between these previous approaches (an overview of which is given in Appendix B).

Specifically, our basis is $w^{(i)} = \frac{Q_L}{q_{\alpha_i}}[(\frac{Q_L}{q_{\alpha_i}})^{-1}]_{q_{\alpha_i}}$ with $q_{\alpha_i} = \prod_{j=\alpha\beta}^{\min(\alpha(\beta+1)-1, L)} q_j$ for $0 \leq i \leq \beta$, $\beta = \lceil (L+1)/\alpha \rceil$ and α a positive integer. Thus, the key-switch keys have the following format:

$$\begin{aligned}
\mathsf{swk}^0_{q_{\alpha_i}} &= [-a_i s + s' \cdot P \cdot \tfrac{Q_L}{q_{\alpha_i}} \cdot [(\tfrac{Q_L}{q_{\alpha_i}})^{-1}]_{q_{\alpha_i}} + e_i]_{Q_L P},\\
\mathsf{swk}^1_{q_{\alpha_i}} &= [a_i]_{Q_L P}.
\end{aligned}$$

We set $P = \prod_{j=0}^{\alpha-1} p_j$, and the bit-size of P such that $q_{\alpha_i} \leq P$, $\forall \alpha_i$. As shown by Han and Ki [8], this leads to a negligible error introduced by the key-switch operation. Algorithm 3 describes the associated key-switch procedure. Since the decomposition basis is already included in the key, there is no need to multiply it to each decomposed components as a part of the final ciphertext re-composition and no need to store additional constants.

Remark 1. We note that the base-$\mathbf{w}$ RNS decomposition in the key-switch algorithm is achieved by switching the polynomial out of the NTT domain, reducing the polynomial modulo each q_{α_i} and switching it back to the NTT domain. We observe that $[[a]_{Q_L}]_{q_{\alpha_i}} = [a]_{q_{\alpha_i}}$ if q_{α_i} divides Q_L. Thus, for all moduli shared between q_{α_i} and Q_L, the vectors before and after the decomposition are unchanged. Hence, the NTT representation of those vectors can be obtained at no cost, as they are directly given by the polynomial to decompose. This observation saves up to $\alpha \cdot \lceil (L+1)/\alpha \rceil$ NTTs during the $\mathsf{Decompose}$ step. The complexity analysis is derived in Appendix C.1.

Remark 2. When q_j does not divide q_{α_i}, then $w^{(i)} \equiv 0 \bmod q_j$; otherwise, $w^{(i)} \equiv 1 \bmod q_j$. This enables a simpler implementation of the key-switch key generation and no constants need to be computed or stored: In the case of q_j dividing q_{α_i}, we can replace $w^{(i)}$ by 1, and for all other q_j and all factors of P we can replace $w^{(i)}$ by 0.

4.2 Improved Hoisted Rotations

The rotation operation in CKKS is defined by the automorphism $\phi_k : X \to X^{5^k} (\text{mod } X^N + 1)$. Its effect is to rotate the message slots by k positions to the left[4]. Rotations are extensively used by circuits such as matrix multiplications. After a rotation, the secret under which the ciphertext is encrypted is changed from s to $\phi_k(s)$. Thus, a key-switch $\phi_k(s) \to s$ must be applied to go back to the original key.

Halevi et al. [13] show that since ϕ_k is an automorphism, it distributes over addition and multiplication, hence it commutes with the power-of-two base decomposition. Since ϕ_k acts individually on the coefficients by permuting them without changing their norm (the modular reduction by $X^N + 1$ will at most induce a sign change), it also commutes with the special RNS decomposition (see Eq. (1) in Appendix B): $[\phi_k(a)]_{q_{\alpha_i}} = \phi_k([a]_{q_{\alpha i}})$. If we view a polynomial of R_{Q_L} as an $(L+1) \times N$ matrix, the effect of ϕ_k is to permute its columns; the coefficient-wise modular reduction (and RNS basis extension) independently acts on each column, and both operations commute.

Hence, when several rotations have to be applied on the same ciphertext, $[a]_{q_{\alpha_i}}$ can be pre-computed an re-used for each subsequent rotation: $\sum \phi_k([a]_{q_{\alpha_i}}) \cdot \mathsf{rot}_{k,q_{\alpha_i}}$. Whereas the procedure proposed by Halevi et al. requires the computation of the automorphism for each of the $a_{q_{\alpha_i}}$, this technique significantly reduces the number of NTTs and CRT reconstruction, and thus the overall complexity (see Appendix C.1).

We further exploit the properties of the automorphism to reduce its execution cost, by observing that ϕ_k^{-1} can be directly pre-applied on the rotation keys:

$$\begin{aligned} \widetilde{\mathsf{rot}}^0_{k,q_{\alpha_i}} &= [-a_i\phi_k^{-1}(s) + s \cdot P \cdot \tfrac{Q_L}{q_{\alpha_i}} \cdot [(\tfrac{Q_L}{q_{\alpha_i}})^{-1}]_{q_{\alpha_i}} + e_i]_{Q_L P}, \\ \widetilde{\mathsf{rot}}^1_{k,q_{\alpha_i}} &= [a_i]_{Q_L P}. \end{aligned}$$

Compared to a $\mathsf{rot}_{k,q_{\alpha_i}}$, a traditional rotation key as defined in Section 2.1, the required number of computed automorphisms per-rotation can be reduced to only one:

$$\sum \phi_k([a]_{q_{\alpha_i}}) \cdot \mathsf{rot}_{k,q_{\alpha_i}} = \phi_k\Big(\sum [a]_{q_{\alpha_i}} \cdot \widetilde{\mathsf{rot}}_{k,q_{\alpha_i}}\Big).$$

Our improved algorithm for hoisted rotations is detailed in Algorithm 4. We minimize the number of evaluations of ϕ in Algorithm 4, making it constant regardless of the ciphertext level or the basis decomposition.

Let r be the number of rotations, N the ring degree, ℓ the current level and $\beta = \lceil(\ell+1)/\alpha\rceil$, with α a positive integer. The total complexity of this algorithm in number of integer multiplications (derived in Appendix C.2), that we denote $\#\mathsf{Mul}_{\mathbb{Z}_p}$, is

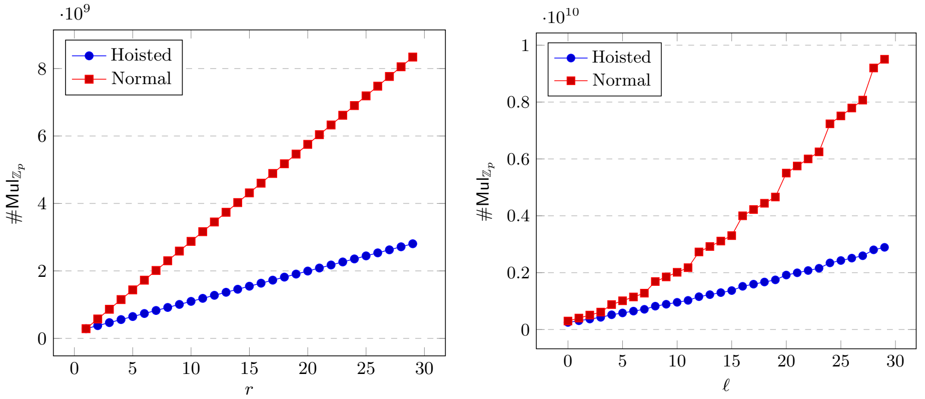
$$\begin{aligned} \#\mathsf{Mul}_{\mathbb{Z}_p} &= N \cdot \log(N) \cdot \Big((\ell+1) \cdot (\beta + 1 + 2r) + 2r\alpha\Big) \\ &\quad + N \cdot (\ell+1) \cdot \Big(\beta\alpha + 2r \cdot (\alpha + \beta + 1)\Big) \\ &\quad + N \cdot \alpha \cdot \Big(\beta + 2r \cdot (\beta + 1)\Big). \end{aligned}$$

Figure 2 compares the complexity of regular and hoisted rotations for varying number of rotations r and ciphertext level ℓ. It shows that using hoisted rotations

[4] To give an intuition of this operation, we refer to the encoding matrix SF_n in Section 5.3. Raising each root of unity to the power of 5^k is the same as shifting the rows of the encoding matrix by k positions.

Algorithm 4: Optimized Hoisting Rotations

```
Input: $\mathsf{ct} = (c_0, c_1) \in R^2_{Q_\ell}$ and a set of $r$ rotation keys $\widetilde{\mathsf{rot}}_{r_k}$ (both in the NTT domain).
Output: $\mathbf{v}$ a list containing each $k_r$ rotation of $\mathsf{ct}$.
$\mathbf{d} \leftarrow \mathsf{Decompose}(c_1)$ (Algorithm 9 in Appendix D)
foreach $r_k$ do
    $(a, b) \leftarrow \sum d^{(i)} \cdot \widetilde{\mathsf{rot}}_{r_k, q_{\alpha_i}}$
    $(a', b') \leftarrow (\mathsf{ModDownNTT}(a), \mathsf{ModDownNTT}(b))$ (Algorithm 10 in Appendix D)
    $\mathbf{v}_{r_k} \leftarrow \{\phi_{r_k}(c_0 + a'), \phi_{r_k}(b')\}$
end
return v
```



(a) Varying number of rotations r. Parameters: $\{N = 2^{16}, \ell = 21, \alpha = 4, \beta = \lceil(\ell+1)/\alpha\rceil\}$.

(b) Varying input levels ℓ for $r = 20$ rotations. Parameters: $\{N = 2^{16}, \alpha = 4, \beta = \lceil(\ell+1)/\alpha\rceil\}$.

Fig. 2

scales significantly better for any $r > 1$ and any level ℓ. This is especially the case when ℓ is large, which is relevant for the bootstrapping, whose first step is a linear transformation computed at the maximum ciphertext level.

4.3 Faster Matrix-Vector Operations

We now discuss the application of homomorphic slot-rotations to the computation of matrix × vector products on packed ciphertexts. The ability to efficiently apply generic linear transformations to encrypted vectors is pivotal for most applications. In particular, the homomorphic computation of the CKKS encoding and decoding procedures, which carried a prohibitive cost in the original bootstrapping procedure, are linear transformations.

Halevi and Shoup proposed to express an $n \times n$ matrix M in diagonal form and to use a baby-step giant-step (BSGS) algorithm to evaluate the matrix product in $\mathcal{O}(\sqrt{n})$ rotations [21], [22]. At the time of writing, all the existing bootstrapping procedures are based on this approach. We now break down the cost of the BSGS algorithm, analyze its components, and, based on our observations, present our improvements to this approach.

Dominant Complexity of Rotations The dominant cost factor of the BSGS algorithm of Halevi et al. is the number of rotations, as these require key-switch operations. These rotations comprise four steps:

1. *Decompose*: Decompose a polynomial of R_{Q_ℓ} in base $\mathbf{w}$ and return the result in $R_{Q_\ell P}$. This operation requires NTTs and CRT basis extensions.
2. *MultSum*: This is a sum of products of polynomials in $R_{Q_\ell P}$. This operation only requires coefficient-wise additions and multiplications.
3. *ModDown*: Divide a polynomial of $R_{Q_\ell P}$ by P and return the result in R_{Q_ℓ}. This operation requires NTTs and CRT basis extensions.
4. *Permute*: Apply the automorphism ϕ_k on a polynomial of R_{Q_ℓ}. This operation is only a permutation of the coefficients and has no impact on complexity.

Let n be the number of non zero diagonals of M, and two integers n_1, n_2 such that $n = n_1 n_2$, the complexity of the original BSGS algorithm (Algorithm 14 in Appendix D) is $n_1 + n_2$ rotations, which is minimized when $n_1 \approx n_2$ (because the n_2 rotations can be pre-computed and reused):

$$(n_2 + n_1) \cdot (\mathit{Decompose} + \mathit{MultSum} + \mathit{ModDown} + \mathit{Permute}),$$

to which we must also add $2n_2n_1$ multiplications and additions in R_{Q_ℓ} (lines 8 and 9 of Algorithm 14). We denote *inner loop* and *outer loop* the loops that respectively depend on the value n_1 and n_2.

Figure 3 shows the weight of each four steps within the total complexity of ℓ rotations (see Appendix C.2 for the underlying complexity analysis). The complexity of the steps *MultSum* and *Permute* is close to negligible compared to *Decompose* and *ModDown*, as products and additions are very cheap compared to NTTs and CRT basis extensions. We base our optimizations on that observation.

Improved BSGS Algorithm We propose a new optimization that we refer to as *double-hoisting*. This optimization reduces the complexity related to the inner-loop rotations by an order of magnitude, and consists in two hoisting levels. The *first level* applies to the inner-loop rotations (lines 7 to 10 of Algorithm 14), as proposed by Halevi et al. [13]. This renders the computation dedicated to *Decompose* independent of the value n_1, so the complexity is reduced to

$$\begin{gathered} n_2 \cdot (\mathit{Decompose} + \mathit{MultSum} + \mathit{ModDown} + \mathit{Permute}) \\ + \; n_1 \cdot (\mathit{MultSum} + \mathit{ModDown} + \mathit{Permute}) + \mathit{Decompose}. \end{gathered}$$

The *second level* introduces an additional hoisting for the inner-loop rotations, by observing that the *ModDown* step is a coefficient-wise operation. Thus, this operation commutes with the *Permute* step and the ciphertext-plaintext multiplications (line 8 of Algorithm 14). We therefore apply it only once after the entire inner loop of n_1 rotations is finished, reducing the number of key-switch operations from $n_1 + n_2$ to

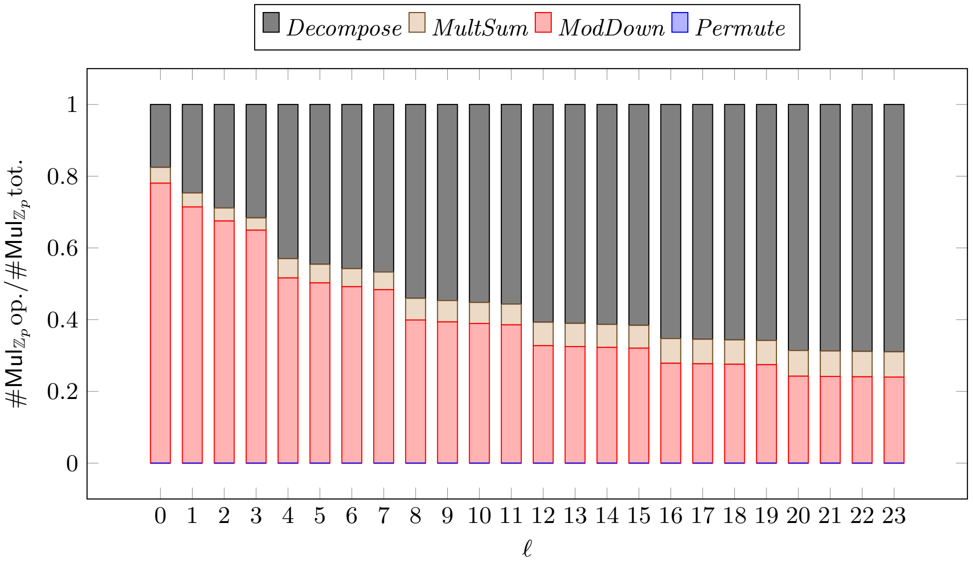


Fig. 3: Normalized complexity of each step (op.) of a rotation. The complexity for each operation was computed with $N = 2^{16}$, $0 \leq \ell \leq 23$ and $\alpha = 4$ using the derivation of Appendix C.2.

