Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-Sparse Keys

Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and
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 procedure is more precise, more efficient (in terms of CPU cost and number of consumed levels), and is more reliable and 128-bit-secure. Unlike the previous approaches, it does not require the use of sparse secret-keys. Therefore, to the best of our knowledge, this is the first procedure that enables a highly efficient and precise bootstrapping with a low probability of failure for parameters that are 128-bit-secure under the most recent attacks on sparse R-LWE secrets.

We achieve this efficiency and precision by introducing three novel contributions: (i) We propose a generic algorithm for homomorphic polynomial-evaluation that takes into account the approximate rescaling and is optimal in level consumption. (ii) We optimize the key-switch procedure and propose a new technique for linear transformations (*double hoisting*). (iii) We propose a systematic approach to parameterize the bootstrapping, including a precise way to assess its failure probability.

We implemented our improvements and bootstrapping procedure in the open-source Lattigo library. For example, bootstrapping a plaintext in $\mathbb{C}^{32768}$ takes 18 seconds, has an output coefficient modulus of 505 bits, a mean precision of 19.1 bits, and a failure probability of $2^{-15.58}$. Hence, we achieve $14.1\times$ improvement in bootstrapped throughput (plaintext-bit per second), with respect to the previous best results, and we have a failure probability $468\times$ smaller and ensure 128-bit security.

Keywords: Fully Homomorphic Encryption · Bootstrapping · Implementation

1 Introduction

Homomorphic encryption (HE) enables computing over encrypted data without decrypting them first; thus, it is becoming increasingly popular as a solution for processing confidential data in untrustworthy environments. Since Gentry's introduction of the first fully homomorphic-encryption (FHE) scheme over ideal lattices [13], continuous efficiency improvements have brought these techniques

closer to practical application domains. As a result, lattice-based FHE schemes are increasingly used in experimental systems [25, 22, 26], and some of them are now proposed as an industry standard [1].

Cheon et al. [10] introduced a *leveled* encryption scheme for approximate arithmetic (CKKS); the scheme is capable of homomorphically evaluating arbitrary polynomial functions over encrypted complex-number vectors. Although the family of *leveled* cryptosystems enables only a finite multiplicative depth, with each multiplication *consuming* one level, the CKKS scheme enables the homomorphic re-encryption of an exhausted ciphertext into an almost *fresh* one. This capability, 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.

Since the initial CKKS bootstrapping procedure by Cheon et al. [9] and until the most recent work by Han and Ki [19] that operates on the full-RNS (residue number systems) version of CKKS, the bootstrapping efficiency has improved by several orders of magnitude. However, this operation remains a bottleneck for its potential applications, and its performance is crucial for the adoption of the scheme. Bootstrapping performance can be improved by following two approaches: (i) adapting the bootstrapping circuit representation by using HE-friendly numerical methods. (ii) optimizing the scheme operations themselves, which also improves the overall scheme performance.

All current CKKS bootstrapping approaches [9, 5, 19] rely, so far, on sparse secret-keys to reduce the depth of their circuit representation, and none of them has proposed parameters with an equivalent security of at least 128 bits under the recent attacks on sparse R-LWE secrets [8, 28]. The lack of stability in the security of sparse R-LWE secrets has lead the standardization initiatives to exclude sparse keys, hence also the bootstrapping operation, from the currently proposed standards [1]. This raises the question about the practicality of a bootstrapping procedure that would not require the use of sparse secret-keys.

1.1   Our Results

We propose an efficient bootstrapping procedure for the full-RNS CKKS scheme; it does not necessarily require the use of sparse secret-keys and provides a greater throughput than the current state of the art (Definition 1 in Section 7.1). To achieve this, we make the following contributions:

Homomorphic Polynomial Evaluation (Section 3). The full-RNS variant of the CKKS scheme restricts the re-scale operation only to the division by the factors q_i of the ciphertext modulus Q. As the choice of these factors is constrained to those enabling a number theoretic transform (NTT), the rescale cannot be done by a power of two (as in the original CKKS scheme) and it introduces a small scale deviation in the process. For complex circuits, such as polynomial evaluations, additions between ciphertexts of slightly different scales will eventually occur and will introduce errors.

We observe that this problem is trivially solved for linear circuits, by scaling the plaintext constants by the modulus q_i by which the ciphertext will be divided during the next rescale. By doing so, the rescale is exact and the ciphertext scale is unchanged after the operation. As a polynomial can be computed by recursive evaluations of a linear function, the linear case can be generalized. In this work, we propose a generic algorithm that consumes an optimal number of $\lceil \log(d+1) \rceil$ levels to homomorphically evaluate degree-d polynomial functions. Starting from a user-defined output scale, the intermediate scales can be back-propagated in the recursion, thus ensuring that each and every homomorphic addition occurs between ciphertexts of the same scale (hence is errorless). Our algorithm 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.

Faster Matrix × Ciphertext Products (Section 4). The most expensive CKKS homomorphic operation is the key-switch. This operation is an integral building block of the homomorphic multiplication, slot rotations, and conjugation. 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 has a significant effect on its performance.

Given an $n \times n$ plaintext matrix M and an encrypted vector $\mathbf{v}$, all previous works on the CKKS bootstrapping [9, 5, 19] use a baby-step giant-step (BSGS) algorithm, proposed by Halevi and Shoup [17], to compute the encrypted product $M\mathbf{v}$ in $\mathcal{O}(\sqrt{n})$ rotations. These works treat the key-switch procedure as a black-box and try to reduce the number of times it is executed. Therefore, they do not exploit the *hoisting* proposed by Halevi and Shoup [18].

We improve this BSGS algorithm by proposing a new format for rotation keys and a modified key-switch procedure that extends the hoisting technique to a second layer. This strategy is generic and it reduces the theoretical minimum complexity (in terms of modular products) of any linear transformation over ciphertexts. In our bootstrapping it reduces the cost of the linear transformations by roughly a factor of *two* compared to the previous hoisting approach.

Improved Bootstrapping Procedure (Section 5). We integrate our proposed improvements in the bootstrapping circuit proposed by Cheon et al. [9], Chen et al. [5], Cheon et al. [6] and Han and Ki [19]. We propose a new high-precision and faster bootstrapping circuit with updated parameters that are 128-bit secure, even if considering the most recent attacks on sparse keys [8, 28].

Parameterization and Evaluation (Section 6). We discuss the parametrization of the CKKS scheme and its bootstrapping circuit, and we propose a procedure to choose and fine-tune the parameters for a given use-case.

We implemented our contributions, as well as our bootstrapping, 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 [10] in its full-RNS variant [7]: the polynomial coefficients are always represented in the RNS and NTT domains.

Notation For a fixed power-of-two N and $L + 1$ distinct primes $q_0, \ldots, q_L$, 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 modulo Q_L. Unless otherwise stated, we consider elements of R_{Q_L} as their unique representative in the RNS domain: $R_{q_0} \times R_{q_1} \times \ldots \times R_{q_L} \cong R_{Q_L}$: a polynomial in R_{Q_L} is represented by a $(L+1) \times N$ matrix of coefficients. We denote single elements (polynomials or numbers) in italics, 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 element at position i 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 in the power basis and $\mathsf{hw}(a)$ the Hamming weight of the polynomial (or vector) a. We denote $\langle \mathbf{a}, \mathbf{b} \rangle$ the inner product between the vectors $\mathbf{a}$ and $\mathbf{b}$. Given two vectors $\mathbf{a}$ and $\mathbf{b}$, each of n values, we denote $\log(\epsilon^{-1})$ the negative log of the L1 norm of their difference: $\epsilon = \frac{1}{n}\sum_{i=0}^{n-1} |a^{(i)} - b^{(i)}|$. $[x]_Q$ denotes reduction of x modulo Q and $\lfloor x \rfloor, \lceil x \rceil, \lfloor x \rceil$ the rounding of x to the previous, the next, and the closest integer, respectively (if x is a polynomial, the operation is applied coefficient-wise). Unless otherwise stated, logarithms are in base 2.

Plaintext and Ciphertext Space A plaintext is a polynomial $\mathsf{pt} = m(Y) \in \mathbb{R}[Y]/(Y^{2n} + 1)$ with $Y = X^{N/2n}$ and n a power-of-two smaller than N. We define the following plaintext encodings: (i) The *coefficient* encoding for which the message $\mathbf{m} \in \mathbb{R}^{2n}$ is directly encoded as the coefficients of a polynomial in Y. (ii) The *slots* encoding for which the message $\mathbf{m} \in \mathbb{C}^n$ is subjected to the canonical embedding $\mathbb{C}^n \to Y^{2n}$ for which the negacyclic convolution in $\mathbb{R}[Y]/(Y^{2n} + 1)$ results in a Hadamard product in $\mathbb{C}^n$.

We represent plaintexts and ciphertexts, respectively, 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 $R_{Q_\ell}^2$. 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 and use $0 \leq \ell \leq L$ to represent a 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

- Setup(N, h, b, σ): For a power-of-two ring degree N, a secret-distribution Hamming weight h, a 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 (i.e. $q_i \equiv 1 \mod 2N$) close to powers of two such that $\log(\prod_{i=0}^{L} q_i \times \prod_{j=0}^{\alpha-1} p_j) \leq b$. 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} with coefficients uniformly distributed over $\{-1, 0, 1\}$ and exactly h non-zero coefficients. χ_{pkenc} with coefficients distributed over $\{-1, 0, 1\}$ with respective probabilities $\{1/4, 1/2, 1/4\}$. χ_{err} with coefficients distributed according to a discrete Gaussian distribution with standard deviation σ and truncated to $[-\lfloor 6\sigma \rfloor, \lfloor 6\sigma \rfloor]$.

- 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 $\mathbb{C}^n \to \mathbb{R}[Y]/(Y^{2n}+1) \to R_{Q_\ell}$ with $Y = X^{N/2n}$. Compute $\mathbf{m}' = \mathrm{FFT}_n^{-1}(\mathbf{m})$ and set $\mathbf{m}'_0 \| \mathbf{m}'_1 \in \mathbb{R}^{2n}$, 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, scale the coefficients by Δ and round them to the nearest integer, apply the change of variable $Y \to X$ and return $\{\mathsf{pt}, Q_\ell, \Delta\}$.

- 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 $R_{Q_\ell} \to \mathbb{R}[Y]/(Y^{2n} + 1) \to \mathbb{C}^n$, with $Y = X^{N/2n}$. Map pt to the vector $\mathbf{m}'_0 \| \mathbf{m}'_1 \in \mathbb{R}^{2n}$ and return $\mathbf{m} = \mathrm{FFT}_n(\Delta^{-1} \cdot (\mathbf{m}'_0 + i \cdot \mathbf{m}'_1))$.

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

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

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

- 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 $\{\mathsf{ct}, Q_\ell, \Delta\}$.

- 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} = $ SwitchKey$(u, \mathsf{pk}) + (\mathsf{pt} + e_0, e_1)$ and return $\{\mathsf{ct}, Q_\ell, \Delta\}$.

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

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

The homomorphic operations of CKKS are detailed in Supplementary material A.

---

[1] 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.

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 $[c'_0 + c'_1 s]_{Q_{L-k}} \approx \mathsf{pt}$. Since $[c_0 + c_1 s]_{Q_L} = \mathsf{pt} + Q_0 \cdot I$, where I is an integer polynomial [9], bootstrapping is equivalent to an extension of the CRT basis, followed by a homomorphic reduction modulo Q_0.

   Cheon et al. proposed the first procedure [9] 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:

$$\underbrace{\mathsf{Encode}(\mathsf{pt} + Q_0 \cdot I) = \mathsf{pt}'}_{\text{(i) } \mathsf{SlotsToCoeffs}(\mathsf{pt}+Q_0\cdot I)} \Rightarrow \underbrace{\frac{Q_0}{2\pi} \sin\left(\frac{2\pi\mathsf{pt}'}{Q_0}\right) = \mathsf{pt}''}_{\text{(ii) } \mathsf{EvalSine}(\mathsf{pt}')} \Rightarrow \underbrace{\mathsf{Decode}(\mathsf{pt}'') \approx \mathsf{pt}}_{\text{(iii) } \mathsf{CoeffsToSlots}(\mathsf{pt}'')} .$$

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 $||I|| \le \mathcal{O}(\sqrt{h})$ 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 that 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 by using a low-degree Taylor series of the complex exponential and then use repeated squaring (the double angle formula) to obtain the correct result. In their approach, the sine evaluation dominates the circuit's depth, whereas the homomorphic evaluation of the encoding and decoding algorithms, which they express as an $n \times n$ matrix-vector product, dominates its width.

   In a subsequent work, Chen et al. [5] propose to compute the encoding by homomorphically evaluating the Cooley-Tukey algorithm. This approach needs $\log(n)$ depth (the number of iterations of the algorithm); to reduce this depth, Chen et al. merge several iterations together, at the cost of an increased complexity. In a concurrent work, Cheon et al. [6] 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 [5, 6] enabled the acceleration of the homomorphic evaluation of the encoding functions by two orders of magnitude. Chen et al. [5] also improved the approximation of the scaled sine function by using a Chebyshev interpolant.

   More recently, Han and Ki port the bootstrapping procedure to the full-RNS variant of CKKS, with several improvements to the bootstrapping circuit and to the CKKS scheme [19]. They propose a generalization of its key-switch procedure by using an intermediate RNS decomposition that enables a trade-off

between the complexity of the key-switch and the homomorphic capacity of a fresh ciphertext. They also give an alternative way to approximate the scaled sine function, which accounts for the magnitude of the underlying plaintext and uses the cosine function and the double angle formula. Combined, these changes yield an acceleration factor of 2.5 to 3, compared to the work of Chen et al. [5].

Both works [6, 5] were implemented with HEAAN [20], yet the implementation of only the former was published. The work of [19] was implemented using SEAL [27], but the implementation has still not been published.

2.3 Security of Sparse Keys

One commonality between all the aforementioned works is the use of sparse secret-keys with a Hamming weight $h = 64$. A key with a small Hamming weight enables a low-depth bootstrapping circuit, essential for its practicality. However, recent advances in the cryptanalysis of the R-LWE problem prove that hybrid attacks specifically targeting such sparse keys can severely affect its security [8, 28]. In light of the most recent attacks, Curtis and Player [11] estimate that, for a sparse key with $h = 64$ and a ring degree $N = 2^{16}$, the modulus needs to be at most 990 bits to achieve a security of 128 bits. In their initial bootstrapping proposal, Cheon et al. [9] use the parameters $\{N = 2^{16}, \log(Q) = 2480, h = 64, \sigma = 3.2\}$ and estimate the security of these parameters to 80 bits. In their work, Han and Ki [19] propose new parameter sets, one of which they claim has 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 [2] that, at the time, did not take into account the most recent attacks on sparse keys. The security of the parameter set $\{N = 2^{16}, \log(Q) = 1250, h = 64, \sigma = 3.2\}$ is estimated at 113 bits in the more recent work by Son and Cheon [28]. This sets a loose upper bound to security of the parameters (which have a 1450-bit modulus) proposed by Han and Ki [19]. Therefore, the bootstrapping parameters must be updated to comply with the most recent security recommendations, as none of the parameters proposed in the current works achieve a security of 128 bits.

3 Homomorphic Polynomial Evaluation

The main disadvantage of the full-RNS variant of CKKS stems from its rescale operation that 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 [7]; under this constraint, the power-of-two rescale of the original CKKS scheme can only be approximated. As a result, ciphertexts at the same level can have slightly different scales (depending on the previous homomorphic operations) and additions between such ciphertexts will introduce an error proportional to the difference between their scale. Addressing this issue in a generic and practical way is crucial for the adoption of CKKS.

For a significant step toward this goal, we introduce a homomorphic polynomial-evaluation algorithm that is depth-optimal and ensures that additions are always made between ciphertexts with the exact same scale.

Algorithm 1: BSGS alg. for polynomials in Chebyshev basis

Input: $p(t) = \sum_{i=0}^{d} c_i T_i(t)$.
Output: The evaluation of $p(t)$.

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 $T_2(t), T_3(t), \ldots, T_{2^{\mathsf{l}}-1}(t)$ and $T_{2^{\mathsf{l}}}(t), T_{2^{\mathsf{l}+1}}(t), \ldots, T_{2^{\mathsf{m}-1}}(t)$ using
$T_{i=a+b}(t) \leftarrow 2T_a(t)T_b(t) - T_{|a-b|}(t)$.

5 Find $q(t)$ and $r(t)$ such that $p(t) = q(t) \cdot T_{2^{\mathsf{m}-1}}(t) + r(t)$.

6 Recurse on step 5 by replacing $p(t)$ by $q(t)$ and $r(t)$ and m by $\mathsf{m} - 1$, until the degree of $q(t)$ and $r(t)$ is smaller than 2^{l}.

7 Evaluate $q(t)$ and $r(t)$ using $T_j(t)$ for $0 \leq j \leq 2^{\mathsf{l}} - 1$.

8 Evaluate $p(t)$ using $q(t)$, $r(t)$ and $T_{2^{\mathsf{m}-1}}(t)$.

9 **return** $p(t)$

3.1 The Baby-Step Giant-Step (BSGS) Algorithm

In order to minimize the number of ciphertext-ciphertext multiplications in their bootstrapping circuit, Han and Ki [19] adapt a generic baby-step giant-step (BSGS) polynomial-evaluation algorithm for polynomials expressed in a Chebyshev basis. 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 $\mathsf{l} = \lfloor \mathsf{m}/2 \rfloor$, the algorithm first decomposes $p(t)$ into $\sum_{i=0}^{\lfloor d/\mathsf{l} \rfloor} u_{i,2^{\mathsf{l}}}(t) \cdot T_{2^{i \cdot \mathsf{l}}}(t)$, with $u_{i,2^{\mathsf{l}}}(t) = \sum_{j=0}^{2^{\mathsf{l}}-1} c_{i,j} \cdot T_j(t)$, $c_{i,j} \in \mathbb{C}$ and $T_{0 \leq j < 2^{\mathsf{l}}}$ a pre-computed power basis. We denote $u_{\lfloor d/\mathsf{l} \rfloor, 2^{\mathsf{l}}}(t)$ as u_{max}. The BSGS algorithm then recursively combines the monomials $u_{i,2^{j+1}}(t) = u_{i+1,2^j}(t) \cdot T_{2^j}(t) + u_{i,2^j}(t)$ in a tree-like manner by using a second pre-computed power basis $T_{2^{\mathsf{l} \leq i < \mathsf{m}}}(t)$ to minimize the number of non-scalar multiplications. The algorithm requires $2^{\mathsf{m}-\mathsf{l}} + 2^{\mathsf{l}} + \mathsf{m} - \mathsf{l} - 3 + \lceil (d+1)/2^{\mathsf{l}} \rceil$ non-scalar products and has, in the best case, depth m.

3.2 Errorless Polynomial Evaluation

We address the errors introduced by the approximate rescale for the evaluation of a polynomial $p(t)$. We scale each of the leaf monomials $u_{i,2^{\mathsf{l}}}(t)$ by some scale Δ such that all evaluations of the subsequent monomials $u_{i,2^{j+1}}(t) = u_{i+1,2^j}(t) \cdot T_{2^j}(t) + u_{i,2^j}(t)$ are done with additions between ciphertexts of the same scale. More formally, let $\Delta_{u_{i,2^{j+1}}(t)}$ be the scale of $u_{i,2^{j+1}}(t)$ (the result of the monomial evaluation), $\Delta_{T_{2^j}(t)}$ the scale of the power-basis element $T_{2^j}(t)$, and $q_{T_{2^j}(t)}$ the modulus by which the product $u_{i+1,2^j}(t) \cdot T_{2^j}(t)$ is rescaled. We set $\Delta_{u_{i+1,2^j}(t)} = \Delta_{u_{i,2^{j+1}}(t)} \cdot q_{T_{2^j}(t)}/\Delta_{T_{2^j}(t)}$ and $\Delta_{u_{i,2^j}(t)} = \Delta_{u_{i,2^{j+1}}(t)}$. Starting from a target scale $\Delta_{p(t)}$ and $p(t) = u_{0,2^{\mathsf{m}}}(t) = u_{1,2^{\mathsf{m}-1}}(t) \cdot T_{2^{\mathsf{m}-1}}(t) + u_{0,2^{\mathsf{m}-1}}(t)$, we recursively compute and propagate down the tree the scale each $u_{i,2^j}(t)$ should have. The recursion ends when reaching $u_{i,2^{\mathsf{l}}}$, knowing the scale that they must have. Since $u_{i,2^{\mathsf{l}}}(t) = \sum_{j=0}^{2^{\mathsf{l}}-1} c_{i,j} T_j(t)$, we can use the same technique to derive by what value

---

Algorithm 2: EvalRecurse

Input: A target scale Δ, an upper-bound m, a stop factor l, a degree-d
polynomial $p(t) = \sum_{i=0}^{d} c_i T_i(t)$, and the power basis $\{T_0, T_1, \ldots, T_{2^{l-1}}\}$
and $\{T_{2^l}, T_{2^{l+1}}, \ldots, T_{2^{m-1}}\}$, pre-computed for a ciphertext ct.

Output: A ciphertext encrypting the evaluation of $p(\mathsf{ct})$.