$n_2 + 1$. Applying the same reasoning for the *ModDown* step of the outer-loop rotations, the complexity is reduced to:

$$\begin{aligned} &n_2 \cdot (\mathit{Decompose} + \mathit{MultSum} + \mathit{ModDown} + \mathit{Permute}) \\ &+ n_1 \cdot (\mathit{MultSum} + \mathit{Permute}) + \mathit{Decompose} + \mathit{ModDown}. \end{aligned}$$

Our double-hoisting BSGS matrix × vector is described in Algorithm 5.

Remark 3. In line 6 of Algorithm 5, the input c_0 is given in R_{Q_ℓ} and must be extended to $R_{Q_\ell P}$. However, c_0 is also multiplied by P, so all its coefficients in R_P would be zero. Hence, the expensive basis extension can be avoided.

Remark 4. Algorithm 5 is presented with M given in plaintext, but the adaptation to an encrypted M is trivial and would not significantly impact the complexity (as long as M does not need to be manipulated or modified). Indeed, hoisting can also be applied to the relinearization of the ciphertext-ciphertext multiplications by summing the results in $R^3_{Q_\ell P}$ in the *inner loop* and applying the relinearization $R^3_{Q_\ell} \rightarrow R^2_{Q_\ell}$ step only once per *outer loop*.

Discussion In addition to benefiting from our improved key-switch (Section 4.1) and rotation (Section 4.2) procedures, Algorithm 5 introduces a trade-off: The *ModDown* step in the *inner loop* is now dependent on the value n_2, and the *ModDown* step of the *outer loop* only has to be performed once. However, the $2n_1n_2$ multiplications and additions have to be performed in $R_{Q_\ell P}$ instead of R_{Q_ℓ}. Hence, the complexity

Algorithm 5: Double-hoisting BSGS matrix×vector algorithm

```
Input: $\mathsf{ct} = (c_0, c_1) \in R^2_{Q_\ell}$ a ciphertext, $\mathbf{M}_i^{diag} \in R_{Q_\ell P}$ the set of pre-rotated
       and encoded diagonals of $\mathbf{M}$ a plaintext $n \times n$ matrix, $n_1 n_2 = n$,
       $\mathsf{rot}_i \in R^2_{Q_\ell P}$ the set of necessary rotations keys.
Output: The evaluation of $\mathbf{M} \times \mathsf{ct}$.
$\mathbf{d} \leftarrow \mathsf{Decompose}(c_1)$ // $Q_\ell \rightarrow Q_\ell P$
foreach $k_{n_1}$ do
    $a_{k_{n_1}} \leftarrow \phi_{k_{n_1}}(P \cdot c_0 + \sum d^{(i)} \odot \widetilde{\mathsf{rot}}^{(0)}_{k_{n_1}, q_{\alpha_i}})$ // $\in Q_\ell P$
    $b_{k_{n_1}} \leftarrow \phi_{k_{n_1}}(\sum d^{(i)} \odot \widetilde{\mathsf{rot}}^{(1)}_{k_{n_1}, q_{\alpha_i}})$ // $\in Q_\ell P$
end
$r_0, r_1, r_2 \leftarrow (0), (0), (0)$
foreach $k_{n_2}$ do
    $u_0, u_1 \leftarrow (0), (0)$
    foreach $k_{n_1}$ do
        $u_0 \leftarrow u_0 + a_{k_{n_1}} \odot \mathbf{M}^{diag}_{k_{n_1}}$ // $\in Q_\ell P$
        $u_1 \leftarrow u_1 + b_{k_{n_1}} \odot \mathbf{M}^{diag}_{k_{n_1}}$ // $\in Q_\ell P$
    end
    $u_0 \leftarrow \mathsf{ModDownNTT}(u_0)$ // $Q_\ell P \rightarrow Q_\ell$
    $u_1 \leftarrow \mathsf{ModDownNTT}(u_1)$ // $Q_\ell P \rightarrow Q_\ell$
    $\mathbf{d} \leftarrow \mathsf{Decompose}(u_1)$ // $Q_\ell \rightarrow Q_\ell P$
    $r_0 \leftarrow r_0 + \phi_{k_{n_2}}(\sum d^{(i)} \odot \widetilde{\mathsf{rot}}^{(0)}_{k_{n_2}, q_{\alpha_i}})$ // $\in Q_\ell P$
    $r_1 \leftarrow r_1 + \phi_{k_{n_2}}(\sum d^{(i)} \odot \widetilde{\mathsf{rot}}^{(1)}_{k_{n_2}, q_{\alpha_i}})$ // $\in Q_\ell P$
    $r_2 \leftarrow r_2 + \phi_{k_{n_2}}(u_0)$
end
$r_0 \leftarrow \mathsf{ModDownNTT}(r_0)$ // $Q_\ell P \rightarrow Q_\ell$
$r_1 \leftarrow \mathsf{ModDownNTT}(r_1)$ // $Q_\ell P \rightarrow Q_\ell$
return $(r_0 + r_2, r_1)$
```

dependency on the value n_1 is significantly reduced at the cost of slightly increasing the complexity dependency on $n_1 n_2$.

Table 2 compares the complexity of a non-hoisted (for reference), single-hoisted (Algorithm 14) and double-hoisted (Algorithm 5) BSGS, with the optimal ratio n_1/n_2 for each of the approaches. Our approach minimizes the complexity when $2^3 \leq n_1/n_2 \leq 2^4$. This shows that the strategy of the previously proposed bootstrapping procedures [7]–[9], which target $n_1 \approx n_2$ to minimize the number of rotations, is not optimal anymore. The maximum gain happens when n (the number of non zero diagonals) is around 128 and then falls again for smaller values. This behavior can be exploited by factorizing the linear transforms used during the bootstrapping into several sparse matrices (see Section 5.3).

It must be noted that, whereas Algorithm 5 reduces the overall complexity of matrix × vector products, it also induces an increase in the number of used rotation keys. Therefore, it introduces a time-memory trade-off.

	Original		1-hoisted [13]		2-hoisted [Ours]		Speed-
n	n_1/n_2	$\log(\#\mathsf{Mul}_{\mathbb{Z}_p})$	n_1/n_2	$\log(\#\mathsf{Mul}_{\mathbb{Z}_p})$	n_1/n_2	$\log(\#\mathsf{Mul}_{\mathbb{Z}_p})$	up
32768	2	37.276	2	36.913	8	36.813	1.378×
16384	1	36.500	4	36.114	16	35.903	1.512×
8192	2	35.865	2	35.364	8	35.055	1.753×
4096	1	35.152	4	34.648	16	34.205	1.927×
2048	2	34.597	2	33.981	8	33.446	2.219×
1024	1	33.927	4	33.337	16	32.672	2.386×
512	2	33.422	2	32.732	8	32.014	2.653×
256	1	32.769	4	32.137	16	31.318	2.733×
128	2	32.282	2	31.568	8	30.753	2.886×
64	1	31.614	4	30.992	16	30.127	2.804×
32	2	31.112	2	30.430	8	29.637	2.779×
16	1	30.375	4	29.842	16	29.311	2.090×
8	2	29.792	2	29.248	2	29.116	1.596×

Table 2: Complexity comparison between the original Algorithm 14, Algorithm 14 with single hoisted rotations and Algorithm 14 with double hoisted rotations (Algorithm 5). M is a $2^{15} \times 2^{15}$ matrix with $n = n_1 n_2$ non zero diagonals. The used parameters are $N = 2^{16}$, 2^{15} slots, $\ell = 18$ and $\alpha = 4$. The speed-up factor is the ratio between the $\#\mathsf{Mul}_{\mathbb{Z}_p}$ of the original algorithm (1-hoisting was not used in the bootstrapping procedures of [7]–[9]) and our double hoisted algorithm.

5 Bootstrapping for the Full-RNS CKKS Scheme

We present our improved bootstrapping procedure for the full-RNS variant of the CKKS scheme. We follow the high level procedure of Cheon et al. [7] (Section 5.1) and adapt each step, relying on the techniques proposed in Sections 3 and 4.

The purpose of the CKKS bootstrapping [7]–[9], in contrast with the BFV [23] and BGV [24] bootstrapping, is not to reduce the error. Instead, it is meant to reset the ciphertext modulus to a higher level, in order to enable further homomorphic multiplications. The approximate nature of the CKKS scheme, due to the plaintext and ciphertext error being mixed together, implies that each homomorphic operation decreases the output precision. Hence, all the currently proposed bootstrapping circuits only approximate the ideal bootstrapping operation, and their output precision determines their practical utility.

5.1 Circuit Overview

Let $\{\mathsf{ct} = (c_0, c_1), Q_0, \Delta\}$ be a ciphertext that encrypts an n-slot message under a secret-key s with hamming weight h, such that $\mathsf{Decrypt}(\mathsf{ct}, s) = c_0 + sc_1 = \lfloor \Delta \cdot m(Y) \rceil + e \in \mathbb{Z}[Y]/(Y^{2n}+1)$, where $Y = X^{N/2n}$. The bootstrapping operation outputs a ciphertext $\{\mathsf{ct}' = (c_0', c_1'), Q_{L-k}, \Delta\}$ such that $c_0' + sc_1' = \lfloor \Delta \cdot m(Y) \rceil + e' \in \mathbb{Z}[Y]/(Y^{2n}+1)$, where $k < L$ is the number of levels consumed by the bootstrapping and $||e'|| \geq ||e||$ is the error that results from the combination of the initial error e, homomorphic operations, rounding during the rescale and encoding, and function approximations.

The bootstrapping circuit is divided into five steps that we detail below. For the sake of conciseness, we describe the plaintext circuit and omit the error terms.

1. ModRaise: ct is raised to the modulus Q_L by applying the CRT map $R_{q_0} \rightarrow R_{q_0} \times R_{q_1} \times \cdots \times R_{q_L}$. This yields a ciphertext $\{\mathsf{ct}, Q_L, \Delta\}$ for which

   $$[c_0 + sc_1]_{Q_L} = Q_0 \cdot I(X) + \lfloor \Delta \cdot m(Y) \rceil = m',$$

   where $Q_0 \cdot I(X) = \big[-[sc_1]_{Q_0} + sc_1\big]_{Q_L}$ is an integer polynomial for which $||I(X)||$ is $\mathcal{O}(\sqrt{h})$ [7]. The next four steps are aimed at removing this unwanted $Q_0 \cdot I(X)$ polynomial by homomorphically evaluating an approximate modular reduction by Q_0.
2. SubSum: if $2n \neq N$, then $Y \neq X$ and $I(X)$ is not a polynomial in Y. SubSum maps $Q_0 \cdot I(X) + \lfloor \Delta \cdot m(Y) \rceil$ to $(N/2n) \cdot (Q_0 \cdot \tilde{I}(Y) + \lfloor \Delta \cdot m(Y) \rceil)$, a polynomial in Y [7].
3. CoeffsToSlots: The message $m' = Q_0 \cdot \tilde{I}(Y) + \lfloor \Delta \cdot m(Y) \rceil$ is in the *coefficient* domain, i.e. $m' = \Delta \cdot \lfloor (Q_0/\Delta) \cdot \tilde{I}(Y) + m(Y) \rceil$, which prevents slot-wise evaluation of the modular reduction. This step homomorphically evaluates the inverse discrete-Fourier-transform (DFT) and produces a ciphertext encrypting $\mathsf{Encode}(m')$ that enables the slot-wise evaluation of the approximated modular reduction.
   Remark: This step returns two ciphertexts, each encrypting $2n$ real values. If $4n \leq N$, these ciphertexts can be repacked into one. Otherwise, the next step is applied separately on both ciphertexts.
4. EvalSine: The modular reduction $f(x) = x \bmod 1$ is homomorphically evaluated on the ciphertext(s) encrypting $\mathsf{Encode}(m')$. This function is approximated by $\frac{Q_0}{2\pi\Delta} \cdot \sin\big(\frac{2\pi\Delta x}{Q_0}\big)$, which is tight when $Q_0/\Delta \gg ||m(Y)||$. Because the range of x is determined by $||\tilde{I}(Y)||$, the approximation needs to account for the secret-key density.
5. SlotsToCoeffs: This step homomorphically evaluates the DFT on the ciphertext(s) encrypting $f(\mathsf{Encode}(m'))$. It returns a ciphertext at level Q_{L-k} that encrypts $\mathsf{Decode}(f(\mathsf{Encode}(m'))) \approx f(m') \approx \lfloor \Delta \cdot m(Y) \rceil$, which is a close approximation of the original message.

We now detail our approach for each step. We focus on CoeffsToSlots/SlotsToCoeffs and EvalSine, because these steps are the most precision- and performance-critical, and because the latter step requires particular attention when considering dense keys.

5.2 ModRaise and SubSum

We base the ModRaise and SubSum operations directly on the initial bootstrapping of Cheon et al. [7] and provide their respective implementations in Algorithm 11 and Algorithm 12 in Appendix D. The SubSum step multiplies the encrypted message by a factor $N/2n$ that needs to be subsequently cancelled. We take advantage of the following CoeffsToSlot step, which is a linear transformation, to scale the corresponding matrices by $2n/N$. Since we also use this trick for grouping other constants, we elaborate more on the matrices scaling in Appendix E.

5.3 CoeffsToSlots and SlotsToCoeffs

We show how to reduce the algorithmic complexity of the CoeffsToSlots and SlotsToCoeffs operations by applying our matrix-vector multiplication procedure and the double hoisting technique of Section 4.3 to the DFT-matrix×vector product.

Overview Let n be a power-of-two integer such that $1 \leq n < N$; the following holds for any two vectors $\mathbf{m}, \mathbf{m}' \in \mathbb{C}^n$ due to the convolution property of the DFT

$$\mathsf{Decode}_n(\mathsf{Encode}_n(\mathbf{m}) \otimes \mathsf{Encode}_n(\mathbf{m}')) \approx \mathbf{m} \odot \mathbf{m}',$$

where $\otimes$ and $\odot$ respectively denote the nega-cyclic convolution and Hadamard multiplication. That is, the encoding and decoding algorithms define an isomorphism between $\mathbb{R}[Y]/(Y^{2n}+1)$ and $\mathbb{C}^n$ [6]. The goal of the $\mathsf{CoeffsToSlots}$ and $\mathsf{SlotsToCoeffs}$ steps is to homomorphically evaluate this isomorphism on a ciphertext.

Let $\psi = e^{i\pi/n}$ be a $2n$-th primitive root of unity. Since 5 and -1 mod $2n$ span $\mathbb{Z}_{2n}$, $\{\psi^{5^k}, \overline{\psi^{5^k}}, 0 \leq k < n\}$ is the set of all $2n$-th primitive roots of unity. Given a polynomial $m(Y) \in \mathbb{R}[Y]/(Y^{2n}+1)$ with $Y = X^{N/2n}$, the decoding algorithm is defined as the evaluation of this polynomial at each root of unity $\mathsf{Decode}_n(m(Y)) = (m(\psi), m(\psi^5), \ldots, m(\psi^{5^{2n-1}}))$. Let $\mathbf{m}$ be the vector $(m_0, \ldots, m_{n-1}) \in \mathbb{C}^n$, then the encoding algorithm is the inverse operation, which is the interpolation of the vector $\mathbf{m}$ at the points (m_i, ψ^{5^i}) and $(\overline{m_i}, \overline{\psi^{5^i}})$. Thus, the encoding isomorphism is completely defined by the $n \times n$ special Fourier transform matrix

$$\mathrm{SF}_n = \begin{bmatrix} 1 & \psi & \ldots & \psi^{n-2} & \psi^{n-1} \\ 1 & \psi^{5} & \ldots & \psi^{(n-2)\cdot 5} & \psi^{(n-1)\cdot 5} \\ 1 & \psi^{5^2} & \ldots & \psi^{(n-2)\cdot 5^2} & \psi^{(n-1)\cdot 5^2} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & \psi^{5^{n-3}} & \ldots & \psi^{(n-2)\cdot 5^{n-3}} & \psi^{(n-1)\cdot 5^{n-3}} \\ 1 & \psi^{5^{n-2}} & \ldots & \psi^{(n-2)\cdot 5^{n-2}} & \psi^{(n-1)\cdot 5^{n-2}} \\ 1 & \psi^{5^{n-1}} & \ldots & \psi^{(n-2)\cdot 5^{n-1}} & \psi^{(n-1)\cdot 5^{n-1}} \end{bmatrix},$$

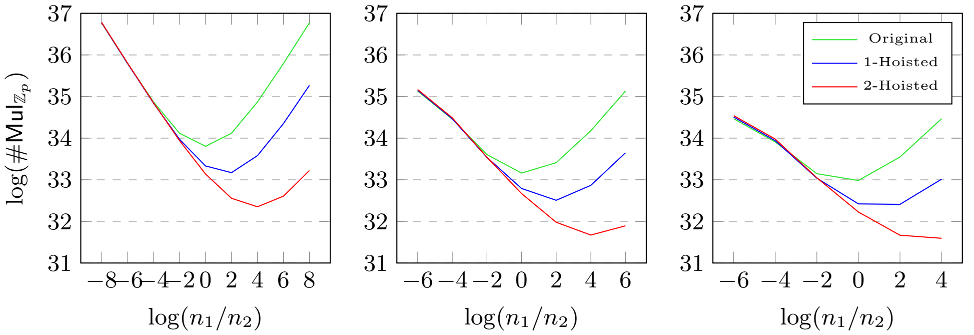
and its inverse $\mathrm{SF}_n^{-1} = \frac{1}{n}\overline{\mathrm{SF}}_n^T$ [14], and their homomorphic evaluation can be expressed in terms of plaintext matrix×vector products:

1. $\mathsf{CoeffsToSlots}(\mathbf{m}) : t_0 = \frac{1}{2}\big(\mathrm{SF}_n^{-1} \times \mathbf{m} + \overline{\mathrm{SF}_n^{-1} \times \mathbf{m}}\big), t_1 = -\frac{1}{2}i(\mathrm{SF}_n^{-1} \times \mathbf{m} - \overline{\mathrm{SF}_n^{-1} \times \mathbf{m}}\big)$
2. $\mathsf{SlotsToCoeffs}(t_0, t_1) : \mathbf{m} = \mathrm{SF}_n \times (t_0 + i \cdot t_1)$.

DFT Evaluation In their initial bootstrapping proposal, Cheon et al. in [7] homomorphically compute the DFT as a matrix-vector product, using a baby-step giant-step (BSGS) approach: Given SF_n represented in diagonal form and two integers n_1 and n_2 such that $n = n_1 n_2$, Algorithm 14 (in Appendix D) evaluates $\mathrm{SF}_n \times \mathbf{m}$ in $n_1 + n_2 \approx 2\sqrt{n}$ rotations and n plaintext multiplications and additions. This algorithm consumes only one level and is much more efficient that the naïve one, which would require n rotations. However, it still remains prohibitive for large n, as it involves a large number of rotations.

To reduce the complexity of Algorithm 14, two recent works from Cheon et al. [14] and Chen et al. [9] exploit the structure of the equivalent FFT algorithm by recursively merging its iterations. In matrix form, both approaches aim at finding appropriate factorizations of SF_n (and its inverse)

$$\mathrm{SF}_n \Longrightarrow \underbrace{\mathrm{M}_0 \times \cdots \times \mathrm{M}_{\rho-1}}_{\text{partially factorized}} \underset{[9],\ [14]}{\Longleftarrow} \underbrace{\mathrm{M}_0 \times \cdots \times \mathrm{M}_{\log(n)-1}}_{\text{fully factorized}}$$



(a) $\mathrm{SF}_n^{-1} = \mathrm{M}_{256} \times \mathrm{M}_{256}$ **(b)** $\mathrm{SF}_n^{-1} = \mathrm{M}_{32} \times \mathrm{M}_{64} \times \mathrm{M}_{64}$ **(c)** $\mathrm{SF}_n^{-1} = \mathrm{M}_{16} \times \mathrm{M}_{32} \times \mathrm{M}_{32} \times \mathrm{M}_{16}$

Fig. 4: Theoretical complexity of CoeffToSlots for different DFT factorizations using Algorithm 14 with no hoisting (original), single hoisting and double hoisting (Algorithm 5).

into ρ matrices. Decreasing ρ reduces the number of consumed levels, but it also results in denser matrices and an increased number of rotations.

We use our variant of the BSGS matrix-vector multiplication approach (see Section 4.3 and Algorithm 5) and combine it with level-merging [9], [14]. This is, we pre-compute a factorization of SF_n (and SF_n^{-1}) into ρ sparse matrices. From the full factorization into $\log(n)$ matrices, we first split the matrices into ρ groups of similar size and then apply the FFT iteration-merging by multiplying among those groups. This ensures that the total complexity is evenly distributed across all the final matrices.

Figure 4 shows the impact of our algorithm on the CoeffsToSlots step, compared with the original BSGS algorithm, for different factorizations corresponding to $\rho_{\mathrm{SF}_n^{-1}} = \{2, 3, 4\}$. The complexity is computed as the number of elemental products, with parameters $N = 2^{16}$, a target $\ell = 17$ (the level after CoeffsToSlots) and $n = 2^{15}$ slots.

It can be seen that each level of hoisting reduces the total complexity by a noticeable amount. The first level of hoisted rotations, as proposed in Section 4.2, changes the parameters n_1, n_2 for which the minimum complexity is achieved to $n_1 \approx 2^2 n_2$ instead of $n_1 \approx n_2$. Using a second level of hoisted rotations further shifts the minimum complexity to $n_1 \approx 2^4 n_2$. On average, our method reduces the complexity of the linear transformations in the bootstrapping by a factor of 2.8×. We will show that, for our best performing parameters, we are able to reduce the run-time of the CoeffsToSlots and SlotsToCoeffs down to that of the EvalSine (see Section 7).

Efficient Repacking of Sparse Plaintexts. The output of the CoeffsToSlots is a ciphertext that encrypts a vector of $\mathbb{C}^n$ values; it cannot be directly fed to the EvalSine, since it contains non-zero imaginary values. To solve this issue, we apply the map $\mathbb{C}^n \rightarrow \mathbb{R}^{2n}$ to the vector. The real values of $\mathbb{C}^n$ are put in the first n slots of the vector in $\mathbb{R}^{2n}$, whereas the imaginary values of $\mathbb{C}^n$ are multiplied by $-i$ and located in the last n slots. During the decoding, the inverse mapping $\mathbb{R}^{2n} \rightarrow \mathbb{C}^n$ is used. We have to apply these operations in the encrypted domain before and after the EvalSine.

This map can be computed with simple operations, e.g. conjugation, multiplication by $-i$ and additions. It outputs two ciphertexts, each encrypting values in $\mathbb{R}^n$. If the original ciphertext is not fully packed ($0 < n < N/2$ slots), the resulting two ciphertexts can be repacked back into one, requiring only one evaluation of EvalSine instead of two.

We observe that decoding a plaintext $\mathbf{m} \in \mathbb{C}^n$ using the decoding algorithm for a plaintext of $\mathbb{C}^{2^k n}$ slots (assuming that $2^k n < N$) outputs a vector comprising 2^k concatenated replicas of $\mathbf{m}$:

$$\mathsf{Encode}_n(\mathbf{m}) = \mathsf{Encode}_{2^k n}(\underbrace{\mathbf{m}|\dots|\mathbf{m}}_{2^k}).$$

Therefore, a ciphertext that encrypts $\mathbf{m} \in \mathbb{C}^n$ can also be seen as a ciphertext encrypting $\mathbf{m}' \in \mathbb{C}^{2n}$ for $\mathbf{m}' = \mathbf{m}||\mathbf{m}$. This property can be used to save two levels when operating the repacking and unpacking ciphertexts before and after the EvalSine.

Repacking Before the EvalSine ($\mathbb{C}^n \rightarrow \mathbb{R}^{2n}$): Repacking into one single ciphertext is done by extending the domain of the plaintext vectors of the last matrix of the CoeffsToSlots step from $\mathbb{C}^n$ to $\mathbb{C}^n||0^n$. Thus, the first n slots are set to zero and can be used to store the imaginary part of the last n slots. The output is therefore a vector of $\mathbb{R}^{2n}$, which is a valid input for the EvalSine. This repacking involves one additional rotation, which is a small cost relative to the whole CoeffsToSlots step, and it does not consume any additional level.

Unpacking After the EvalSine ($\mathbb{R}^{2n} \rightarrow \mathbb{C}^n$): For this operation, we evaluate the following $2n \times 2n$ matrix on the ciphertext

$$\begin{bmatrix} I_n & i \cdot I_n \\ I_n & i \cdot I_n \end{bmatrix},$$

where I_n is the $n \times n$ identity matrix. This matrix has only two non-zero diagonals, and its effect is to homomorphically apply the map $\mathbb{R}^{2n} \rightarrow \mathbb{C}^n||\mathbb{C}^n$. The first n slots of the vector in $\mathbb{R}^{2n}$ are put in the real part of the complex vector in $\mathbb{C}^n$, whereas the last n slots of the real vector in $\mathbb{R}^{2n}$ are put in the imaginary part of the complex vector. Finally, the vector in $\mathbb{C}^n$ is duplicated to form a vector in $\mathbb{C}^{2n} = \mathbb{C}^n||\mathbb{C}^n$, which is a valid encoding of $\mathbb{C}^n$ due to the properties of the encoding algorithm. This additional matrix (transformation) is combined with the first group of the SlotsToCoeffs matrices, slightly increasing its density.

5.4 EvalSine

EvalSine implements the homomorphic modular reduction of the message $\Delta_m \cdot \lfloor (Q_0/\Delta_m) \cdot \tilde{I}(Y) + m(Y) \rceil$ modulo Q_0. The modular reduction is approximated by the function

$$f(x) = \frac{Q_0}{\Delta_m} \frac{1}{2\pi} \sin\big(2\pi x \frac{\Delta_m}{Q_0}\big) \approx \frac{Q_0}{\Delta_m} \cdot \big(\frac{\Delta_m}{Q_0} x \mod 1\big),$$

which scales the message down to $\Delta_m \cdot \lfloor \tilde{I}(Y) + (\Delta_m/Q_0) \cdot m(Y) \rceil$, removes the $\tilde{I}(Y)$ polynomial by taking the message modulo 1, and scales the message up to $\Delta \cdot \lfloor m(Y) \rceil$. Because $\tilde{I}(Y)$ mostly determines the range and degree of the approximation, the EvalSine step is has to take the secret-key density h into account. More specifically, the range of approximation $(-K, K)$ is chosen as $K = \lfloor c(\delta) \cdot \sqrt{h} \rceil$ such that $\Pr[||\tilde{I}(Y)|| > K] \leq 2^{-\delta}$. We elaborate more on how we choose K with respect to h and δ in Section 6.2.