1 **if** $d < 2^l$ **then**

2   **if** $p(t) = u_{max}(t)$ **and** $l > 2^m$ - 2^{l-1} **and** $l > 1$ **then**

3   **return** EvalRecurse$(\Delta, m = \lceil \log(d+1) \rceil, l = \lfloor \lceil \log(d+1) \rceil/2 \rfloor, p(t), T)$

4   **else**

5 $\mathsf{ct} \leftarrow \lfloor c_0 \cdot \Delta \cdot q_{T_d} \rceil$

6   **for** $i = d; i > 0; i = i - 1$ **do**

7 $\mathsf{ct} \leftarrow \mathsf{Add}(\mathsf{ct}, \mathsf{MultConst}(T_i, \lfloor (c_i \cdot \Delta \cdot q_{T_d})/\Delta_{T_i} \rceil))$

8   **end**

9   **return** $\mathsf{Rescale}(\mathsf{ct})$

10   **end**

11 **end**

12 Express $p(t)$ as $q(t) \cdot T_{2^{m-1}} + r(t)$

13 $\mathsf{ct}_0 \leftarrow \mathsf{EvalRecurse}((\Delta \cdot q_{T_{2^{m-2}}})/\Delta_{T_{2^{m-1}}}, m - 1, l, q(t), T)$

14 $\mathsf{ct}_1 \leftarrow \mathsf{EvalRecurse}(\Delta, m - 1, l, r(t), T)$

15 $\mathsf{ct}_0 \leftarrow \mathsf{Mul}(\mathsf{ct}_0, T_{2^{m-1}})$

16 **if** $level(ct_0) > level(ct_1)$ **then**

17 $\mathsf{ct}_0 \leftarrow \mathsf{Add}(\mathsf{Rescale}(\mathsf{ct}_0), \mathsf{ct}_1)$

18 **else**

19 $\mathsf{ct}_0 \leftarrow \mathsf{Rescale}(\mathsf{Add}(\mathsf{ct}_0, \mathsf{ct}_1))$

20 **end**

21 **return** ct_0

---

each of the coefficients $c_{i,j}$ must be scaled, so that the evaluation of $u_{i,2^l}(t)$ is also done with exact additions and ends up with the desired scale.

Algorithm 2 is our proposed solution: it integrates our scale-propagation technique to the recursive decomposition of $p(t)$ into $q(t)$ and $r(t)$. We compare Algorithms 1 and 2 in Table 1 by evaluating a Chebyshev interpolant of the homomorphic modular reduction done during the bootstrapping circuit. This function plays a central role in the bootstrapping hence is an ideal candidate for evaluating the effect of the proposed approaches (see Section 5.4). To verify that our algorithm correctly avoids additions between ciphertexts of different scales,

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 $N = 2^{16}$, $n = 2^{15}$, $h = 196$ and $q_i \approx 2^{55}$. $\Delta_\epsilon = |\Delta_{\text{in}} - \Delta_{\text{out}}| \cdot \Delta_{\text{in}}^{-1}$.

		$\log(1/\epsilon)$ for (K, d, r)				
	Δ_ϵ	(12, 34, 2)	(15, 40, 2)	(17, 44, 2)	(21, 52, 2)	(257, 250, 3)
Algorithm 1 ([19])	$2^{-31.44}$	30.36	30.05	29.73	29.19	25.00
Algorithm 2 (ours)	0	37.37	37.16	37.15	37.04	29.46

we forced both algorithms to always rescale a ciphertext before an addition (in practice, it is better to check the levels of the ciphertexts before an addition, and dynamically assess if a level difference can be used to scale one ciphertext to the scale of the other). We observe that our algorithm yields two advantages: It enables (i) a scale-preserving polynomial evaluation (the output-scale is identical to the input scale), and (ii) a much better precision by successfully avoiding errors due to additions between ciphertexts of different scales.

3.3   Depth-Optimal Polynomial Evaluation

In practice, Algorithm 1 will consume more than the optimal m levels for a specific class of d due to the way the rescale and level management work in the full-RNS variant of the CKKS scheme. This discrepancy arises from the following interactions (recall that Algorithm 1 evaluates each $u_i(t)$ as a linear combination of a pre-computed power-basis $\{T_0(t), T_1(t), \ldots, T_{2^l-1}(t)\}$):

1. If $l > 1$, then the depth to evaluate $T_{2^l-1}(t)$ is l and evaluating the $u_i(t)$ will necessarily cost $l+1$ levels due to the constant multiplications.
2. If $l = 1$, then the depth to evaluate $T_1(t)$ is zero, hence the depth to evaluate the $u_i(t)$ is and remains l.
3. If $d > 8$, then Algorithm 1 sets $l > 1$.
4. If $2^m - 2^{l-1} \leq d < 2^m$, then all the elements of the power basis $\{T_{2^l}, T_{2^{l+1}}, \ldots, T_{2^{m-1}}\}$ need to be used during the recombination step of Algorithm 1.

Hence, if $l > 1$ and $d > 2^m - 2^{l-1}$, the total depth to execute Algorithm 1 is necessarily $m + 1$. This could be avoided by always setting $l = 1$ regardless of d, but it would lead to a very costly evaluation, as the number of non-scalar multiplications would grow proportionally to d. To mitigate this additional cost, we only enforce $l = 1$ on the coefficient of $p(t)$ whose degree is $\geq 2^m - 2^{l-1}$. Hence, Algorithm 2 first splits $p(t)$ into $p(t) = a(t) + b(t) \cdot T_{2^m-2^{l-1}}(t)$. It then evaluates $a(t)$ with the optimal l and recurses on $b(t)$ until $l = 1$. The number of additional recursions is bounded by $\log(m)$, because each recursion sets the new degree to half of the square root of the previous one. In practice, these additional recursions add only $\lceil \log(d+1-(2^m-2^{l-1})) \rceil$ non-scalar multiplications but enable the systematic evaluation of any polynomial by using exactly m levels.

3.4   Conclusions

For an extra cost of $\lceil \log(d + 1 - (2^m - 2^{l-1})) \rceil$ ciphertext-ciphertext products, our proposed algorithm guarantees an optimal depth hence an optimal-level consumption. This extra cost is negligible, compared to the base cost of Algorithm 1, i.e., $2^{m-l} + 2^l + m - l - 3 + \lceil (d + 1)/2^l \rceil$. It also guarantees exact additions throughout the entire polynomial evaluation, hence preventing the precision loss related to additions between ciphertexts of different scales and making the procedure easier to use. It also enables the possibility to choose the output scale that can be set to the same as the input scale, making the polynomial evaluation

scale-preserving. As linear transformations and constant multiplications can already be made to be scale-preserving, our polynomial evaluation is the remaining building block for enabling scale-preserving circuits of arbitrary depth.

4   Key-switch and Improved Matrix-Vector Product

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

We propose an optimized key-switch key format and key-switch algorithm (Section 4.1). We then apply them to rotation-keys and further improve the hoisted-rotation technique (Section 4.2) introduced by Halevi et al. [18]. Finally, we propose a modified procedure for matrix-vector multiplications over packed ciphertexts (Section 4.3) which features a novel *double-hoisting* optimization.

4.1   Improved Key-switch Keys

Given a ciphertext modulus $Q_L = \prod_{j=0}^{L} q_j$, we use a basis $\mathbf{w}$ composed of products among the q_j, as described by Han and Ki [19]. We also include the entire basis $\mathbf{w}$ in the keys, as done by Bajard et al. and Halevi et al. [3, 15]; this saves one constant multiplication during the key-switch and enables a simpler key-switch keys generation. A more detailed overview of these works can be found in Supplementary material B.

We propose a simpler and more efficient hybrid approach. Specifically, we use the basis $w^{(i)} = \frac{Q_L}{q_{\alpha_i}}[(\frac{Q_L}{q_{\alpha_i}})^{-1}]_{q_{\alpha_i}}$ with $q_{\alpha_i} = \prod_{j=\alpha i}^{\min(\alpha(i+1)-1,L)} q_j$ for $0 \le i < \beta$, $\beta = \lceil (L+1)/\alpha \rceil$ and α a positive integer. In other words, Q is factorized into β equally-sized composite-numbers q_{α_i}, each composed of up to α different primes. Thus, our key-switch keys have the following format:

$$\left(\mathsf{swk}_{q_{\alpha_i}}^0, \mathsf{swk}_{q_{\alpha_i}}^1 \right) = \left([-a_i s + s' \cdot P \cdot \frac{Q_L}{q_{\alpha_i}} \cdot [(\frac{Q_L}{q_{\alpha_i}})^{-1}]_{q_{\alpha_i}} + e_i]_{PQ_L}, [a_i]_{PQ_L} \right).$$

We set $P = \prod_{j=0}^{\alpha-1} p_j$, and the bit-size of P such that $q_{\alpha_i} \le P, \forall \alpha_i$. As shown by Gentry et al. [14], this leads to a negligible error introduced by the key-switch operation. Algorithm 3 describes the associated key-switch procedure that corresponds to the standard one adapted to our keys.

Algorithm 3: Key-switch

Input: $c \in R_{Q_\ell}$, the key-switch key $\mathsf{swk}_{s \to s'}$.
Output: $(a, b) \in R_{Q_\ell}^2$.

1 $\mathbf{d} \leftarrow \left[[c]_{q_{\alpha_{0 \le i < \beta}}}\right]_{PQ_\ell}$

2 $(a, b) \leftarrow (\langle \mathbf{d}, \mathsf{swk}^0 \rangle, \langle \mathbf{d}, \mathsf{swk}^1 \rangle)$

3 $(a, b) \leftarrow (\lfloor P^{-1} \cdot a \rceil, \lfloor P^{-1} \cdot b \rceil)$

4 return (a, b)

4.2   Improved Hoisted-Rotations

The slot-rotation operation in CKKS is defined by the automorphism $\phi_k : X \to X^{5^k} (\mathrm{mod}\ X^N + 1)$. It rotates the message slots by k positions to the left. After a rotation, the secret under which the ciphertext is encrypted is changed from s to $\phi_k(s)$, and a key-switch $\phi_k(s) \to s$ is applied to return to the original key.

Halevi et al. [18] show that as ϕ_k is an automorphism, it distributes over addition and multiplication, and commutes with the power-of-two base decomposition. As ϕ_k acts individually on the coefficients by permuting them without changing their norm (the modular reduction by $X^N + 1$ at most induces a sign change), it also commutes with the special RNS decomposition (see Supplementary material B): $[\phi_k(a)]_{q_{\alpha_i}} = \phi_k([a]_{q_{\alpha i}})$.

Hence, when several rotations have to be applied on the same ciphertext, $[a]_{q_{\alpha_i}}$ can be pre-computed and re-used for each subsequent rotation: $\sum \phi_k([a]_{q_{\alpha_i}}) \cdot \mathsf{rot}_{k,q_{\alpha_i}}$. This technique proposed by Halevi et al., called *hoisting*, significantly reduces the number of NTTs and CRT reconstructions, at the negligible cost of having to compute the automorphism for each of the $[a]_{q_{\alpha_i}}$.

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:

$$\left(\widetilde{\mathsf{rot}}_{k,q_{\alpha_i}}^0, \widetilde{\mathsf{rot}}_{k,q_{\alpha_i}}^1\right) = \left([-a_i \phi_k^{-1}(s) + s \cdot P \cdot \tfrac{Q_L}{q_{\alpha_i}} \cdot [(\tfrac{Q_L}{q_{\alpha_i}})^{-1}]_{q_{\alpha_i}} + e_i]_{PQ_L}, [a_i]_{PQ_L}\right)$$

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

$$\langle \phi_k(\mathbf{a}), \mathsf{rot}_k \rangle = \phi_k\left(\langle \mathbf{a}, \widetilde{\mathsf{rot}}_k \rangle\right).$$

Our improved algorithm for hoisted rotations is detailed in Algorithm 4.