Previous Work Chen et al. [9] directly approximate the function $\frac{1}{2\pi} \cdot \sin(2\pi x)$ using a standard Chebyshev interpolant of degree $d = 119$ in an interval of $(-K, K)$ for $K = 12$ (using a sparse key with $h = 64$), that they evaluate using a modified Paterson-Stockmeyer algorithm. The multiplications by Q_0/Δ_m and Δ_m/Q_0 are respectively performed before and after the evaluation of the Chebyshev interpolant. Han and Ki [8] choose to approximate $\cos(2\pi \frac{1}{2^r}(x - 0.25))$ followed by r iterations of the double angle formula $\cos(2x) = 2\cos(x)^2 - 1$ to approximate $\sin(2\pi x)$, and then multiply the result by $1/2\pi$. The factor $1/2^r$ reduces the range of the approximation to $(-K/2^r, K/2^r)$, enabling the use of a smaller-degree interpolant. They combine it with a specialized Chebyshev interpolation that places the node around the expected intervals of the input, further reducing the degree of the approximation. In their work, they use an interpolant of degree 30 with a scaling factor $r = 2$ (they also use a sparse key with $h = 64$). This results in a much smaller Chebyshev interpolant degree that reduces the total amount of steps to evaluate the polynomial and leads to a faster evaluation. However, their interpolant has a minimum degree of $2K - 1$ and is not always numerically stable, especially if $r = 0$.

Our Work Both methods have $d = \mathcal{O}(K)$, therefore doubling K requires at most doubling d, and the evaluation will require at most one additional level since the Chebyshev interpolant can be evaluated in $\mathcal{O}(\log(K))$ levels. Hence, precision put aside, the level consumption should not be a fundamental problem when evaluating the large degree interpolant required by dense keys. However, the effects of the approximate rescaling procedure (as described in Section 3), if not properly managed, would significantly reduce the output precision. Our EvalSine makes use of our *scale-invariant* polynomial evaluation technique (Section 3).

We propose a new modular reduction function $f(x) = \frac{1}{2\pi} \cdot \sin(2\pi x)$ is approximated by $g_r(x)$ a modified cosine approximation followed by r iteration of the double-angle formula:

$$g_0(x) = \frac{1}{\sqrt[2^r]{2\pi}} \cos\left(2\pi \tfrac{1}{2^r}(x - 0.25)\right) \text{ and } g_{i+1} = 2g_i^2 - \left(\frac{1}{\sqrt[2^r]{2\pi}}\right)^{2^i}.$$

Our technique includes the $1/(2\pi)$ factor directly in the function to approximate, even when using the double angle formula, without consuming an additional level, impacting the precision, or fundamentally changing its evaluation.

The ciphertext must be divided by Q_0/Δ_m before the polynomial evaluation and multiplied by Q_0/Δ_m after. However, Q_0 is an NTT-friendly prime and the value Q_0/Δ_m is not an integer (we assume that Δ_m is a power of two). Hence, extra care must be taken during those scalings to maximize the resulting precision after the bootstrapping. We solve this problem in three steps. We first pre-multiply the ciphertext by a correcting factor $Q_0/2^{\lfloor \log(Q_0) \rceil}$ during the CoeffsToSlots step, then perform the multiplication and division by $\lfloor Q_0/\Delta_m \rceil$ by manipulating the scale during the EvalSine step, and finally correct back the ciphertext with $2^{\lfloor \log(Q_0) \rceil}/Q_0$ during the SlotsToCoeffs step (see Appendix E for further details). With this approach the division is exact and the shape of Q_0 can therefore be chosen to be small and with relatively loose bounds as long as the ratio between Q_0 and the message permits a good approximation of the modular reduction by the sine function.

We also observed that, when considering dense keys, the specialized interpolation method of Han and Ki leads to higher interpolant degree than the standard Chebyshev

```
Algorithm 6: EvalSine
   Input: $\{\mathsf{ct}, Q_\ell, \Delta\}$ a ciphertext, $p(t)$ a Chebyshev interpolant of degree $d$ of
          $f(x) = x \mod 1$, $K$ the range of interpolation, $r$ a scaling factor.
   Output: The scale invariant evaluation $\mathsf{ct}' = \lfloor Q_0/\Delta_m \rceil \cdot p(\lfloor Q_0/\Delta_m \rceil^{-1} \cdot \mathsf{ct})$.
 1 $\Delta \leftarrow \Delta \cdot \lfloor Q_0/\Delta_m \rceil$ // Division by $\lfloor Q_0/\Delta_m \rceil$
 2 $T_0 \leftarrow 1$
 3 $T_1 \leftarrow \mathsf{MultConst}(\mathsf{ct}, 2/(2^{r+1}K))$ // Change of variable and division by $2^r$
 4 $T_1 \leftarrow \mathsf{AddConst}(T_1, -0.5/(2^{r+1}K))$
 5 $T_1 \leftarrow \mathsf{Rescale}(T_1)$
 6 $\mathsf{m} \leftarrow \lceil \log(d+1) \rceil$
 7 $\mathsf{l} \leftarrow \lfloor \mathsf{m}/2 \rfloor$
 8 $T \leftarrow \{T_0, T_1, \dots, T_{2^\mathsf{l}}, \dots, T_{2^{\mathsf{m}-1}}\}$ // Compute the power basis
 9 for $i = 0; i < r; i = i + 1$ do
10     $\Delta \leftarrow \sqrt{\Delta \cdot q_{L-\mathsf{CtS\ depth}-\mathsf{EvalSine\ depth}-r+i}}$ // Compute the appropriate $\Delta$
11 end
12 $\mathsf{ct}' \leftarrow \mathsf{EvalRecurse}(\Delta, \mathsf{m}, \mathsf{l}, p(t), T)$ (Algorithm 2) // Scale invariant
13 $\theta \leftarrow (1/2\pi)^{1/2^r}$
14 for $i = 0; i < r; i = i + 1$ do
15     $\theta \leftarrow \theta^2$
16     $\mathsf{ct}' \leftarrow \mathsf{Mul}(\mathsf{ct}', \mathsf{ct}')$
17     $\mathsf{ct}' \leftarrow \mathsf{Add}(\mathsf{ct}', \mathsf{ct}')$
18     $\mathsf{ct}' \leftarrow \mathsf{AddConst}(\mathsf{ct}', -\theta)$
19     $\mathsf{ct}' \leftarrow \mathsf{Rescale}(\mathsf{ct}')$ // $\Delta \leftarrow \Delta^2/q_{L-\mathsf{CtS\ depth}-\mathsf{EvalSine\ Depth}-i}$
20 end
21 $\Delta \leftarrow \Delta \cdot \lfloor Q_0/\Delta_m \rceil^{-1}$ // Multiplication by $\lfloor Q_0/\Delta_m \rceil$
22 return $\mathsf{ct}'$
```

interpolation. When K is large, the minimum degree of Han and Ki's interpolant for a given precision grows slower than the minimum degree of $2K - 1$ imposed by their method, which is further amplified when using a scaling factor r. Hence, we use the standard Chebyshev interpolation technique when $d \leq 2K - 1$ and the modified technique of Han and Ki otherwise.

Algorithm 6 details our implementation of the EvalSine procedure. Lines 3 and 5 can be omitted if the multiplication by $2/(2^{r+1}K)$ is performed during the CoeffsToSlots step, therefore saving a level. The Δ of the ciphertext after polynomial evaluation (EvalRecurse) takes into account the subsequent evaluations of the double angle formula such that it remains unchanged after the EvalSine step.

6 Parameter Selection

A proper parameterization is paramount to the security and correctness of the bootstrapping procedure. Whereas security is based on traditional hardness assumptions, setting the correctness-related parameters is mostly accomplished through experimental processes to find appropriate trade-offs between performance and the probability of decryption errors. In Sections 6.1–6.2, we discuss various constraints and interdependencies in the parameter selection. Then, we propose a generic procedure to find appropriate parameter sets in Section 6.3.

h	$\log(QP)$			K		
	$\log(QP,N), \lambda = 128$	$N = 2^{15}$	$N = 2^{16}$	K	$\Pr[\lvert x_i\rvert \geq K]$	$K/\sqrt(h)$
64	$0.015121N - 8.248756$	496	982	12	$2^{-38.8}$	1.500
96	$0.018896N - 3.671642$	619	1234	15	$2^{-40.5}$	1.531
128	$0.021370N - 3.601990$	699	1396	17	$2^{-39.4}$	1.502
192	$0.023448N - 3.611940$	767	1533	21	$2^{-41.3}$	1.515
$N/2$	[12]	881	1782	257	$2^{-40.1}$	1.42

Table 3: Modulus size $\log(QP)$ and Sine approximation interval size K for different secret-key densities h (fixed $\lambda = 128$, and $\Pr[|x_i| \geq K] \approx 2^{-40}$).

6.1 Security

Our security parameter selection is driven by the works of Curtis and Player [12], Cheon et al. [11], and Son et al. [10]. Our goal is to select a modulus size for $h = 128$ and $h = 196$ giving us a security estimated to 128 bits. All three works suggest slightly different parameters. Extrapolating the work of Cheon et al. [11], we deduce that the parameters $(N = 65536, \log(Q) = 1010, h = 64)$ would achieve a security of about 128 bits. In a more recent work, Son et al. [10] report that parameters $(65536, 1250, 64)$ provide a security estimated to 113 bits. Both works focus on showing that the parameters that are currently commonly used for the bootstrapping do not meet the security standards, but they do not propose, or show how to select, updated parameters. Conversely, the work of Curtis and Player takes a more general approach and proposes a systematic way to extrapolate the security for large rings while taking the key density into account. We base the choice of the maximum modulus size for 128-bit security on their work. These values are shown in Table 3 for several choices of h. These are slightly more conservative for $h = 64$ than those of Cheon et al. [11] and Son et al. [10].

6.2 Choosing K for EvalSine

The previous works on bootstrapping [7]–[9], [14] use a sparse key with $h = 64$ and $K = 12$. This value $K = 12$ was experimentally determined by Cheon et al. using the heuristic assumption that $||\tilde{I}(Y)|| \approx \mathcal{O}(\sqrt{h})$ [7]. In practice, this value works well as the coefficients of $\tilde{I}(Y)$ hardly ever go above 10. However, this is no longer true for other values of h, for which new heuristic bounds must be found. For this purpose, we conducted the following experiment: For each $h \in [64, 96, 128, 192, 16384]$ we sampled 10^5 encryptions of zero in $Z_{Q_0}[X]/(X^{2^{16}} + 1)$, each encrypted with a freshly sampled secret key; we then decrypted the result in $Q_L > h \cdot Q_0^2$ and finally recorded the distribution of $\tilde{I}(Y)$.

Using a quadratic polynomial approximation we extrapolated the probability distribution obtained by this experiment in Table 3. Our extrapolation shows that for $h = 64$ and $K = 12$ the overflow probability per plaintext coefficient is approximately $2^{-38.8}$. Using the same technique as for $(h = 64, K = 12)$, we validated other values (h, K) that give a similar probability of decryption failure. We observe that taking $K \approx \lfloor 1.5\sqrt{h} \rceil$ is a good approximation for a probability of decryption failure of about 2^{-40} per plaintext coefficient.

Algorithm 7: Heuristic Parameter Selection

```
   Input: $\lambda$ a security parameter.
   Output: The parameters $(N, n, h, Q, P, \alpha, d, r, \rho_{\mathrm{SF}_n^{-1}}, \rho_{\mathrm{SF}_n})$.
1  Select $n$, $N$ and $h$ and derive $\log(QP)$ according to $\lambda$.
2  Select $\Delta_m$ the plaintext scale, $\delta$ the bootstrapping output precision and $Q_0$
     such that $Q_0 \gg \Delta_m$.
3  Compute $K$ from $h$ and find $d$ and $r$ such that the polynomial of the EvalSine
     step in the interval $(-K/2^r, K/2^r)$ and degree $d$ gives a precision of about
     $\log(Q_0/\Delta_m) + \delta$ bits.
4  Select $\rho_{\mathrm{SF}_n^{-1}}$ and $\rho_{\mathrm{SF}_n}$ (the depth of the CoeffsToSlots and SlotsToCoeffs steps).
5  Allocate the $q_j$ of the CoeffsToSlots, EvalSine and SlotsToCoeffs steps, with the
     maximum possible bit-size for all $q_j$.
6  Select $\alpha$ and allocate $P = \prod_{j=0}^{\alpha-1} p_j$, ensuring that $P \approx \beta||q_{\alpha_i}||$.
7  Run the bootstrapping and find the minimum bit-size for the $q_j$ of the EvalSine
     such that the output reaches the desired precision or until it plateaus.
8  Run the bootstrapping and find the minimum bit-size for the $q_j$ of the
     CoeffsToSlots such that the output precision is not affected.
9  Run the bootstrapping and find the minimum bit-size for the $q_j$ of the
     SlotsToCoeffs such that the output precision is not affected.
10 Allocate the rest of the moduli of $Q$ such that $\log(QP)$ ensures a security of $\lambda$
     and check again line 6.
11 If additional residual homomorphic capacity is needed or the security $\lambda$ cannot
     be achieved:
     1. Reduce $\alpha$, $\rho_{\mathrm{SF}_n^{-1}}$ and/or $\rho_{\mathrm{SF}_n}$ and check again line 6.
     2. Increase $h$ to increase $\log(QP)$ and restart at line 1.
     3. Increase $N$ to increase $\log(QP)$ and restart at line 1.

   return $(N, n, h, Q, P, \alpha, d, r, \rho_{\mathrm{SF}_n^{-1}}, \rho_{\mathrm{SF}_n})$
```

6.3 Finding parameters

In this section we describe a general heuristic procedure to select and fine tune bootstrapping parameters. Each operation of the bootstrapping requires a different scaling and a different precision, therefore different moduli. Choosing each modulus optimally for each operation not only leads to a better performance and a better final precision, but also optimizes the bit consumption of each operation and increases the remaining homomorphic capacity after the bootstrapping. We describe our procedure to find suitable parameters for the bootstrapping in Algorithm 7.

Selected Parameters We propose four reference parameter sets, each resulting from following Algorithm 7. The parameter sets were selected for their performance and similarity with the ones in previous works, allowing for an easier comparison. For each set, Table 4 shows the parameters related to the CKKS scheme and to the bootstrapping circuit.

Parameters															
Set	CKKS										Bootstrapping				
	h	N	Δ	$\log(QP)$	L	$\log(q_i)$				$\log(p_j)$	StC & StC		Sine		
						$q_{0\leq i\leq(L-k)}$	StC	Sine	CtS		$\rho_{\mathrm{SF}_n^{-1}}$	ρ_{SF_n}	K	d	r
I	192	2^{16}	2^{45}	1521	24	$55+10\cdot 45$	$56+28$	$8\cdot 55$	$4\cdot 53$	$5\cdot 56$	4	3	21	52	2
II	192		2^{30}	1553	21	$55+7.5\cdot 60$	$1.5\cdot 60$	$8\cdot 55$	$4\cdot 53$	$5\cdot 61$	4	3	21	52	2
III	32768		2^{45}	1782	25	$55+9\cdot 45$	$56+28$	$11\cdot 60$	$4\cdot 53$	$6\cdot 61$	4	3	257	250	3
IV	192	2^{15}	2^{25}	768	14	$35+50+25$	60	$8\cdot 50$	$2\cdot 49$	$2\cdot 50$	2	2	21	52	2

Table 4: The sets of parameters of the full-RNS variant of CKKS used to evaluate the performance of our bootstrapping implementation.

7 Evaluation

We implemented the improved algorithm of Sections 3 and 4, along with the bootstrapping procedure of Section 5 in the Lattigo library [25] and evaluated it using the parameters of Section 6.3. Lattigo is an open-source library that implements the RNS variants of the BFV [20], [23] and CKKS schemes in Golang [26]. All experiments were conducted single threaded on an i5-6600k at 3.5 GHz with 32 GB of RAM running Windows 10 (Go version 1.14.2, GOARCH=amd64, GOOS=windows).

The Bootstrapping Utility Metric While CPU costs are one important aspect when evaluating a bootstrapping procedure, these factors have to be considered together with other performance-related metrics such as the size of the output plaintext space, its precision and the remaining multiplicative depth. In order to evaluate our bootstrapping procedure against the existing ones, we will use the *bootstrapping utility metric*, as introduced by Han and Ki [9] for the same purpose. It is defined as

$$\textit{bootstrapping utility} = \frac{n \times \log(1/\epsilon) \times \log(Q_{L-k})}{\textit{complexity}},$$

where n is the number of plaintext slots, $\log(1/\epsilon)$ is the precision, $\log(Q_{L-k})$ is the size of the remaining coefficient modulus after the bootstrapping (remaining homomorphic capacity) and *complexity* measures the computational cost (in CPU time). The *bootstrapping* utility can also be interpreted as *bootstrapping throughput* in bits/sec.

Note that we chose to express the remaining homomorphic capacity in terms of the modulus size instead of the number of levels, because Q_{L-k} can be re-allocated differently at each bootstrapping call, e.g., a small number of moduli with a large plaintext scale or a large number of moduli with a small plaintext scale.

7.1 Results

We run our benchmarks and compute the bootstrapping utility for each parameter set of Table 4 (Section 6.1) and compare them with the previous works of Chen et al. [9] and Han and Ki [8]. Unfortunately, the implementations of these works have not been publicly released, so we were not able to reproduce their results on our own hardware for a totally fair comparison[5]. The results are summarized in Table 5. Appendix G reports

[5] Chen et al. [8] use an i9-9820X @ 3.3GHz, single-threaded (∼5-10% more instructions per cycle than our bench). Han and Ki [9] use an i7 @ 2.8 GHz, single-threaded.

Bootstrapping Performance														
Set	h	N	n	Timings (sec)						Data (GB)		$\log(Q_{L-k})$	$\mathbf{\log(1/\epsilon)}$	$\boldsymbol{\log(bits/sec)}$
				MU	SS	CtS	StC	Sine	Total	Keys	DFT			
[9]	64	2^{16}	2^{14}	119.8				38.5	158.3	-	-	172	18.6	18.33
[9]	64		2^{12}	127.5				40.4	167.9	-	-	301	20.9	17.22
[8]	64	2^{16}	2^{14}	-	-	-	-	-	52.8	-	-	370	10.8	20.24
[8]	64		2^{10}	-	-	-	-	-	37.6	-	-	370	15.3	17.23
I	192	2^{16}	2^{15}	0.07	0	7.0	3.5	11.5	21.9	15.9	2.8	550	19.8	23.95
I	192		2^{14}	0.07	0.4	7.0	3.8	5.8	16.0	14.9	2.6	550	20.2	23.43
I	192		2^{10}	0.07	2.1	4.8	2.3	5.8	14.9	6.9	1.1	550	**23.2**	19.74
II	192	2^{16}	2^{15}	0.07	0	5.5	2.4	9.3	17.3	14.3	2.5	505	19.2	**24.13**
II	192		2^{14}	0.07	0.4	5.5	2.6	4.7	13.1	13.4	2.2	505	19.2	23.53
II	192		2^{10}	0.07	1.8	4.2	1.5	4.7	12.3	6.22	1.0	505	21.3	19.77
III	32768	2^{16}	2^{15}	0.08	0	8.0	3.2	29.7	41.0	17.5	2.9	460	14.9	22.39
III	32768		2^{14}	0.08	0.5	7.7	3.2	14.8	26.3	16.4	2.7	460	15.2	22.05
III	32768		2^{10}	0.08	2.4	5.7	1.9	14.8	25.0	7.6	1.2	460	18.0	18.37
[9]	64	2^{15}	2^{10}	28.8				9.5	38.3	-	-	150	6.9	14.75
[9]	64		2^{8}	16.9				9.2	26.0	-	-	75	10.03	12.85
[8]	64	2^{15}	2^{2}	-	-	-	-	-	7.5	-	-	185	15.0	10.53
[8]	64		2^{1}	-	-	-	-	-	7.0	-	-	185	16.8	9.79
IV	192	2^{15}	2^{14}	0.02	0	3.7	0.7	2.6	7.1	7.7	2.1	110	15.1	21.87
IV	192		2^{10}	0.02	0.4	1.6	0.4	1.3	3.9	5.1	0.6	110	16.6	18.87

Table 5: Performance comparison of prior bootstrapping in [8], [9] and our proposed bootstrapping for the full-RNS variant of CKKS with parameter sets I, II, III, IV. MU, SS, CtS, StC designate ModUp, SubSum, SlotstoCoeffs, CoeffstoSlots respectively.

on several experiments demonstrating the numerical stability of our bootstrapping procedure.

We observe that, for our best performing parameter set (Set V) we get a bootstrapping throughput 15× larger than the best result reported in the work of Han and Ki, which was implemented using the SEAL library [17] and conducted on similar hardware. Set V also provides a throughput 55.9× larger than the best result reported in the work of Chen et al. [9] which was implemented using the library HEAAN [16]. HEAAN does not implement the full-RNS variant of CKKS, so this second comparison shows the significant performance gains that can be achieved by combining optimized algorithms with a full-RNS implementation. Figure 5 plots our best performing instances against those of the previous works.

We observe that the best results of our bootstrapping consistently happen when the number of slots is set to the maximum (fully-packed plaintexts). The reason is that using 2^{15} slots requires the evaluation of two EvalSine compared to 2^{14} slots, but the complexity of the linear transformations stays nearly the same because no repacking involving a matrix multiplication is needed. Hence, at a computational cost that is only slightly larger than an additional EvalSine and without impacting the precision, we are able to bootstrap twice the number of slots. Moreover, if we look at Set II or Set V, we observe that there is no significant difference for the computational time of the linear transformation across all the different slot values.

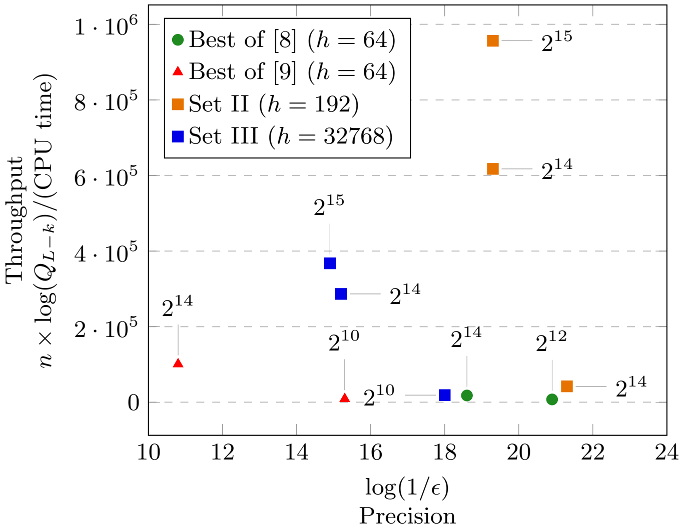


Fig. 5: Bootstrapping utility comparison. We plot the results for our best performing parameter set against the state of the art. Nodes are labeled with n, the number of plaintext slots.

8 Conclusion

In this work, we introduced a practical bootstrapping procedure for the full-RNS CKKS scheme that does not require the use of sparse secret-keys. To the best of our knowledge, this is the first instance of a 128-bit secure bootstrapping.

To achieve this, we proposed a generic algorithm for the homomorphic evaluation of polynomials that is both scale-invariant and optimal in level consumption. In addition to the increase in precision and efficiency, this also improves the usability of the full-RNS variant of CKKS (for which managing a changing scale in large circuits is known to be a difficult task). We also proposed improved key-switch procedures and applied them to the homomorphic matrix×vector multiplication. Our novel *double-hoisted* algorithm reduces the evaluation time of the CoeffsToSlots and SlotsToCoeffs by roughly a factor of 2 compared to the previous works. The performance of these procedures makes them also appealing for applications where the conversion between coefficients and slots domains will enable much more efficient homomorphic circuits (e.g., in the training of convolutional neural networks).

The measured utility of our bootstrapping procedure with "dense" secret-keys ($h = N/2$) is up to 4× larger than the best state-of-the-art results with sparse keys ($h = 64$). When considering the sparse-keys-adjusted parameters of Curtis and Player [12] for $h = 192$ and 128-bits of security, our procedure has a 15× larger utility than the previous works using a sparse key with $h = 64$.

We implement these contributions in the Lattigo library (`https://github.com/ldsec/lattigo`); this is, to the best of our knowledge, the first open-source implementation of a bootstrapping procedure for the Full-RNS variant of the CKKS scheme.

References

[1] C. Gentry, “Fully homomorphic encryption using ideal lattices,” in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

[2] O. Masters, H. Hunt, E. Steffinlongo, J. Crawford, F. Bergamaschi, M. E. D. Rosa, C. C. Quini, C. T. Alves, F. de Souza, and D. G. Ferreira, “Towards a homomorphic machine learning big data pipeline for the financial services sector,” *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 1113, 2019.

[3] M. Kim, A. Harmanci, J.-P. Bossuat, S. Carpov, J. H. Cheon, I. Chillotti, W. Cho, D. Froelicher, N. Gama, M. Georgieva, *et al.*, “Ultra-fast homomorphic encryption models enable secure outsourcing of genotype imputation,” *bioRxiv*, 2020. DOI: 10.1101/2020.07.02.183459.

[4] S. Sav, A. Pyrgelis, J. R. Troncoso-Pastoriza, D. Froelicher, J.-P. Bossuat, J. S. Sousa, and J.-P. Hubaux, “Poseidon: Privacy-preserving federated neural network learning,” *arXiv preprint arXiv:2009.00349*, 2020.

[5] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan, “Homomorphic encryption security standard,” HomomorphicEncryption.org, Toronto, Canada, Tech. Rep., Nov. 2018.

[6] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic encryption for arithmetic of approximate numbers,” in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, 2017, pp. 409–437.

[7] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, “Bootstrapping for approximate homomorphic encryption,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2018, pp. 360–384.

[8] K. Han and D. Ki, “Better bootstrapping for approximate homomorphic encryption,” in *Cryptographers' Track at the RSA Conference*, Springer, 2020, pp. 364–390.

[9] H. Chen, I. Chillotti, and Y. Song, “Improved bootstrapping for approximate homomorphic encryption,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2019, pp. 34–54.

[10] Y. Son and J. H. Cheon, “Revisiting the hybrid attack on sparse and ternary secret LWE,” *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 1019, 2019.

[11] J. H. Cheon, M. Hhan, S. Hong, and Y. Son, “A hybrid of dual and meet-in-the-middle attack on sparse and ternary secret LWE,” *IEEE Access*, vol. 7, pp. 89 497–89 506, 2019.

[12] B. R. Curtis and R. Player, “On the feasibility and impact of standardising sparse-secret LWE parameter sets for homomorphic encryption,” *Proceedings of the 7th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, 2019.

[13] S. Halevi and V. Shoup, "Faster homomorphic linear transformations in HElib," in *Annual International Cryptology Conference*, Springer, 2018, pp. 93–120.

[14] J. H. Cheon, K. Han, and M. Hhan, "Faster homomorphic discrete fourier transforms and improved FHE bootstrapping," *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 1073, 2018.

[15] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full RNS variant of approximate homomorphic encryption," in *International Conference on Selected Areas in Cryptography*, Springer, 2018, pp. 347–368.

[16] *HEAAN*, Online:`https://github.com/snucrypto/HEAAN`.

[17] *Microsoft SEAL (release 3.5)*, Online: `https://github.com/Microsoft/SEAL`, Microsoft Research, Redmond, WA., Apr. 2020.

[18] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *Journal of Mathematical Cryptology*, vol. 9, no. 3, pp. 169–203, 2015.

[19] S. Halevi, Y. Polyakov, and V. Shoup, "An improved RNS variant of the BFV homomorphic encryption scheme," in *Cryptographers' Track at the RSA Conference*, Springer, 2019, pp. 83–105.

[20] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A full RNS variant of FV like somewhat homomorphic encryption schemes," in *International Conference on Selected Areas in Cryptography*, Springer, 2016, pp. 423–442.

[21] S. Halevi and V. Shoup, "Algorithms in HELib," in *Annual Cryptology Conference*, Springer, 2014, pp. 554–571.

[22] S. Halevi and V. Shoup, "Bootstrapping for HELib," in *Annual International conference on the theory and applications of cryptographic techniques*, Springer, 2015, pp. 641–670.

[23] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption.," *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012.

[24] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[25] *Lattigo 2.0.0*, Online: `https://github.com/ldsec/lattigo`, EPFL-LDS, 2020.

[26] *The Go programming language*, Online: `https://golang.org/`, 2020.

A CKKS Homomorphic Operations

This section introduces the available homomorphic operation of the CKKS scheme. Whenever homomorphic operations involve ciphertexts and/or plaintexts whose respective modulus Q_ℓ differ, the operations are carried on with the moduli shared between both operands (i.e. the smallest modulus) and the other moduli are discarded.