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 a wide variety of applications of homomorphic encryption. In particular, the homomorphic evaluation of the CKKS encoding and decoding procedures, which are linear transformations, dominates the cost in the original bootstrapping procedure.

Algorithm 4: Optimized Hoisting-Rotations

Input: $\mathsf{ct} = (c_0, c_1) \in R_{Q_\ell}^2$ and a set of r rotation keys $\widetilde{\mathsf{rot}}_{r_k}$.

Output: $\mathbf{v}$ a list containing each r_k rotation of ct.

1 $\mathbf{d} \leftarrow \left[[c_1]_{q_{\alpha_{0 \leq i < \beta}}}\right]_{PQ_\ell}$ // (*Decompose*)

2 foreach r_k **do**

3 $\quad (a, b) \leftarrow (\langle \mathbf{d}, \widetilde{\mathsf{rot}}_{r_k}^0 \rangle, \langle \mathbf{d}, \widetilde{\mathsf{rot}}_{r_k}^1 \rangle)$ // (*MultSum*)

4 $\quad (a, b) \leftarrow (\lfloor P^{-1} \cdot a \rceil, \lfloor P^{-1} \cdot b \rceil)$ // (*ModDown*)

5 $\quad \mathbf{v}_{r_k} \leftarrow (\phi_{r_k}(c_0 + a), \phi_{r_k}(b))$ // (*Permute*)

6 end

7 return v

Halevi and Shoup propose to express an $n \times n$ matrix M in diagonal form and to use a baby-step giant-step (BSGS) algorithm (Algorithm 5) to evaluate the matrix product in $\mathcal{O}(\sqrt{n})$ rotations [16, 17]. At the time of this writing, all the existing bootstrapping procedures for the CKKS scheme are based on this approach and are not reported to use *hoisting*, unlike done for BGV [17, 18]. We now break down the cost of this BSGS algorithm, analyze its components and, using our observations, we present our improvements to this approach.

**Algorithm 5:** BSGS Algorithm of [18] For Matrix × Vector Multiplication

---

Input: 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 $\mathsf{ct}' = \mathbf{M} \times \mathsf{ct}$.

1 for $i = 0; i < n_1; i = i + 1$ **do**

2 $\quad \mathsf{ct}_i \leftarrow \mathsf{Rotate}_i(\mathsf{ct})$

3 end

4 $\mathsf{ct}' \leftarrow (0, 0)$

5 for $j = 0; j < n_2; j = j + 1$ **do**

6 $\quad \mathbf{r} \leftarrow (0, 0)$

7 $\quad$ **for** $i = 0; i < n_1; i = i + 1$ **do**

8 $\quad\quad \mathbf{r} \leftarrow \mathsf{Add}(\mathbf{r}, \mathsf{Mul}(\mathsf{ct}_i, \mathsf{Rotate}_{-n_1 \cdot j}(\mathbf{M}_{diag}^{(n_1 \cdot j + i)})))$

9 $\quad$ **end**

10 $\quad \mathsf{ct}' \leftarrow \mathsf{Add}(\mathsf{ct}', \mathsf{Rotate}_{n_1 \cdot j}(\mathbf{r}))$

11 end

12 $\mathsf{ct}' \leftarrow \mathsf{Rescale}(\mathsf{ct}')$

13 return ct'

Dominant Complexity of Rotations The dominant cost factor of Algorithm 5 is the number of rotations, as each rotation requires key-switch operations. These rotations comprise four steps (see Algorithm 4):

1. *Decompose*: Decompose a polynomial of R_{Q_ℓ} in base $\mathbf{w}$ and return the result in R_{PQ_ℓ}. This operation requires NTTs and CRT basis extensions.
2. *MultSum*: Compute a sum of products of polynomials in R_{PQ_ℓ}. This operation only requires coefficient-wise additions and multiplications.
3. *ModDown*: Divide a polynomial of R_{PQ_ℓ} by P and return the result in R_{Q_ℓ}. This operation requires NTTs and CRT basis extensions.
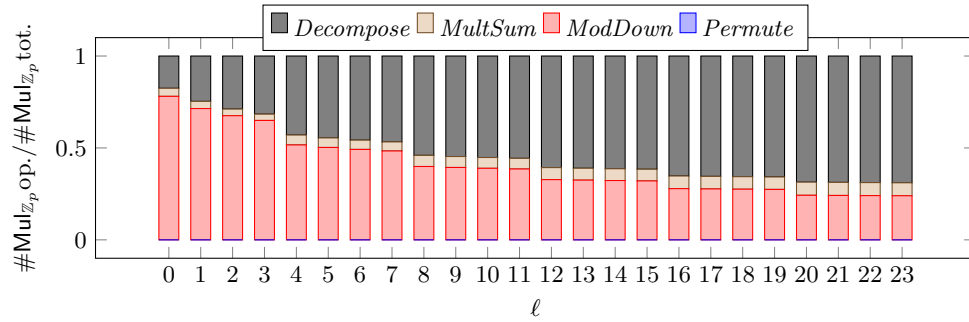
Fig. 1: Normalized complexity of each step (op.) of a rotation. The complexity for each operation was computed with $N = 2^{16}$, $0 \leq \ell \leq 23$ and $\alpha = 4$. The complexity derivation can be found in Supplementary material C.2.

4. *Permute*: Apply the automorphism ϕ_k on a polynomial of R_{Q_ℓ}. It represents a permutation of the coefficients and has in theory no impact on complexity.

Let n be the number of non-zero diagonals of M, and n_1, n_2 be two integers such that $n = n_1 n_2$; the complexity of the original BSGS algorithm (Algorithm 5) is $n_1 + n_2$ rotations and it is minimized when $n_1 \approx n_2$:

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

to which $2 n_2 n_1$ multiplications in R_{Q_ℓ} should also be added (line 8 of Algorithm 5). We denote *inner-loop* and *outer-loop* the lines that depend, respectively, on n_1 and n_2. Figure 1 shows the weight of each of the four steps in the total complexity. The complexity of the steps *MultSum* and *Permute* is negligible compared to the complexity of *Decompose* and *ModDown*, as products and additions are very inexpensive compared to NTTs and CRT basis extensions. We base our optimization on this observation.

Improved BSGS Algorithm We propose a new optimization that we refer to as *double-hoisting*. This optimization improves the *hoisting* technique proposed by Halevi et al. [18] and further reduces the complexity related to the *inner-loop* rotations by adding a second layer of *hoisting*.

The *first level*, proposed by Halevi et al. [18], applies to the *inner-loop* rotations (line 8 of Algorithm 5). This renders the computation devoted to *Decompose* independent of the value n_1, so the complexity is reduced to

$$n_2 \cdot (Decompose + MultSum + ModDown + Permute)$$
$$+ n_1 \cdot (MultSum + ModDown + Permute) + Decompose.$$

The *second level*, which we propose, introduces an additional hoisting for the *inner-loop* rotations, as the *ModDown* step is a coefficient-wise operation. Similarly to the *Decompose* step, this operation commutes with the *Permute* step and the ciphertext-plaintext multiplications (line 8 of Algorithm 5). Therefore,

---

Algorithm 6: Double-hoisting BSGS matrix×vector algorithm

Input: $\mathsf{ct} = (c_0, c_1) \in R_{Q_\ell}^2$, $\mathbf{M}_{diag} \in R_{PQ_\ell}$ the pre-rotated diagonals of $\mathbf{M}_{n \times n}$,
$\quad\quad n_1 n_2 = n$, $\mathsf{rot}_i \in R_{PQ_\ell}^2$ the set of necessary rotations keys.
Output: The evaluation of $\mathbf{M} \times \mathsf{ct}$.

1   $\mathbf{d} \leftarrow \left[[c_1]_{q_{\alpha_{0 \leq i < \beta}}}\right]_{PQ_\ell}$ // $Q_\ell \to PQ_\ell$
2   **for** $i = 0; i < n_1; i = i + 1$ **do**
3     $(a_i, b_i) \leftarrow (P \cdot c_0 + \langle \mathbf{d}, \widetilde{\mathsf{rot}}_i^0 \rangle, \langle \mathbf{d}, \widetilde{\mathsf{rot}}_i^1 \rangle)$ // $\in PQ_\ell$
4   **end**
5   $(r_0, r_1), r_2 \leftarrow (0, 0), (0)$
6   **for** $j = 0; j < n_2; j = j + 1$ **do**
7     $(u_0, u_1) \leftarrow (0, 0)$
8   **for** $i = 0; i < n_1; i = i + 1$ **do**
9       $(u_0, u_1) \leftarrow (u_0, u_1) + (a_i, b_i) \cdot \mathbf{M}_{diag}^{(n_1 \cdot j + i)}$ // $\in PQ_\ell$
10   **end**
11     $(u_0, u_1) \leftarrow (\lfloor P^{-1} \cdot u_0 \rceil, \lfloor P^{-1} \cdot u_1 \rceil)$ // $(PQ_\ell \to Q_\ell)$
12     $\mathbf{d} \leftarrow \left[[u_1]_{q_{\alpha_{0 \leq i < \beta}}}\right]_{PQ_\ell}$ // $Q_\ell \to PQ_\ell$
13     $(r_0, r_1) \leftarrow (r_0, r_1) + \left(\phi_{n_1 \cdot j}\left(\langle \mathbf{d}, \widetilde{\mathsf{rot}}_{n_1 \cdot j}^0 \rangle\right), \phi_{n_1 \cdot j}\left(\langle \mathbf{d}, \widetilde{\mathsf{rot}}_{n_1 \cdot j}^1 \rangle\right)\right)$ // $\in PQ_\ell$
14     $r_2 \leftarrow r_2 + \phi_{n_1 \cdot j}(u_0)$ // $\in Q_\ell$
15   **end**
16   $(r_0, r_1) \leftarrow (\lfloor P^{-1} \cdot r_0 \rceil, \lfloor P^{-1} \cdot r_1 \rceil)$ // $(PQ_\ell \to Q_\ell)$
17   **return** $(r_0 + r_2, r_1)$

---

we need to apply it only once after the entire *inner-loop* of n_1 rotations. Applying the same reasoning for the *ModDown* step of the *outer-loop* rotations we can reduce the number of key-switch operations from $n_1 + n_2$ to $n_2 + 1$:

$$n_2 \cdot (Decompose + MultSum + ModDown + Permute)$$
$$+ \, n_1 \cdot (MultSum + Permute) + Decompose + ModDown.$$

Algorithm 6 describes our double-hoisting BSGS for matrix-vector products.

Discussion In addition to benefiting from our improved key-switch (Section 4.1) and rotation (Section 4.2) procedures, Algorithm 6 introduces a trade-off: The *ModDown* step in the *inner-loop* now depends on the value n_2, and the *ModDown* step of the *outer-loop* is performed only once. However, the $2n_1 n_2$ multiplications and additions are performed in R_{PQ_ℓ} instead of R_{Q_ℓ}. Hence, the complexity dependency on n_1 is significantly reduced at the cost of slightly increasing the dependency on $n_1 n_2$. Applying the *ModDown* step at the end of each loop has the additional benefit of introducing the rounding error only once.

Table 2 compares the complexity of a non-hoisted, single-hoisted (Algorithm 5) and double-hoisted (Algorithm 6) BSGS, each with its optimal ratio n_1/n_2. 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 [9, 5, 19], which minimize the number of rotations by setting $n_1 \approx n_2$, is not optimal anymore. The maximum gain occurs when n (the number of non zero diagonals)

Table 2: Complexity of Algorithm 5 [16], *1-hoisted* Algorithm 5 [18] and our *2-hoisted* Algorithm 6. M is a $2^{15} \times 2^{15}$ matrix with $n = n_1 n_2$ non zero diagonals. The used parameters are $N = 2^{16}$, $n = 2^{15}$, $\ell = 18$, $\alpha = 4$. The speed-up factor is the ratio between the $\#\mathsf{Mul}_{\mathbb{Z}_p}$, taking as baseline the 1-hoisted approach.

	No hoisting [16]			1-hoisted [18]		2-hoisted (proposed)		
n	n_1/n_2	$\log(\#\mathsf{Mul}_{\mathbb{Z}_p})$	Speed-up	n_1/n_2	$\log(\#\mathsf{Mul}_{\mathbb{Z}_p})$	n_1/n_2	$\log(\#\mathsf{Mul}_{\mathbb{Z}_p})$	Speed-up
32768	2	37.276	$0.777\times$	2	36.913	8	36.813	$1.071\times$
16384	1	36.500	$0.765\times$	4	36.114	16	35.903	$1.157\times$
8192	2	35.865	$0.706\times$	2	35.364	8	35.055	$1.238\times$
4096	1	35.152	$0.705\times$	4	34.648	16	34.205	$1.359\times$
2048	2	34.597	$0.652\times$	2	33.981	8	33.446	$1.448\times$
1024	1	33.927	$0.664\times$	4	33.337	16	32.672	$1.585\times$
512	2	33.422	$0.619\times$	2	32.732	8	32.014	$1.644\times$
256	1	32.769	$0.645\times$	4	32.137	16	31.318	$1.764\times$
128	2	32.282	$0.609\times$	2	31.568	8	30.753	$1.759\times$
64	1	31.614	$0.649\times$	4	30.992	16	30.127	$1.821\times$
32	2	31.112	$0.623\times$	2	30.430	8	29.637	$1.732\times$
16	1	30.375	$0.682\times$	4	29.842	16	29.311	$1.445\times$
8	2	29.792	$0.685\times$	2	29.248	2	29.116	$1.094\times$

is around 128. This can be exploited by factorizing the linear transforms, used during the bootstrapping, into several sparse matrices (see Section 5.3).

Increasing the ratio from $n_2/n_1 \approx 1$ to $n_2/n_1 \approx 16$ in our bootstrapping parameters (Section 6) increases the number of keys by a factor around 1.6 and reduces the computation time by 20%. Hence, Algorithm 6 reduces the overall complexity of matrix-vector products, by introducing a time-memory trade-off.

We also observe that these improvements are not restricted to plaintext matrices or to the CKKS scheme and can be applied to other R-LWE scheme, such as BGV [4] or BFV [12], as long as the scheme (or its implementation) allows for the factorization of an expensive operation. For example, in the BFV scheme, the quantization (division by Q/t) (as well as the re-linearization if the matrix is in ciphertext) can be delayed to the *outer-loop*.

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. [9] and adapt each step by relying on the techniques proposed in Sections 3 and 4.

The purpose of the CKKS bootstrapping [9] is, in contrast with BFV's [12], not to reduce the error. Instead, and similarly to BGV [4] bootstrapping, it is meant to reset the ciphertext modulus to a higher level in order to enable further homomorphic multiplications. The approximate nature of CKKS, due to the plaintext and ciphertext error being mixed together, implies that each homomorphic operation decreases the output precision. As a result, all the currently proposed bootstrapping circuits only approximate the ideal bootstrapping operation, and their output precision also 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 and the error induced by the bootstrapping circuit.

The bootstrapping circuit is divided into the five steps detailed below. We provide a schematic view of this circuit (for our implementation) in Supplementary material F. 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 $\mathcal{O}(\sqrt{h})$ [9]. The next four steps remove 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 [9].

3. CoeffsToSlots: The message $m' = Q_0 \cdot \tilde{I}(Y) + \lfloor \Delta \cdot m(Y) \rceil$ is in the *coefficient* domain, 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 $\dfrac{Q_0}{2\pi\Delta} \cdot \sin\left(\dfrac{2\pi\Delta x}{Q_0}\right)$, which is tight when $Q_0/\Delta \gg ||m(Y)||$. As 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 encrypting $f(\mathsf{Encode}(m'))$. It returns a ciphertext at level $L - k$ 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.

5.2   ModRaise and SubSum

We base the ModRaise and SubSum operations directly on the initial bootstrapping of Cheon et al. [9]. 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, a linear transformation, to scale the corresponding matrices by $2n/N$. As we also use this trick for grouping other constants, we elaborate more on the matrices scaling in Section 5.5.

5.3   CoeffsToSlots and SlotsToCoeffs

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 complex 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. I.e., the encoding and decoding algorithms define an isomorphism between $\mathbb{R}[Y]/(Y^{2n}+1)$ and $\mathbb{C}^n$ [10]. The goal of the CoeffsToSlots and 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. As 5 and $-1 \bmod 2n$ 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 $\mathsf{Decode}_n(m(Y)) = (m(\psi), m(\psi^5), \ldots, m(\psi^{5^{2n-1}}))$. The decoding isomorphism is fully defined by the $n \times n$ special Fourier transform matrix $\mathrm{SF}_{n,(j,k)} = \psi^{j5^k}$, with inverse (the encoding matrix) $\mathrm{SF}_n^{-1} = \frac{1}{n}\overline{\mathrm{SF}}_n^T$ [6]. Its homomorphic evaluation can be expressed in terms of plaintext matrix-vector products:

1. $\mathsf{CoeffsToSlots}(\mathbf{m}) : t_0 = \frac{1}{2}\left(\mathrm{SF}_n^{-1} \times \mathbf{m} + \overline{\mathrm{SF}_n^{-1} \times \mathbf{m}}\right), t_1 = -\frac{1}{2}i(\mathrm{SF}_n^{-1} \times \mathbf{m} - \overline{\mathrm{SF}_n^{-1} \times \mathbf{m}})$
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. [9] homomorphically compute the DFT as a single matrix-vector product in $\mathcal{O}(\sqrt{n})$ rotations and depth 1, by using the baby-step giant-step (BSGS) approach of Halevi and Shoup [17] (Algorithm 5 in Section 4.3). To further reduce the complexity, two recent works from Cheon et al. [6] and Chen et al. [5] exploit the structure of the equivalent FFT algorithm by recursively merging its iterations, reducing the complexity to $\mathcal{O}(\sqrt{r}\log_r(n))$ rotations at the cost of increasing the depth to $\mathcal{O}(\log_r(n))$, for r a power-of-two radix between 2 and n.

We base our approach on the work of [6] and [5], and we use our *double hoisting* BSGS to evaluate the matrix-vector products (see Section 4.3 and Algorithm 6). This step is parameterized by $\rho = \lceil\log_r(n)\rceil$, the depth of the linear transformation (i.e., the number of matrices that we need to evaluate).
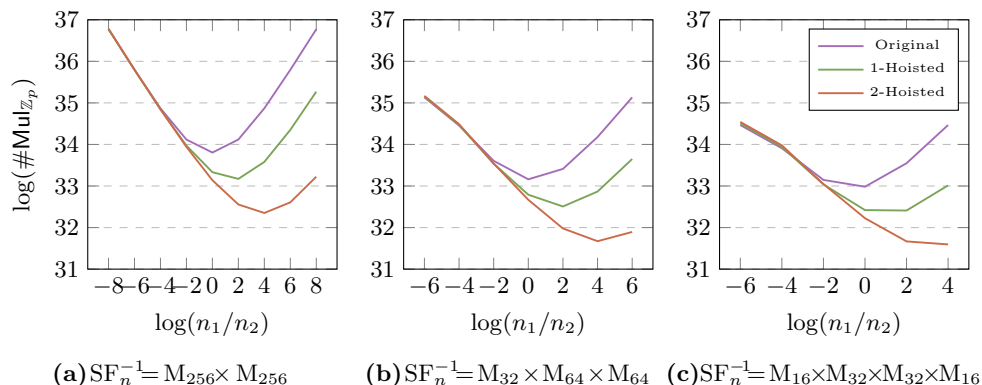
Fig. 2: Theoretical complexity of CoeffToSlots for different $\rho_{\mathrm{SF}_n^{-1}}$ using Algorithm 5 with no hoisting, single hoisting, and double hoisting (Algorithm 6).

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

Each level of hoisting reduces the total complexity by a noticeable amount. Regular *hoisting*, as proposed by Halevi and Shoup [18], achieves its minimum complexity when $n_1 \approx 2^2 n_2$ instead of $n_1 \approx n_2$. Using our *double hoisting*, the minimum complexity is further shifted 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 $2\times$ compared to the single *hoisting* technique of Halevi and Shoup.

Efficient Repacking of Sparse Plaintexts. The first part of CoeffsToSlots is a DFT that outputs a vector of $\mathbb{C}^n$ values; the second part of CoeffsToSlots applies the map $\mathbb{C}^n \to \mathbb{R}^{2n}$ to this vector. During the decoding, the inverse mapping $\mathbb{R}^{2n} \to \mathbb{C}^n$ is used. This map can be computed with simple operations, e.g., conjugation, multiplication by $-i$, and additions. If the original ciphertext is not fully packed ($0 < n < N/2$ slots), the two resulting ciphertexts can be merged into one, requiring one evaluation of EvalSine instead of two.

We observe that decoding a plaintext $\mathbf{m} \in \mathbb{C}^n$ by 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 $\mathbf{m}$. 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 repacking and unpacking ciphertexts before and after the EvalSine:

- **Repacking before the EvalSine** ($\mathbb{C}^n \to \mathbb{R}^{2n}$): Repacking into one 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 last n slots are set to zero and can be used to store the imaginary part of the first n slots.

This repacking involves one additional rotation and it does not consume any additional levels.

- **Unpacking after the EvalSine** ($\mathbb{R}^{2n} \to \mathbb{C}^n$): For this operation, we evaluate the following $2n \times 2n$ matrix on the ciphertext before the DFT

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

  where I_n is the $n \times n$ identity matrix. Its effect is to homomorphically apply the map $\mathbb{R}^{2n} \to \mathbb{C}^n || \mathbb{C}^n$, 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, thus slightly increasing its density.