- $\mathsf{Add}(\{\mathsf{ct}, Q_\ell, \Delta\}, \{\mathsf{ct}', Q_{\ell'}, \Delta'\})$: Scale ct and ct' to $\max(\Delta, \Delta')$ and return $\{\mathsf{ct}+\mathsf{ct}'$, $\min(Q_\ell,\ Q_{\ell'})$, $\max(\Delta, \Delta')\}$.
- $\mathsf{AddPlain}(\{\mathsf{ct}, Q_\ell, \Delta\}, \{\mathsf{pt}, Q_{\ell'}, \Delta'\})$: Scale ct and pt to $\max(\Delta, \Delta')$ and return $\{\mathsf{ct} + (\mathsf{pt}, 0)$, $\min(Q_\ell,\ Q_{\ell'})$, $\max(\Delta, \Delta')\}$.
- $\mathsf{AddConst}(\{\mathsf{ct}, Q_\ell, \Delta\}, a + bi \in \mathbb{C})$: Return $\{\mathsf{ct} + (\lfloor \Delta \cdot (a + b \cdot X^{N/2})\rceil,\ 0),\ Q_\ell,\ \Delta\}$.
- $\mathsf{Multi}(\{\mathsf{ct}, Q_\ell, \Delta\})$: Returns $\{\mathsf{ct} \cdot X^{N/2},\ Q_\ell, \Delta\}$, the homomorphic product by the imaginary unit.
- $\mathsf{MultConst}(\{\mathsf{ct}, Q_\ell, \Delta\}, a{+}bi \in \mathbb{C}, \Delta')$: Return $\{\lfloor \Delta' a\rceil \cdot \mathsf{ct} + \lfloor \Delta' b\rceil \cdot \mathsf{ct} \cdot X^{N/2}, Q_\ell, \Delta\Delta'\}$.
- $\mathsf{Mul}(\{\mathsf{ct}, Q_\ell, \Delta\}, \{\mathsf{ct}', Q_{\ell'}, \Delta'\})$: For $\mathsf{ct} = (c_0, c_1)$ and ct' $= (c'_0, c'_1)$, compute $(d_0, d_1, d_2) = (c_0c'_0, c_0c'_1 + c_1c'_0, c_1c'_1)$, and return $\{\mathsf{ct}_{\mathsf{mul}} = (d_0, d_1) + \mathsf{SwitchKey}(d_2, \mathsf{rlk})$, $\min(Q_\ell,$ $Q_{\ell'}), \Delta\Delta'\}$.
- $\mathsf{MulPlain}(\{\mathsf{ct}, Q_\ell, \Delta\}, \{\mathsf{pt}, Q_{\ell'}, \Delta'\})$: For $\mathsf{ct} = (c_0, c_1)$, return $\{(c_0{\cdot}\mathsf{pt}, c_1{\cdot}\mathsf{pt})$, $\min(Q_\ell,$ $Q_{\ell'}), \Delta\Delta'\}$.
- $\mathsf{Rescale}(\{\mathsf{ct}, Q_\ell, \Delta\})$: Return $\{\lfloor q_\ell^{-1} \cdot \mathsf{ct}\rceil, Q_{\ell-1}, \Delta/q_\ell\}$, for $q_\ell^{-1} \in \mathbb{R}$.
- $\mathsf{DropLevel}(\{\mathsf{ct}, Q_\ell, \Delta\}, k)$: Return $\{\mathsf{ct}, Q_{\ell-k}, \Delta\}$.
- $\mathsf{Rotate}(\{\mathsf{ct}, Q_\ell, \Delta\}, k)$: For $\mathsf{ct} = (c_0, c_1)$, return $\{\mathsf{ct}_{\mathsf{rot}_k} = (c_0^{5^k}, 0) + \mathsf{SwitchKey}(c_1^{5^k}, \mathsf{rot}_k), Q_\ell, \Delta\}$.
- $\mathsf{Conjugate}(\{\mathsf{ct}, Q_\ell, \Delta\})$: For $\mathsf{ct} = (c_0, c_1)$, return $\{\mathsf{ct}_{\mathsf{conj}} = (c_0^{-1}, 0) + \mathsf{SwitchKey}(c_1^{-1}, \mathsf{conj}), Q_\ell, \Delta\}$.

B Key-switch : Current Approaches

In this section we review the current approaches taken by the state of the art for the key-switch. Given a ciphertext $(c_0, c_1) = (-as' + m + e, a)$ that decrypts under s', the most efficient approach to switch it to s would be to generate public key-switch keys of the form $\mathsf{swk} = (-bs + s' + e', b)$, and perform re-encryption from s' to s as

$$(c_0, 0) + c_1 \cdot \mathsf{swk} = (-abs + ae' + m + e, ab).$$

However, the term ae' would introduce too much error for the ciphertext to be correctly decrypted. Fan and Vercauteren [23] propose two switching-keys types to control this error term:

I. Use $\mathsf{swk}^{(i)} = (-b_i s + w^{(i)} s' + e'_i, b_i)$ for a base $\mathbf{w}$ with the reconstruction formula $x = \sum x_{\mathbf{w}}^{(i)} w^{(i)}$, decompose c_1 under base $\mathbf{w}$ and compute $(c_0, 0) + \sum c_{\mathbf{w},1}^{(i)} \mathsf{swk}^{(i)} = (-a's + \sum a_{\mathbf{w}}^{(i)} e'_i + m + e, a')$. This solution is highly inefficient if the target is to make $\sum a_{\mathbf{w}}^{(i)} e'_i$ small because it will increases the number of keys, and therefore the number of operations, by an amount proportional to $Q/||\mathbf{w}||$.

II. Use $\mathsf{swk} = (-bs + P \cdot s' + e', b)$, for P a large integer, and compute $(c_0, 0) + \lfloor P^{-1} \cdot c_1 \cdot \mathsf{swk} \rceil = (-a's + \lfloor P^{-1} \cdot ae' \rceil + m + e, a')$. If $P \approx ||ae'||$ then the added error is negligible. This solution is more efficient than the type I, but the modulus of the keys is multiplied by P, so the size of the ring degree must be increased or the ciphertext modulus reduced to compensate for the security loss, thus also affecting the overall performances or the homomorphic capacity.

Han and Ki [8] propose a hybrid version that combines both approaches and uses keys of the form $\mathsf{swk}^{(i)} = (-b_i s + w^{(i)} \cdot P \cdot s' + e'_i, b_i)$. Similarly to the type II, if $||\mathbf{w}|| \approx P$, it results in a negligible added error. Moreover, it allows the user to balance the trade-off between the complexity of the first approach and the modulus increase of the second approach. This hybrid solution is well suited for large parameters, as it can greatly reduce the size of the key-switch keys and the complexity of the key-switch operation without impacting much the homomorphic capacity.

While the above high-level description of the key-switch is agnostic of the representation of the coefficients, the base $\mathbf{w}$ must be chosen to be compatible with the latter: When dealing with integers represented in the positional domain, a decomposition with a power of two basis $w^{(i)} = 2^{b^i}$ (i.e., a bit-wise decomposition basis where elements of the decomposed basis are of size at most b bits) is straightforward and efficient to implement using bit-wise arithmetic. Such a decomposition, however, cannot directly be used when dealing with coefficients in the RNS representation, due to its non-linearity. Instead, an alternate base $\mathbf{w}$, derived from the RNS reconstruction, can be used. Similarly to the reconstruction from a power basis: $a = \sum a_{\mathbf{w}}^{(i)} w^{(i)}$, the RNS reconstruction is also a linear operation over a vector, i.e. a sum of products:

$$a \equiv \sum [a]_{q_i} \frac{Q}{q_i} \Big[\Big(\frac{Q}{q_i} \Big)^{-1} \Big]_{q_i} \mod Q. \tag{1}$$

Hence, it can also be used as a decomposition basis and is especially well-suited for dealing with integers represented in the RNS domain, as shown in [19], [20]. On top of this, it is possible to apply an additional power-basis decomposition for each of the elements $[a]_{q_i}$, to further reduce the size of the noise terms, if needed.

C Complexity Analysis

This section contains the complexities derivations of the algorithms used in our work for the Key-switch and hoisted rotation.

C.1 Key-Switch

In this section, we analyse the the complexity of a homomorphic multiplication with a key-switch using our Algorithm 3 (Section **??** in term of its number of modular multiplication in $\mathbb{Z}$ and compare it with the results reported in [8].

We assume that the inputs and outputs of the procedure are both in the NTT domain. We set $\alpha = \#p_j$ and $\beta = \lceil(\ell+1)/\alpha\rceil$, $\mathsf{ct} = (c_0, c_1)$ and $\mathsf{ct}' = (c_0', c_1') \bmod Q_\ell$.

Step 1: *Tensoring*. We compute the tensor product of the ciphertexts (of degree 1) : $(\hat{c}_0, \hat{c}_1, \hat{c}_2) \leftarrow (c_0c_0', c_0c_1' + c_1c_0', c_1c_1') \bmod Q_\ell$. In theory the optimal way would be to use a Karatsuba approach to trade multiplications with additions. However we observed that, due to our Montgomery arithmetic, it was more efficient in our implementation to do 4 multiplications and 1 addition rather than 3 multiplications and 4 additions. The total complexity is therefor $4 \cdot N \cdot (\ell + 1)$.

Step 2: *NTT*. We switch $\hat{c}_2 = d \in R_{Q_\ell}$ out of the NTT domain has complexity $N \cdot \log(N) \cdot (\ell + 1)$.

Step 3: *MultSum*. We decompose d' base q_{α_i}, multiply it with evk and sum. So for $0 < i < \beta$:

1. We apply $\mathsf{Decompose}$ to $d' \in R_{Q_\ell}$: we are given an input vector of $\ell + 1$ elements that we take modulo q_{α_i}, reducing its size to α elements. This first operation is free since $q_{\alpha_i} | Q_\ell$. Using the ModUp algorithm we then extend this vector to a vector of size $(\ell+1)+\alpha$, but for which we already know α elements, so the complexity is $\alpha \cdot (\ell + 1 + \alpha - \alpha) = \alpha \cdot (\ell + 1)$. We also have to run α pre-computations on the fly. Since we have to run this for N values, the total complexity is $N \cdot (\alpha + \alpha \cdot (\ell + 1))$.
2. We switch $d'_{q_{\alpha_i}} \in R_{Q_\ell P}$ back to the NTT domain : we need to compute $(\ell+1)+\alpha$ NTT, but we already have α of those NTT vectors available from $d \in R_{Q_\ell}$, so the total number of NTT is reduced to $\ell+1$, therefor the complexity is $N \cdot \log(N) \cdot (\ell+1)$.
3. We multiply $d_{q_{\alpha_i}} \in R_{Q_\ell P}$ with $\mathsf{evk}^j_{q_{\alpha_i}}$ for $j \in 0, 1$. The complexity is $2 \cdot N \cdot (\ell+1+\alpha)$.

The total complexity of *Step 2* is $\beta \cdot N \cdot ((\ell + 1) \cdot (\alpha + \log(N) + 2) + 3 \cdot \alpha)$.

Step 4: *ModDown*. For $i \in 0, 1$:

1. We switch the P basis of $d_i \in R_{Q_\ell P}$ out of the NTT domain. The complexity is $N \cdot \log(N) \cdot \alpha$.
2. We ModUp $d_i \in R_P$ to change its basis from P to Q_ℓ : we are given a vector of size α and want to extend it to a vector of size $\ell + 1$, but which do not share any moduli with the initial vector. Therefor, and similarly to $\mathsf{Decompose}$ in *Step 2*, the complexity is $N \cdot (\alpha + \alpha \cdot (\ell + 1))$.
3. We switch $d_i \in R_{Q_\ell}$ back to the NTT domain : $N \cdot \log(N) \cdot (\ell + 1)$.
4. The last step of $\mathsf{ModDown}$ is a subtraction followed by a multiplication with P^{-1} : $N \cdot (\ell + 1)$.

The total complexity of *Step 4* is $2 \cdot N \cdot ((\ell+1) \cdot (\alpha + \log(N) + 1) + \alpha \cdot (\log(N) + 1))$.

Step 5. We add the polynomials $d_i \in R_{Q_\ell}$ for $i \in 0, 1$ to the ciphertext, there is no multiplication : $\mathsf{ct}_{\mathsf{mul}} = (d_0 + \hat{c}_0, d_1 + \hat{c}_1) \bmod Q_\ell$.

Hence the total complexity of our homomorphic multiplication in term of modular multiplications is

$$N \cdot \Big((\ell + 1) \cdot \big(\log(N) \cdot (3 + \beta) + \beta \cdot (\alpha + 2) + 2\alpha + 6\big) + \alpha \cdot (2 \cdot \log(N) + 2 + 3\beta)\Big).$$

Remark 5. The complexity of the key-switch itself can be obtained by setting subtracting $4 \cdot N(\ell+1)$ to the homomorphic multiplication complexity.

whereas in [8] they report a complexity of :

$$N \cdot \Big((\ell+1)^2 + (\ell+1) \cdot (\alpha + 2\beta + 6) + 3 + \log(N) \cdot \big((\ell+1) \cdot (\alpha + \beta + 5) + 3 \big) \Big),$$

Table 6 compares both complexities using the same parameters as the original table of [8] with $\ell = 23$ and a variable $\#p_j = \alpha$. The size of the moduli of Q and P is of 45 bits and q_0 is 55 bits. Our tweaked algorithm has the same asymptotic complexity but it introduces a change in the constants which is enough to induces a non negligible difference. The number of NTT, which is the dominant term, is $(\ell+1) \cdot (\alpha+\beta+5)+3$ in [8] while it is $(\ell+1) \cdot (\beta+3) + 2\alpha$ in our work. The number of NTT in our algorithm decreases much faster for larger α than the ones of Han and Ki, e.g. for $\alpha = 6$ it already shows a factor of two difference. Since the number of NTT is the dominant term of the key-switch, this translates into a non-negligible difference in the final complexities.

		log(#Mul in $\mathbb{Z}_p$)	
α	$\log(QP)$	Work in [8]	Our work
1	1136	29.70	29.59
2	1181	29.08	28.83
3	1227	28.84	28.46
4	1272	28.75	28.25
6	1363	28.74	28.02
8	1454	28.82	27.92
12	1635	29.04	27.88
24	2180	29.65	28.08

Table 6: Comparison of the homomorphic multiplication complexity in log.