5.4   EvalSine

EvalSine implements the homomorphic modular reduction of the message $m' = Q_0 \cdot \tilde{I}(Y) + \Delta \cdot m(Y)$ modulo Q_0. The modular reduction is approximated by

$$f(x) = \frac{Q_0}{\Delta} \frac{1}{2\pi} \sin\left(2\pi x \frac{\Delta}{Q_0}\right) \approx \frac{Q_0}{\Delta} \cdot \left(\frac{\Delta}{Q_0} x \bmod 1\right),$$

which scales the message m' down to $\tilde{I}(Y) + (\Delta/Q_0) \cdot m(Y)$, removes the $\tilde{I}(Y)$ polynomial by reducing the message modulo 1, and scales the message back to $\Delta \cdot m(Y)$. As $\tilde{I}(Y)$ determines the range and degree of the approximation, the EvalSine step has to account for the secret-key density h. In particular, the range of the approximation $(-K, K)$ is chosen such that $\Pr[||\tilde{I}(Y)|| > K] \leq \kappa$ for a user-defined κ. We elaborate more on how we parameterize K, in Section 6.2.

Previous Work Chen et al. [5] directly approximate the function $\frac{1}{2\pi} \cdot \sin(2\pi x)$ by using a standard Chebyshev interpolant of degree $d = 119$ in an interval of $(-K, K)$ for $K = 12$ (using a sparse key with $h = 64$). Han and Ki [19] 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 obtain $\sin(2\pi x)$. 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. This reduces the degree of the approximation and the cost of its evaluation. In their work, they use an interpolant of degree 30 with a scaling factor $r = 2$ (they also use a sparse key with $h = 64$).

In a recent work, Lee et al. [24] propose to compose the sine/cosine function with a low degree arcsine. This additional step corrects the error introduced by the sine, especially if Q_0/Δ is small (when the values are not close to the origin). This improves the overall precision of the bootstrapping and enables bootstrapping messages with larger values. However, this comes at the cost of increasing the depth of the EvalSine step, as a second polynomial must be evaluated.

Our Work Both the methods of Chen et al. and Han and Ki have $d = \mathcal{O}(K)$, therefore doubling K requires at most doubling d, and the evaluation will require at most one additional level, as 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 (as required by dense keys). However, the effects of the approximate rescale procedure, if not properly managed, can significantly reduce the output precision. Our EvalSine makes use of our novel polynomial evaluation technique (Section 3).

We propose a more compact expression of the modular reduction function $f(x) = \frac{1}{2\pi} \sin(2\pi x)$, which is approximated by $g_r(x)$, a modified scaled cosine followed by r iterations of the double-angle formula:

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

We include the $1/2\pi$ factor directly in the function we approximate, even when using the double angle formula, without consuming an additional level, impacting the precision, or fundamentally changing its evaluation. We observed that even though the approximation technique of Han and Ki is well suited for small K, the standard Chebyshev interpolation technique, as used by Chen et al., remains more efficient when K is large. The reason is that Han and Ki's interpolant has a minimum degree of $2K - 1$, so it grows in degree with respect to K much faster than the standard Chebyshev interpolation. Hence, we use the approximation method of Han and Ki when K is small (for sparse keys) and the standard Chebyshev approximation, as done by Chen et al., for dense keys.

As suggested by Lee et al. [24], we can further improve this step by composing it with $\arcsin(x)$, i.e., $\frac{1}{2\pi} \arcsin(\sin(2\pi x))$, which corrects the error $e_{g_r(x)} = |g_r(x) - x \bmod 1|$. Unlike Lee et al., we do not interpolate the arcsine, rather we choose to use a low degree Taylor polynomial and show in our results (see Section 7.2) that it is sufficient to achieve similar results.

Algorithm 7 details our implementation of the EvalSine procedure. The ciphertext must be multiplied by several constants, before and after the polynomial evaluation. For efficiency, we merge these constants with the linear transformations. See Section 5.5 for further details.

5.5   Matrix Scaling

Several steps of the bootstrapping circuit require the ciphertexts to be multiplied by constant plaintext values. This is most efficiently done by merging them and pre-multiplying the resulting constants to the SF_n^{-1} and SF_n matrices.

Before EvalSine, the ciphertext has to be multiplied (i) by $1/N$ to cancel the $N/2n$ and $2n$ factors introduced by the SubSum and CoeffsToSlots steps, (ii) by $1/(2^r K)$ for the scaling by $1/2^r$ and change of variable for the polynomial evaluation in Chebyshev basis, and (iii) by $Q_0/2^{\lceil \log(Q_0) \rceil}$ to compensate for the error introduced by the approximate division by $\lfloor Q_0/\Delta \rfloor$. Therefore, the matrices

---

Algorithm 7: EvalSine

Input: $\{\mathsf{ct}, Q_\ell, \Delta\}$ a ciphertext, $p(t)$ a Chebyshev interpolant of degree d of
$f(x) = x \bmod 1$, K the range of interpolation, r a scaling factor.
Output: The evaluation $\mathsf{ct}' = \lfloor Q_0/\Delta \rceil \cdot p(\lfloor Q_0/\Delta \rceil^{-1} \cdot \mathsf{ct})$.

1 $\Delta \leftarrow \Delta \cdot \lfloor Q_0/\Delta \rceil$ // Division by $\lfloor Q_0/\Delta \rceil$
2 $T_0 \leftarrow 1$
3 $T_1 \leftarrow \mathsf{AddConst}(\mathsf{ct}, -0.5/(2^{r+1}K))$
4 $\mathsf{m} \leftarrow \lceil \log(d+1) \rceil$
5 $\mathsf{l} \leftarrow \lfloor \mathsf{m}/2 \rfloor$
6 $T \leftarrow \{T_0, T_1, \ldots, T_{2^l}; T_{2^l+1}, \ldots, T_{2^{m-1}}\}$ // Compute the power basis
7 **for** $i = 0; i < r; i = i+1$ **do**
8 $\quad \Delta \leftarrow \sqrt{\Delta \cdot q_{L-\mathsf{CtS\ depth}-\mathsf{EvalSine\ depth}-r+i}}$ // Pre-compute target Δ
9 **end**
10 $\mathsf{ct}' \leftarrow \mathsf{EvalRecurse}(\Delta, \mathsf{m}, \mathsf{l}, p(t), T)$ (Algorithm 2) // Outputs ct' with target
$\quad \Delta$ scale
11 **for** $i = 0; i < r; i = i+1$ **do**
12 $\quad \mathsf{ct}' \leftarrow \mathsf{AddConst}(2 \cdot \mathsf{Mul}(\mathsf{ct}', \mathsf{ct}'), -(1/2\pi)^{1/2^{r-i}})$
13 $\quad \mathsf{ct}' \leftarrow \mathsf{Rescale}(\mathsf{ct}')$ // $\Delta \leftarrow \Delta^2/q_{L-\mathsf{CtS\ depth}-\mathsf{EvalSine\ depth}-i}$
14 **end**
15 $\Delta \leftarrow \Delta \cdot \lfloor Q_0/\Delta \rceil^{-1}$ // Multiplication by $\lfloor Q_0/\Delta \rceil$
16 **return** ct'

---

resulting from the factorization of SF_n^{-1} are scaled by

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

where $\rho_{\mathrm{SF}_n^{-1}}$ is the degree of factorization of SF_n^{-1}. Evenly spreading the scaling factors across all matrices ensures that they are scaled by a value as close as possible to 1.

After EvalSine, the ciphertext has to be multiplied (i) by $2^{\lceil \log(q_0) \rceil}/Q_0$ to compensate for the error introduced by the approximate multiplication by $\lfloor Q_0/\Delta \rceil$, and (ii) by Δ/δ, where Δ is the scale of the ciphertext after the EvalSine step and δ is the desired ciphertext output scale. Therefore, the matrices resulting from the factorization of SF_n are scaled by

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

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

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 accomplished mostly

Table 3: Modulus size $\log(QP)$ for different secret-key densities h ($\lambda \geq 128$).

h	$\log(QP)$		
	$\log(QP,N), \lambda \geq 128$	$N = 2^{15}$	$N = 2^{16}$
64	$0.015121N - 8.248756$	496	982
96	$0.018896N - 3.671642$	619	1234
128	$0.021370N - 3.601990$	699	1396
192	$0.023448N - 3.611940$	767	1533
$N/2$	[11]	881	1782

through experimental processes for finding appropriate trade-offs between the performance and the probability of decryption errors. In this Section, we discuss various constraints and inter-dependencies in the parameter selection. Then, we propose a generic procedure for finding appropriate parameter sets.

6.1 Security

For each parameter set, we select a modulus size with an estimated security of 128 bits. These values are shown in Table 3 for several choices of the secret-key Hamming weight h, and are based on the work of Curtis and Player [11]. According to the authors, these parameters result from conservative estimations, and account for hypothetical future improvements to the most recent attacks of Cheon et al. [8] and Son et al. [28]. Therefore, their actual security is underestimated.

6.2 Choosing K for EvalSine

Each coefficient of the polynomial $\tilde{I}(Y) \in \mathbb{R}[Y]/(Y^{2n} + 1)$ is the result of the sum of $h + 1$ uniformly distributed variables in $\mathbb{Z}_{Q_0}$ [9], hence it follows an Irwin–Hall distribution [24]. By centering and normalizing the coefficients of $\tilde{I}(Y)$, we get instead the sum of $h + 1$ uniformly distributed variables in $(-0.5, 0.5)$. The probability $\Pr[\|\tilde{I}(Y)\| > K]$ can be computed by adapting the cumulative probability function of the Irwin-Hall distribution:

$$1 - \left(\left(\frac{2}{(h+1)!} \sum_{i=0}^{\lfloor K+0.5(h+1) \rfloor} (-1)^i \binom{h+1}{i} (K + 0.5(h+1) - i)^{h+1} \right) - 1 \right)^{2n}.$$

(1)

The previous works [9, 5, 6, 19] use a sparse key with $h = 64$ and $K = 12$, which regardless of the security, gives a failure probability of $2^{-14.7}$ and $2^{-6.7}$ for $n = 2^7$ and $n = 2^{15}$ respectively, according to Equation (1). Clearly, these parameters were not chosen for large n and are most likely an artifact of the first proposal for a bootstrapping for CKKS [9], for which only a small number of slots was practical. In our work, we increase h to ensure an appropriate security and use a much larger number of slots (e.g., $n = 2^{15}$), hence we need to adapt K. Table 4 shows that if we target a failure probability $\leq 2^{-15.0}$ for $n = 2^{15}$ slots and take h as a parameter, then $K \approx 1.81\sqrt{h}$.

Table 4: $\Pr[||\tilde{I}(Y)|| > K] \approx 2^{-16}$ for $n = 2^{15}$ and variable h.

$\log_2(h)$	6	7	8	9	10	11	12	13	14	15
K	14	20	29	41	58	82	116	163	232	328
$\log_2(\Pr[||\tilde{I}(Y)|| > K])$	-14.6	-14.6	-15.7	-15.6	-15.5	-15.4	-15.4	-15.4	-15.4	-15.4
$K/\sqrt{h}$	1.75	1.76	1.81	1.81	1.81	1.81	1.81	1.81	1.81	1.81

Table 5: The sets of parameters of the full-RNS CKKS used to evaluate the performance of our bootstrapping code. + means concatenation in the chain and $a \cdot b$ denotes the consecutive concatenation of a different moduli of size b. Moduli with fractional a are only partially used by the step they are allocated to.