C.2 (Hoisted) Rotations

In this section we analyse the complexity of hoisted rotations in term of its number of modular multiplication in $\mathbb{Z}$.
Step 1 : NTT. We switch c_1 out of the NTT domain : $N \cdot \log(N) \cdot (\ell+1)$.

Step 2 : Decompose + NTT. We decompose c_1' mod each q_{α_i}, extend the RNS basis from Q_ℓ to $Q_\ell P$ and switch back the result in the NTT domain : $\beta \cdot N \cdot (\alpha + (\ell+1) \cdot (\log(N) + \alpha))$.

Step 3 : MultSum. For each k rotation we multiply d_i^k with $\mathsf{rot}_{k,q_{\alpha_i}}$ and sum : $2k\beta \cdot N \cdot (\ell+1+\alpha)$.

Step 4 : ModDown. For each k rotation we rescale a and b by P and reduce the RNS basis from $Q_\ell P$ back to Q_ℓ: $2k \cdot N \cdot ((\ell+1) \cdot (\alpha + \log(N) + 1) + \alpha \cdot (\log(N) + 1))$.

Step 5 : Permute. For each k rotation we apply the automorphism ϕ_k on $c_0 + a$ and b : there is no multiplication.

Hence the total complexity for k hoisted rotations is :

$$N\cdot\log(N)\cdot\Big((\ell+1)\cdot(\beta+1+2k)+2k\alpha\Big)+N\cdot(\ell+1)\cdot\Big(\beta\alpha+2k\cdot(\alpha+\beta+1)\Big)+N\cdot\alpha\cdot\Big(\beta+2k\cdot(\beta+1)\Big).$$

Remark : the complexity of a single non hoisted rotation can be obtained by setting $k = 1$.

D Algorithms

This section contains the extra algorithms that are referred to but not presented in the main body of this work.

Exact base conversion using floating point arithmetic from Halevi et al. [19]:

$$\mathtt{Conv}^{\mathtt{exact}}_{Q\to P}([a]_Q) = \left(\sum_{j=0}^{\ell-1}[a\cdot\hat{q}_j^{-1}]_{q_j}\cdot\hat{q}_j - v\cdot Q \pmod{p_i}\right)_{0\le i<k}$$

with $\hat{q}_j = Q/q_j$ and where v can be computed with

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

Algorithm 8: ModUp

```
Input: $[a]_{Q_\ell}$ a polynomial with $N$ coefficients, $P$.
Output: $[a]_P$.
for $0 \le i < N$ do
    $[a^{(i)}]_P = \mathtt{Conv}^{\mathtt{exact}}_{Q_\ell\to P}([a^{(i)}]_{Q_\ell})$
end
return $[a^{(i)}]_P$
```

Algorithm 9: Decompose

```
Input: $a \in R_{Q_\ell}$ in the NTT domain, $Q_\ell = \prod_{j=0}^{\ell} q_j$, $q_{\alpha_i} = \prod_{i=\alpha\beta}^{\min(\alpha(\beta+1)-1,\ell)} q_j$
    for $0 \leq i \leq \beta$, $beta = \lceil (\ell+1)/\alpha \rceil$ and $P = \prod_{j=0}^{\alpha-1} p_j$.
Output: $\mathbf{d} \in R_{Q_\ell P}^{\beta}$ in the NTT domain.
$a' \leftarrow \mathsf{NTT}^{-1}(a)$
foreach $q_{\alpha_i}$ do
    $d^{(i)} \leftarrow \mathsf{ModUp}_{q_{\alpha_i} \to Q_\ell P}([a']_{q_{\alpha_i}})$
    foreach $q_j$ do
        if $q_j | q_{\alpha_i}$ then
            $[d^{(i)}]_{q_j} \leftarrow [a]_{q_j}$
        else
            $[d^{(i)}]_{q_j} \leftarrow \mathsf{NTT}([d^{(i)}]_{q_j})$
        end
    end
    foreach $p_j$ do
        $[d^{(i)}]_{p_j} \leftarrow \mathsf{NTT}([d^{(i)}]_{p_j})$
    end
end
return $\mathbf{d}$
```

Algorithm 10: ModDownNTT

```
Input: $a \in R_{Q_\ell P}$ in the NTT domain
Output: $\lfloor P^{-1} \cdot a \rceil \in R_{Q_\ell}$ in the NTT domain
$[b]_P \leftarrow \mathsf{NTT}^{-1}([a]_P)$
$[b]_{Q_\ell} \leftarrow \mathsf{ModUp}_{P \to Q_\ell}([b]_P + \lfloor Q_\ell/2 \rfloor) - \lfloor Q_\ell/2 \rfloor$
$c \leftarrow \mathsf{NTT}([b]_{Q_\ell})$
$[a]_{Q_\ell} \leftarrow [a]_{Q_\ell} - [c]_{Q_\ell}$
return $[P^{-1}]_{Q_\ell} \cdot [a]_{Q_\ell}$
```

Algorithm 11: ModRaise

```
Input: $p$ a polynomial of $N$ coefficients in basis $Q_0 = q_0$.
Output: $p$ the same polynomial in basis $Q_\ell = \prod_{i=0}^{\ell} q_i$, with $p[j][i]$ the $i$-th
        coefficient of the basis $q_j$.
$p \leftarrow \mathsf{NTT}^{-1}(p)$
for $0 \leq i < N$ do
    $x \leftarrow p[0][\mathrm{i}]$
    for $1 \leq j \leq \ell$ do
        if $x \geq \lfloor q_0/2 \rfloor$ then
            $p[j][i] = q_j - (q_0 - x) \mod q_j$
        else
            $p[j][i] = x \mod q_j$
        end
    end
end
$p \leftarrow \mathsf{NTT}(p)$
return $p$
```

Algorithm 12: SubSum

```
Input: ct a ciphertext, $n$ the number of slots of the encoded in the plaintext,
       $N$ the degree of the ring.
for $\log(n) \leq i < \log(N) - 1$ do
    $\mathsf{tmp} \leftarrow \mathsf{Rotate}_{2^i}(\mathsf{ct})$
    $\mathsf{ct} \leftarrow \mathsf{Add}(\mathsf{ct}, \mathsf{tmp})$
end
return
```

Algorithm 13: FFT Algorithm For Evaluating SF_n^{-1}

```
Input: $\mathbf{w} \in \mathbb{C}^n$, $n > 1$ a power of 2 integer, $\Psi$ a pre-computed table of $4n$-th
       roots of unity such that $\Psi^{(j)} = e^{i\pi j/2n}$ for $0 \leq j \leq 2n$.
Output: $\mathrm{SF}_n^{-1} \cdot \mathbf{w}$.
for $m = n; m \geq 2; m = m/2$ do
    for $i = 0; i < n; i = i + m$ do
        for $j = 0; j < m/2; j = j + 1$ do
            $k = 4m - (5^j \bmod 4m) \cdot (n/m)$
            $u = w^{(i+j)} + w^{(i+j+m/2)}$
            $v = w^{(i+j)} - w^{(i+j+m/2)}$
            $w^{(i+j)} = u$
            $w^{(i+j+m/2)} = v \cdot \Psi^{(k)}$
        end
    end
end
bitReverse($\mathbf{w}$, $n$)
for $i = 0; i < n; i = i + 1$ do
    $w^{(i)} = n^{-1} \cdot w^{(i)}$
end
return $\mathbf{w}$
```

Algorithm 14: BSGS Algorithm For Matrix × Vector Multiplication

```
Input: $\mathsf{ct}$ a ciphertext encrypting $\mathbf{m} \in \mathbb{C}^n$, $\mathbf{M}_{diag}$ the diagonal rows of $\mathbf{M}$ a
       $n \times n$ matrix with $n = n_1 n_2$.
Output: The evaluation $\mathsf{ct}' = \mathsf{M} \times \mathsf{ct}$.
foreach $i = 0; i < n_1; i = i + 1$ do
    $\mathsf{ct}_i \leftarrow \mathsf{Rotate}_i(\mathsf{ct})$
end
$\mathsf{ct}' \leftarrow (0, 0)$
foreach $j = 0; j < n_2; j = j + 1$ do
    $\mathbf{r} \leftarrow (0, 0)$
    foreach $i = 0; i < n_1; i = i + 1$ do
        $\mathbf{u} \leftarrow \mathsf{Mul}(\mathsf{ct}_i, \mathsf{Rotate}_{-n_1 \cdot j}(M_{diag}^{(n_1 \cdot j + i)}))$
        $\mathbf{r} \leftarrow \mathsf{Add}(\mathbf{r}, \mathbf{u})$
    end
    $\mathsf{ct}' \leftarrow \mathsf{Add}(\mathsf{ct}', \mathsf{Rotate}_{n_1 \cdot j}(\mathbf{r}))$
end
$\mathsf{ct}' \leftarrow \mathsf{Rescale}(\mathsf{ct}')$
return $\mathsf{ct}'$
```

E Further Optimizations: Details

We recall all the values by which the ciphertext is multiplied before entering the $\mathsf{EvalSine}$ step:

- During the SubSum and $\mathsf{CoeffsToSlots}$ steps, the ciphertext is multiplied respectively by $N/2n$ and $2n$ such that it is, regardless of the number of slots, always multiplied by N, which must be canceled by a $1/N$ constant multiplication.
- We operate the change of variable on the ciphertext to prepare it for the Chebyshev polynomial evaluation, so the ciphertext must be multiplied by $2/(b-a)$.
- We divide the ciphertext by 2^r, where r is the number of iterations of the double angle formula during the $\mathsf{EvalSine}$.
- We multiply the ciphertext by $Q_0/2^{\lceil \log(Q_0) \rceil}$ to compensate for the error introduced by the approximate division by $Q_0/\Delta \approx 2^{10}$.

We merge all the previous constants into one variable called μ_{CtS} and scale the $\mathsf{CoeffsToSlots}$ matrices appropriately to include this multiplication during this step. One could apply this scaling on only one of the matrices, however this will cause precision problems because of how small this scaling factor is. Indeed, if we merge all scaling factors into one, depending on the parameters, we can get values as small as 2^{-21}. In such a case, to retain the same precision as the other matrices, the precision of the scaled matrix, must be increased by 21 bits. This is not always doable because the required scale might become larger than 2^{64} and will prevent an efficient use of the moduli during the rescale. To avoid this problem, we evenly spread μ_{CtS} across the all matrices resulting from the factorization of SF_n^{-1}, and therefor ensure that each matrix is scaled by a small and equivalent amount :

$$\mu_{\mathsf{CtS}} = \left(\frac{2}{(b-a) \cdot N \cdot 2^r} \cdot \frac{Q_0}{2^{\lfloor \log(Q_0) \rceil}} \right)^{\dfrac{1}{\rho_{\mathrm{SF}_n^{-1}}}}$$

where $\rho_{\mathrm{SF}_n^{-1}}$ is the degree of factorization of SF_n^{-1}.

We use the same approach for the multiplication that must occur after the $\mathsf{EvalSine}$ step. In this case we need to :

- Multiply the ciphertext by $2^{\lceil \log(q_0) \rceil}/q_0$ to compensate for the error introduced by the approximate multiplication by $q_0/\Delta \approx 2^{10}$.
- Multiply the ciphertext by Δ/δ where Δ is the scale of the ciphertext after the $\mathsf{EvalSine}$ step and δ is the desired ciphertext output scale.

Therefore, the matrices resulting from the factorization of SF_n (for the $\mathsf{SlotsToCoeffs}$ step) must each be multiplied by:

$$\mu_{\mathsf{StC}} = \left(\frac{\Delta}{\delta} \cdot \frac{2^{\lfloor \log(Q_0) \rceil}}{Q_0} \right)^{\dfrac{1}{\rho_{\mathrm{SF}_n}}},$$

where ρ_{SF_n} is the degree of factorization of SF_n.

Remark 6. To save space and computation, we also pre-rotate the diagonals of the matrices and encode them only at the level they will be used, which reduces the memory footprint. By encoding the matrices with a scale equal to the moduli by which the ciphertext will be rescaled, we ensure that the rescale process will be exact:

$$\mathsf{Rescale}(\mathsf{Mul}(\{\mathsf{ct}, Q_\ell, \Delta\}, \{\mathsf{pt}, Q_\ell, q_\ell\})) = \mathsf{Rescale}(\{\mathsf{ct}', Q_\ell, \Delta \cdot q_\ell\}) = \{\mathsf{ct}', Q_{\ell-1}, \Delta\}$$

F Basic Operations Performances

lvls	α	$\mathsf{Enc}_{\mathsf{pk}}$	$\mathsf{Enc}_{\mathsf{sk}}$	Dec	Add	$\mathsf{Mul}_{\mathsf{pt}}$	$\mathsf{Mul}_{\mathsf{ct}}$	ϕ	$\mathsf{KeySwitch}$	$\mathsf{Rescale}$
29	1	228	128	8	3	9	16	11	1378	71
28	2	227	122	8	3	9	16	10	816	69
27	3	233	120	7	3	9	15	10	596	66
25	5	232	113	6	3	8	15	9	414	62
24	6	237	108	6	3	7	14	8	388	60
20	10	238	93	5	2	6	11	6	294	49
15	15	239	72	4	2	4	8	3	229	36

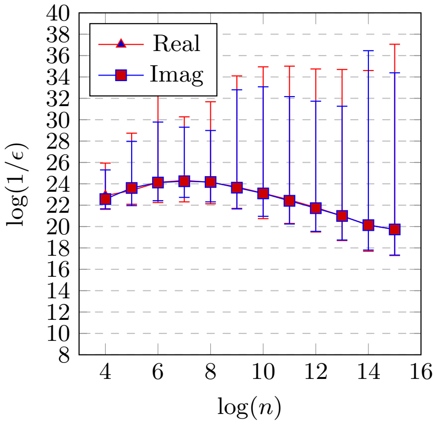
Table 7: Performance in ms of Lattigo for the basic operations for $N = 2^{16}$ and different values of $\mathsf{lvls} = \#q_i$ and $\alpha = \#p_j$ with $\mathsf{lvls} + \alpha = 30$ (so that λ isn't changed when α varies). The timings for the $\mathsf{ct} \times \mathsf{ct}$ multiplication are given without the relinearization (keyswitching). The benchmarks were conducted single threaded on an i5-6600k at 3.5 GHz with 32Gb of RAM running Windows 10 and Go 1.14.2, GOARCH=amd64, GOOS=windows.