Parameters										
Set	h	N	Δ	$\log(QP)$	L	$q_{0 \le i \le (L-k)}$	StC	Sine	CtS	$\log(p_j)$
I	192		2^{40}	1546	24	$60 + 9 \cdot 40$	$3 \cdot 39$	$8 \cdot 60$	$4 \cdot 56$	$5 \cdot 61$
II	192	2^{16}	2^{45}	1547	23	$60 + 5 \cdot 45$	$3 \cdot 42$	$11 \cdot 60$	$4 \cdot 58$	$4 \cdot 61$
III	192		2^{30}	1553	21	$55 + 7.5 \cdot 60$	$1.5 \cdot 60$	$8 \cdot 55$	$4 \cdot 53$	$5 \cdot 61$
IV	32768		2^{45}	1792	27	$50 + 9 \cdot 40$	$56 + 28$	$12 \cdot 60$	$4 \cdot 53$	$6 \cdot 61$
V	192	2^{15}	2^{25}	768	13	$33 + 50 + 25$	60	$8 \cdot 50$	$2 \cdot 49$	$2 \cdot 50$

6.3 Finding Parameters

We describe a general heuristic procedure for selecting and fine-tuning 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 8 and propose five reference parameter sets that result from this algorithm. The parameter sets were selected for their performance and similarity with those in previous works, thus enabling a comparison. For each set, Table 5 shows the parameters related to CKKS and to the bootstrapping circuit.

7 Evaluation

We implemented the improved algorithm of Sections 3 and 4, along with the bootstrapping procedure of Section 5 in the Lattigo library [23]. We evaluated it by using the parameters of Section 6.3. Lattigo is an open-source library that implements the RNS variants of the BFV [12, 3, 15] and CKKS [7] schemes in Golang [29]. All experiments were conducted single-threaded on an i5-6600k at 3.5 GHz with 32 GB of RAM running Windows 10 (Go version 1.15.6, GOARCH=amd64, GOOS=windows).


Algorithm 8: Heuristic Parameter Selection

Input: λ a security parameter.
Output: The parameters $(N, n, h, Q_L, P, \kappa, \alpha, d_{\sin}, r, d_{\arcsin}, \rho_{\mathrm{SF}_n^{-1}}, \rho_{\mathrm{SF}_n})$.

1 Select n, N and h and derive $\log(PQ_L)$ according to λ.

2 Given Δ (the scale of the message), compute the ratio Q_0/Δ and select the bootstrapping output precision δ.

3 Given a target failure probability κ, estimate K using Equation (1).

4 Given the bootstrapping output precision δ, find $d_{\sin}$ (the degree of the sine polynomial), r (the number of double angle) and $d_{\arcsin}$ (the degree of the arcsine polynomial) such that the polynomial approximation of x mod 1 of the EvalSine step in the interval $(-K/2^r, K/2^r)$ gives a precision greater than $\log(Q_0/\Delta) + \delta$ bits.

5 Select $\rho_{\mathrm{SF}_n^{-1}}$ and ρ_{SF_n} (the depth of the CoeffsToSlots and SlotsToCoeffs steps).

6 Allocate the q_j of the CoeffsToSlots, EvalSine and SlotsToCoeffs steps, with the maximum possible bit-size for all q_j.

7 Select α and allocate $P = \prod_{j=0}^{\alpha-1} p_j$, ensuring that $P \approx \beta \|q_{\alpha_i}\|$.

8 Run the bootstrapping and find the minimum values for d_{sin}, r and d_{arcsin} such that the output has δ bits of precision.

9 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.

10 Run the bootstrapping and find the minimum bit-size for the q_j of the CoeffsToSlots such that the output precision is not affected.

11 Run the bootstrapping and find the minimum bit-size for the q_j of the SlotsToCoeffs such that the output precision is not affected.

12 Allocate the rest of the moduli of Q_L such that $\log(PQ_L)$ ensures a security of at least λ and check again step 7.

13 If additional residual homomorphic capacity is needed or the security λ cannot be achieved
    1. Reduce α, $\rho_{\mathrm{SF}_n^{-1}}$ and/or ρ_{SF_n} and check again line 6.
    2. Increase h to increase $\log(PQ_L)$ and restart at line 1.
    3. Increase N to increase $\log(PQ_L)$ and restart at line 1.

return $(N, n, h, Q_L, P, \kappa, \alpha, d_{\sin}, r, d_{\arcsin}, \rho_{\mathrm{SF}_n^{-1}}, \rho_{\mathrm{SF}_n})$

7.1 The Bootstrapping Metrics

Although CPU costs are an 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, the failure probability, the precision, and the remaining multiplicative depth. To compare our bootstrapping procedure with the existing ones, we use the same concept of a *bootstrapping utility metric*, as introduced by Han and Ki [5].

Definition 1 (Bootstrapping Throughput). *For n a number of plaintext slots, $\log(\epsilon^{-1})$ the output precision, $\log(Q_{L-k})$ the output coefficient-modulus size after the bootstrapping (remaining homomorphic capacity) and complexity a measure of the computational cost (in CPU time), the bootstrapping throughput*
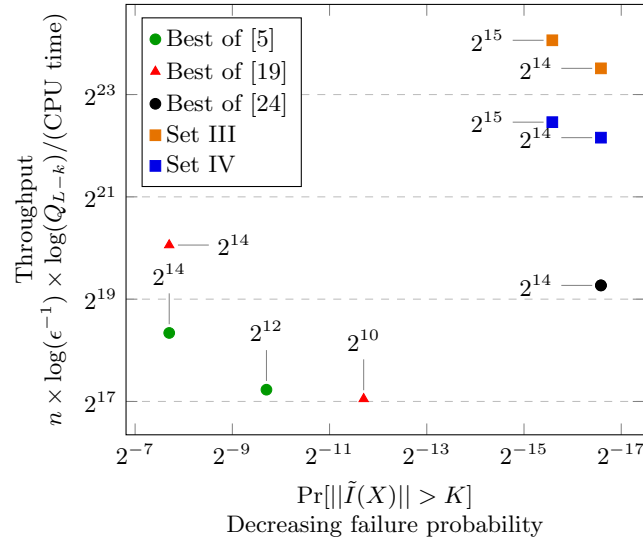
Fig. 3: Bootstrapping throughput comparison. We plot the results for our best performing parameter set against the state of the art. Nodes are labeled with n, the number of plaintext slots.

is defined as:

$$throughput = \frac{n \times \log(\epsilon^{-1}) \times \log(Q_{L-k})}{complexity}.$$

Note that we 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.

As κ, the bootstrapping failure probability, is a probability and not a metric, we chose to not include it directly in Definition 1. However, we still believe it should be taken into account as an opportunity-cost variable. Indeed, the event of a bootstrapping failure will likely result in the need to re-run the entire circuit. Hence, the probability of failure should be weighed vs. the cost of having to re-run a circuit to determine if κ is in an acceptable range.

7.2   Results

We run our benchmarks and report the bootstrapping performance for each parameter set of Table 5, and we compare them with the previous works of Chen et al. [5], Han and Ki [19], and the recent and concurrent work of Lee et al. [24]. Unfortunately, the implementations of these works have not been publicly released and we were not able to reproduce their results on our own hardware for a fair comparison. The parameters and results are summarized in Table 6 and 7,

Table 6: Parameter comparison of [5, 19, 24] and our work. "-" means that value was not reported. Lee et al.'s [24] parameters are based on our Set III.

Bootstrapping Parameters											
Set	N	$\log(QP)$	h	λ	$\rho_{\mathrm{SF}_n^{-1}}$	ρ_{SF_n}	Q_0/Δ	K	$d_{\sin(x)}$	r	$d_{\arcsin(x)}$
[24]		1553	192	≈ 128	2	2	$\dfrac{256}{8}$	25	$\dfrac{66}{68}$	2	$\dfrac{0}{5}$
[5]		2480	64	< 80	4	4	1024	12	119	0	0
[19]	2^{16}	1452	64	< 100	-	-	1024	12	31	2	0
I		1546	192	≈ 128	4	3	256	25	63	2	0
II		1547	192	≈ 128	4	3	256	25	63	2	0
III		1553	192	≈ 128	4	3	4	25	63	2	7
IV		1792	32768	≈ 128	4	3	256	325	255	4	0
[5]		1240	64	< 80	2	2	1024	12	119	0	0
[19]	2^{15}	910	64	< 90	-	-	1024	12	31	2	0
V		768	192	≈ 128	2	2	256	25	63	2	0

respectively. Reports on experiments that demonstrate the numerical stability of our bootstrapping can be found in Supplementary material E.

Focusing only on the overall performance, our most performing set (Set III) achieves throughput $14.1\times$ and $28.4\times$ larger than the best result reported by Han and Ki [19] and Lee et al. [24] respectively. Our Set IV uses dense keys and achieves a throughput $4.6\times$ and $9\times$ larger than the work of Han and Ki and Lee et al. respectively. Both these works use SEAL [27] and are evaluated on similar hardware. Our sets III and IV achieve a throughput $54.2\times$ and $17.4\times$ larger than the best result reported by Chen et al. [5], implemented using the HEAAN library [20]. HEAAN does not implement the full-RNS variant of CKKS, hence the latter comparison shows the significant performance gains that can be achieved by combining optimized algorithms with a full-RNS implementation.

The implementation of Lee et al. makes use of the recent work of Kim et al. [21] which proposes new techniques to minimize the error during computation, notably a *delayed rescaling* that consists in rescaling the ciphertext before a multiplication and not after, so that the error is as small as possible when doing the multiplication. This enables Lee et al. to achieve a slightly higher precision than ours (our implementation does not use the work of Kim et al.). Lee et al. results are also the ones with the most residual homomorphic capacity. The primary reason is the implementation of the CKKS scheme in SEAL, which can only use one special prime ($\alpha = 1$, see Section 4) during the key-switching. This increases the ciphertext homomorphic capacity, but at the cost of an increased key-switch complexity. The second reason is that they allocate less levels to the linear transformations (in total, three less than our parameters). This enables them to reduce the depth of the bootstrapping, at the cost of increasing its complexity, which shows in their timings.

We observe that there is a correlation between the value Q_0/Δ and the precision. A better precision is achieved when using a smaller ratio, even when the arcsin is not composed with the scaled sine. Previous works usually assume

Table 7: Comparison of the bootstrapping performances of [5, 19, 24] and our proposed bootstrapping for the full-RNS variant of CKKS with parameter sets I, II, III, IV and V. MU, SS, CtS, StC designate ModUp, SubSum, SlotstoCoeffs, CoeffstoSlots. "-" indicates that the prior work did not report the value. All timings are single threaded. The plaintext real and imaginary part are uniformly distributed in the interval -1 and 1.