G Bootstrapping Stability Experiments

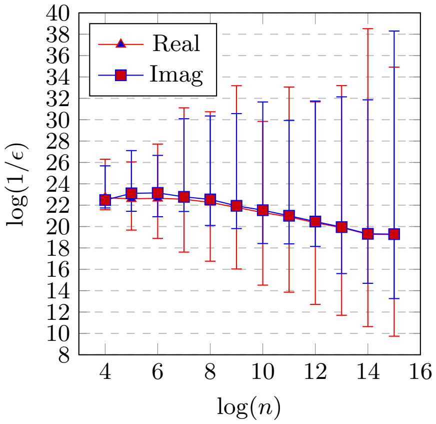
We carried out several experiments to validate the stability of our bootstrapping procedure. Appendix G reports on the following checks: The mean precision across all the slots against the number of slots (Appendix G.1), the probability of each slot to fall under some given precision (Appendix G.2), and the mean precision across all the slots after each bootstrapping for 50 successive bootstrapping (Appendix G.3).

G.1 Precision vs. Slots

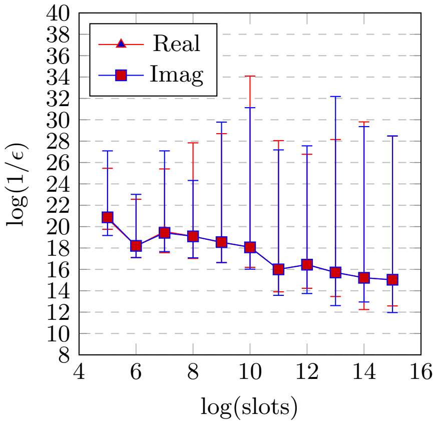
In this section we plot the mean precision across all the slots against the number of slots for the different parameters presented in Section 6.1. The plaintext values that were bootstrapped where of the form $a + bi$ for a, b random reals between -1 and 1. Note that the comparison is made against an unencoded plaintext vector and that those results therefore also include the inherent error of the encoding algorithms which might be much greater than the actual precision of the bootstrapping circuit.



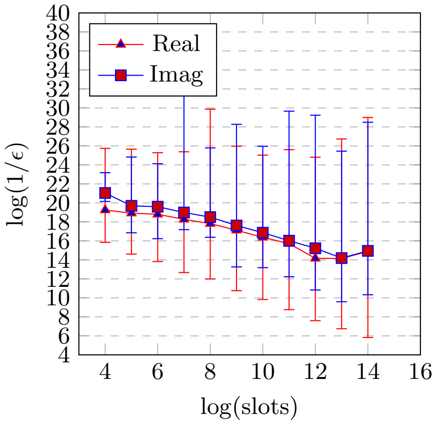
(a) Set I



(b) Set II



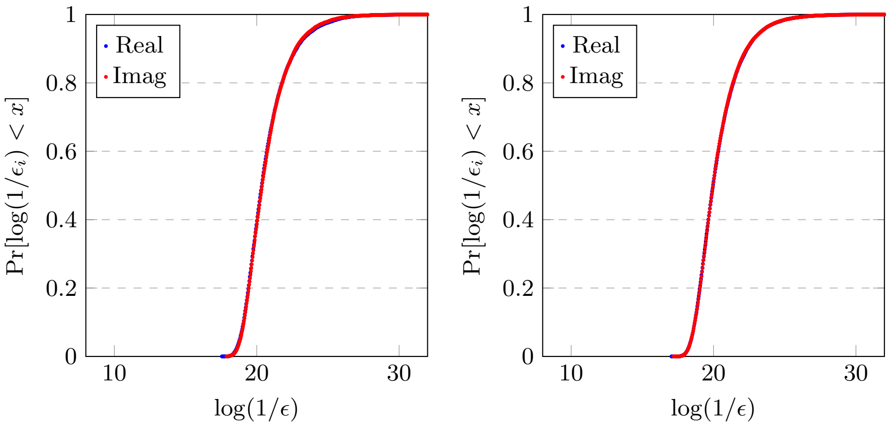
(a) Set III



(b) Set IV

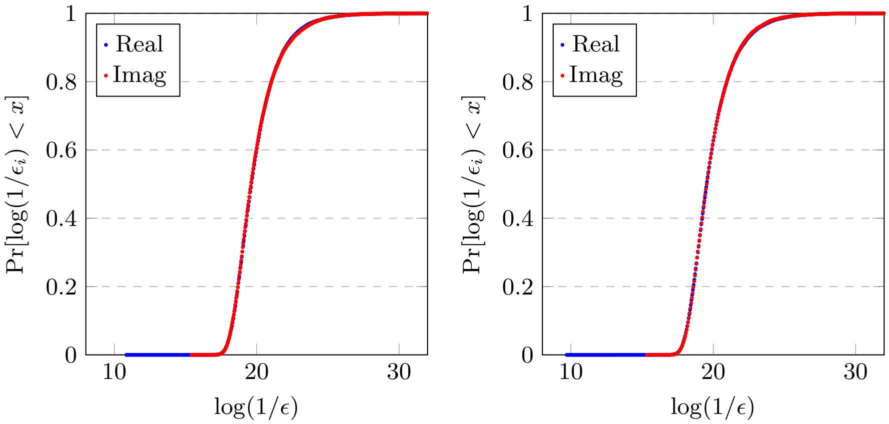
G.2 Precision distribution

In this section we plot the probability of each slot to fall under some given precision for the different parameters presented in Section 6.1. The goal is to show that while we have a good mean precision, the overall distribution also behaves well and is not scattered. The plaintext values that were bootstrapped where of the form $a + bi$ for a, b a random float between -1 and 1. The shape of all the plots show that the distribution is smooth across all the different parameters sets and that very few elements are below or above some threshold that is close to the mean. Note that the comparison is made against an unencoded plaintext vector and that those results therefore also include the inherent error of the encoding algorithms which might be much greater than the actual precision of the bootstrapping circuit.



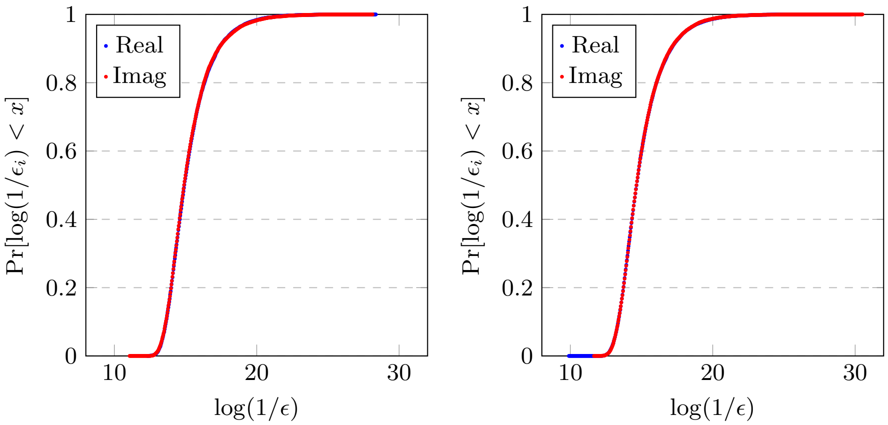
(a) Set I - 2^{14}

(b) Set I - 2^{15}



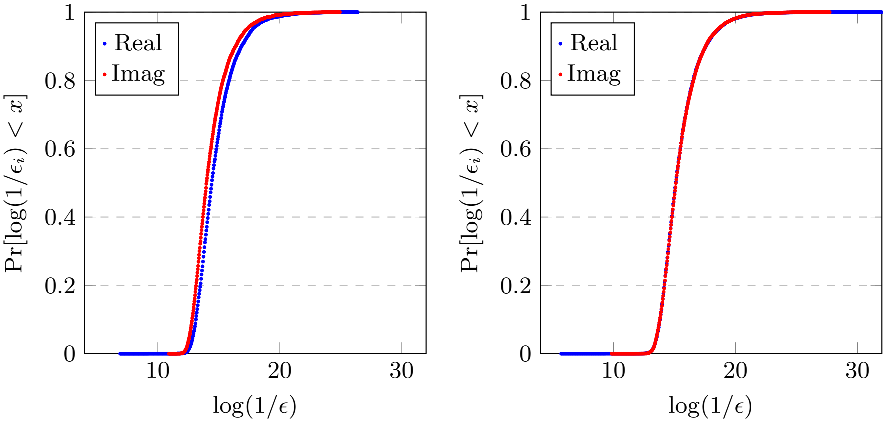
(a) Set II - 2^{14}

(b) Set II - 2^{15}



(a) Set III - 2^{14}

(b) Set III - 2^{15}



(a) Set IV - 2^{13}

(b) Set IV - 2^{14}

G.3 Successive Bootstrappings

In this section we plot the plaintext precision values after each bootstrapping with 50 iterations for the different parameters presented in Section 6.1. Each plaintext was encoding $N/2$ values of the form $a + bi$ for a, b a random float between -1 and 1. The plots show the mean precision along with the absolute upper and lower precision bound (no value had a larger of smaller precision). We observe a logarithmic decrease in the precision that seems consistent with an error with a norm close to the initial precision is added after each iteration. Note that the comparison is made against an unencoded plaintext vector and that those results therefore also include the inherent error of the encoding algorithms which might be much greater than the actual precision of the bootstrapping circuit.

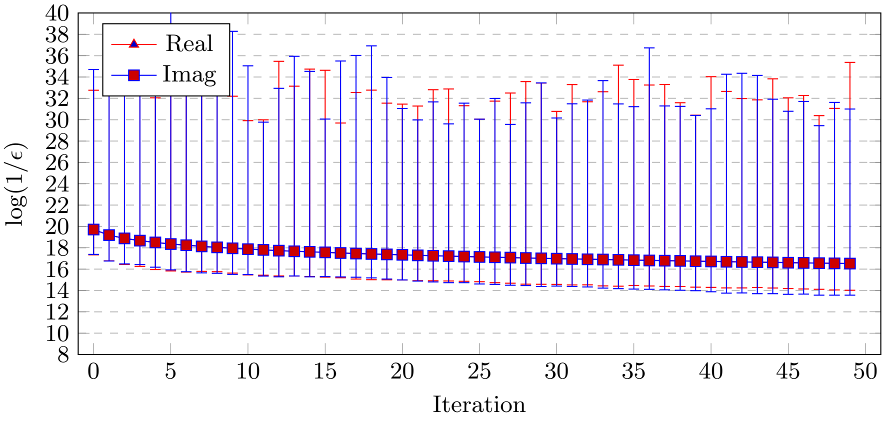


Fig. 12: Set I - 2^{15} slots

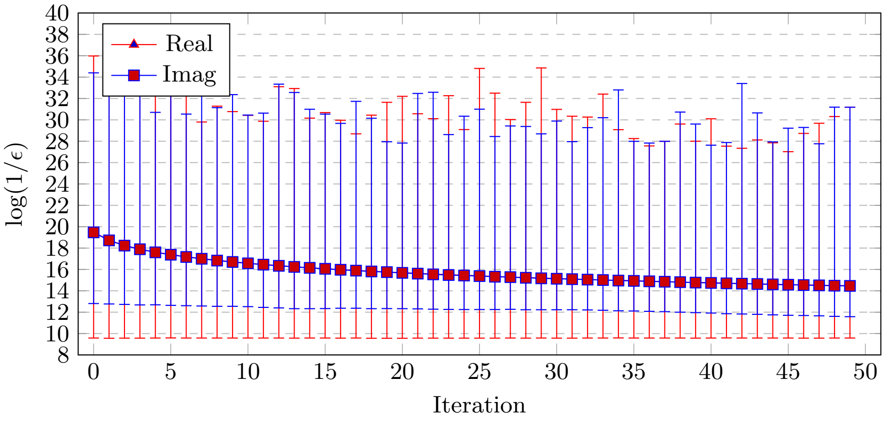


Fig. 13: Set II - 2^{15} slots

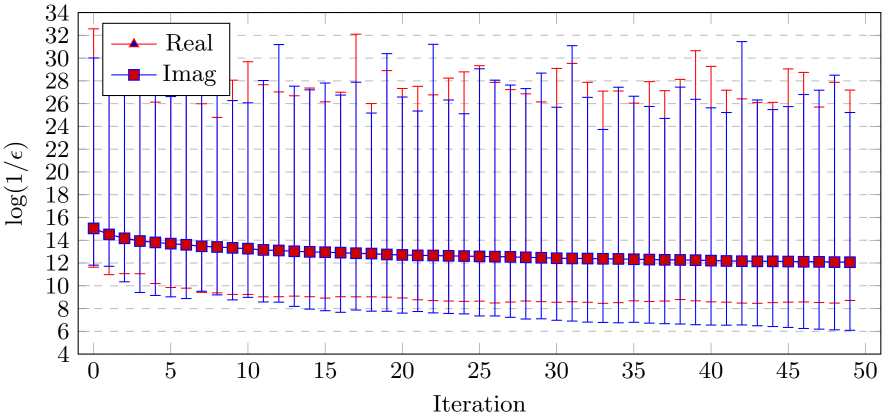


Fig. 14: Set III - 2^{15} slots

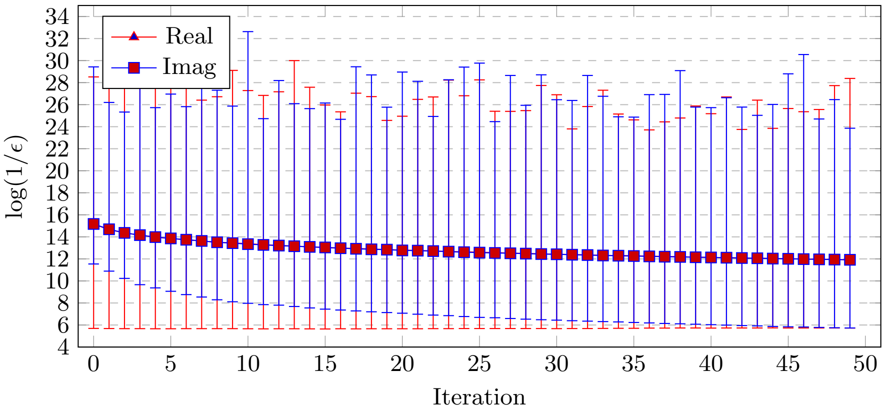


Fig. 15: Set IV - 2^{14} slots