Set	n	Timing(s)						$\log(Q_{L-k})$	$\log(\epsilon^{-1})$	$\log(\text{bits}/s)$	$\log(\kappa)$
		MU	SS	CtS	StC	Sine	Total				
[24]	2^{14}	-	-	-	-	-	461.5	**653**	27.2	19.26	-16.58
[24]	2^{14}	-	-	-	-	-	451.5	533	**32.6**	19.26	-16.58
[5]	2^{14}		119.8			38.5	158.3	172	18.6	18.33	-7.70
[5]	2^{12}		127.5			40.4	167.9	301	20.9	17.22	-9.70
[19]	2^{14}	-	-	-	-	-	52.8	370	10.8	20.24	-7.70
[19]	2^{10}	-	-	-	-	-	37.6	370	15.3	17.23	-11.70
I	2^{15}	0.06	0	6.5	3.7	12.8	23.0	420	25.7	23.87	-15.58
I	2^{14}	0.06	0.3	6.3	3.8	6.3	16.9	420	26.0	23.33	-16.58
II	2^{15}	0.06	0	6.8	2.2	14.2	23.4	240	31.5	23.33	-15.58
II	2^{14}	0.06	0.3	6.0	2.4	7.1	16.0	240	31.6	22.88	-16.58
III	2^{15}	0.06	0	5.4	2.4	10.1	**18.1**	505	19.1	**24.06**	-15.58
III	2^{14}	0.06	0.3	5.0	2.6	5.0	13.1	505	18.9	23.50	-16.58
IV	2^{15}	0.07	0	7.9	28.2	3.0	39.2	410	16.8	22.45	-14.90
IV	2^{14}	0.07	0.4	7.1	14.1	3.2	24.9	410	17.3	22.15	-15.90
[5]	2^{10}		28.8			9.5	38.3	150	6.9	14.75	-11.70
[5]	2^{8}		16.9			9.2	26.0	75	10.03	12.85	-13.70
[19]	2^{2}	-	-	-	-	-	7.5	185	15.0	10.53	-19.70
[19]	2^{1}	-	-	-	-	-	7.0	185	16.8	9.79	-20.70
V	2^{14}	0.02	0	3.7	0.7	2.9	7.5	110	15.5	21.82	-16.58
V	2^{13}	0.02	0.4	1.6	0.4	1.5	3.9	110	15.4	21.76	-17.58

that $||m|| \approx ||\text{FFT}^{-1}(m)||$ to set Q_0/Δ and derive the expected precision of the scaled sine. In practice, since each coefficient of $\text{FFT}^{-1}(m)$ is a dot product between the vector m and a complex vector of roots of unity (zero-mean and small variance), if the mean of m is close to zero, then $||\text{FFT}^{-1}(m)|| \ll ||m||$ with overwhelming probability. For example, given m uniform in $(-1, 1)$ and $n = 2^{15}$ slots, then $||m||/||\text{FFT}^{-1}(m)|| \approx 100$. Hence the message is much closer to the origin than expected, which reduces the inherent error of the scaled sine and amplifies the effectiveness of the arcsine. We note that even if the distribution of m is not known, it is possible to enforce this behavior with a single plaintext multiplication by homomorphically negating half of its coefficients before the bootstrapping. One could even homomorphically split m in half and create two symmetric vectors to enforce a zero mean. A more detailed analysis of this behavior and how to efficiently exploit it or integrate it into the linear transforms of the bootstrapping could be a interesting future research line.

All our sets have a failure probability that is two to three orders magnitude smaller than previous works, except for the results of Lee et al. which use our

suggested parameters. For example, following Equation (1), if successive boot-strappings are carried out with $n = 2^{15}$ slots, then [5] and [19] would reach a $1/2$ failure probability after 52 bootstrappings, whereas ours would reach the same probability after 24,656 bootstrappings.

Figure 3 plots the best performing instances of Table 7.

8   Conclusion

In this work, we have introduced a secure, reliable, precise and efficient boot-strapping 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 reported instance of a practical bootstrapping parameterized for at least 128-bit security.

To achieve this, we have proposed a generic algorithm for the homomorphic evaluation of polynomials with reduced error and optimal in level consumption. In addition to the increase in precision and efficiency, our algorithm 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 have also proposed an improved key-switch format that we apply to the homomorphic matrix-vector multiplication. Our novel *double hoisting* algorithm reduces the complexity of the CoeffsToSlots and SlotsToCoeffs by roughly a factor of 2 compared to previous works. The performance gain for these procedures enables their use outside of the bootstrapping, for applications where the conversion between coefficient- and slot-domains would enable much more efficient homomorphic circuits (e.g., in the training of convolutional neural networks or R-LWE to LWE ciphertext conversion).

We have also proposed a systematic approach to parameterize the bootstrapping, including a way to precisely assess its failure probability. We have evaluated our bootstrapping procedure and have shown that its throughput with "dense" secret-keys ($h = N/2$) is up to $4.6\times$ larger than the best state-of-the-art results with sparse keys ($h = 64$). When the sparse-keys-adjusted parameters of Curtis and Player [11] for $h = 192$ and 128-bits of security are considered, our procedure has a $14.1\times$ larger throughput than the previous work that uses a sparse key with $h = 64$ with insecure parameters. Additionally, all our parameters lead to a more reliable instance than the previous works, with a failure probability orders of magnitude lower.

We have implemented our contributions in the Lattigo library [23]. This is, to the best of our knowledge, the first open-source implementation of a boot-strapping procedure for the full-RNS variant of the CKKS scheme.

Acknowledgments

References

[1]   Martin Albrecht et al. *Homomorphic Encryption Security Standard.* Tech. rep. Toronto, Canada: HomomorphicEncryption.org, Nov. 2018.

[2]   Martin R Albrecht, Rachel Player, and Sam Scott. "On the concrete hardness of learning with errors". In: *Journal of Mathematical Cryptology* 9.3 (2015), pp. 169–203.

[3]   Jean-Claude Bajard et al. "A full RNS variant of FV like somewhat homomorphic encryption schemes". In: *International Conference on Selected Areas in Cryptography.* Springer. 2016, pp. 423–442.

[4]   Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping". In: *ACM Transactions on Computation Theory (TOCT)* 6.3 (2014), pp. 1–36.

[5]   Hao Chen, Ilaria Chillotti, and Yongsoo Song. "Improved bootstrapping for approximate homomorphic encryption". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Springer. 2019, pp. 34–54.

[6]   Jung Hee Cheon, Kyoohyung Han, and Minki Hhan. "Faster Homomorphic Discrete Fourier Transforms and Improved FHE Bootstrapping". In: *IACR Cryptol. ePrint Arch.* 2018 (2018), p. 1073.

[7]   Jung Hee Cheon et al. "A full RNS variant of approximate homomorphic encryption". In: *International Conference on Selected Areas in Cryptography.* Springer. 2018, pp. 347–368.

[8]   Jung Hee Cheon et al. "A hybrid of dual and meet-in-the-middle attack on sparse and ternary secret LWE". In: *IEEE Access* 7 (2019), pp. 89497–89506.

[9]   Jung Hee Cheon et al. "Bootstrapping for approximate homomorphic encryption". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Springer. 2018, pp. 360–384.

[10]   Jung Hee Cheon et al. "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.

[11]   Benjamin R. Curtis and Rachel Player. "On the Feasibility and Impact of Standardising Sparse-secret LWE Parameter Sets for Homomorphic Encryption". In: *Proceedings of the 7th Workshop on Encrypted Computing and Applied Homomorphic Cryptography* (2019).

[12]   Junfeng Fan and Frederik Vercauteren. "Somewhat Practical Fully Homomorphic Encryption." In: *IACR Cryptol. ePrint Arch.* 2012 (2012), p. 144.

[13]   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.

[14]   Craig Gentry, Shai Halevi, and Nigel P. Smart. "Homomorphic Evaluation of the AES Circuit". In: *Advances in Cryptology – CRYPTO 2012.* Ed. by Reihaneh Safavi-Naini and Ran Canetti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 850–867. ISBN: 978-3-642-32009-5.

[15]   Shai Halevi, Yuriy Polyakov, and Victor Shoup. "An improved RNS variant of the BFV homomorphic encryption scheme". In: *Cryptographers' Track at the RSA Conference*. Springer. 2019, pp. 83–105.

[16]   Shai Halevi and Victor Shoup. "Algorithms in HELib". In: *Annual Cryptology Conference*. Springer. 2014, pp. 554–571.

[17]   Shai Halevi and Victor Shoup. "Bootstrapping for HELib". In: *Annual International conference on the theory and applications of cryptographic techniques*. Springer. 2015, pp. 641–670.

[18]   Shai Halevi and Victor Shoup. "Faster homomorphic linear transformations in HElib". In: *Annual International Cryptology Conference*. Springer. 2018, pp. 93–120.

[19]   Kyoohyung Han and Dohyeong Ki. "Better bootstrapping for approximate homomorphic encryption". In: *Cryptographers' Track at the RSA Conference*. Springer. 2020, pp. 364–390.

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

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

[22]   Miran Kim et al. "Ultra-Fast Homomorphic Encryption Models enable Secure Outsourcing of Genotype Imputation". In: *bioRxiv* (2020). DOI: `10.1101/2020.07.02.183459`.

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

[24]   Joon-Woo Lee et al. *High-Precision Bootstrapping of RNS-CKKS Homomorphic Encryption Using Optimal Minimax Polynomial Approximation and Inverse Sine Function*. Cryptology ePrint Archive, Report 2020/552. `https://eprint.iacr.org/2020/552`. 2020. Accepted to Eurocrypt 2021.

[25]   Oliver Masters et al. "Towards a Homomorphic Machine Learning Big Data Pipeline for the Financial Services Sector". In: *IACR Cryptol. ePrint Arch.* 2019 (2019), p. 1113.

[26]   Sinem Sav et al. "POSEIDON: Privacy-Preserving Federated Neural Network Learning". In: *arXiv preprint arXiv:2009.00349* (2020).

[27]   *Microsoft SEAL (release 3.6)*. Online: `https://github.com/Microsoft/SEAL`. Microsoft Research, Redmond, WA. Nov. 2020.

[28]   Yongha Son and Jung Hee Cheon. "Revisiting the Hybrid attack on sparse and ternary secret LWE". In: *IACR Cryptol. ePrint Arch.* 2019 (2019), p. 1019.

[29]   *The Go Programming Language*. Online: `https://golang.org/`. Sept. 2020.

Supplementary Material

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 out 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 $\{\mathsf{ct} + (\lfloor \Delta \cdot (a + b \cdot X^{N/2}) \rceil, 0), Q_\ell, \Delta\}$.

- $\mathsf{Multi}(\{\mathsf{ct}, Q_\ell, \Delta\})$: Return $\{\mathsf{ct} \cdot X^{N/2}, Q_\ell, \Delta\}$, the homomorphic product by the imaginary unit.

- $\mathsf{MultConst}(\{\mathsf{ct}, Q_\ell, \Delta\}, a+bi \in \mathbb{C}, \Delta')$: Return $\{\lfloor \Delta' a \rfloor \cdot \mathsf{ct} + \lfloor \Delta' b \rfloor \cdot \mathsf{ct} \cdot X^{N/2}, Q_\ell, \Delta\Delta'\}$.

- $\mathsf{Mul}(\{\mathsf{ct}, Q_\ell, \Delta\}, \{\mathsf{ct}', Q_{\ell'}, \Delta'\})$: For $\mathsf{ct} = (c_0, c_1)$ and $\mathsf{ct}' = (c_0', c_1')$, compute $(d_0, d_1, d_2) = (c_0 c_0', c_0 c_1' + c_1 c_0', c_1 c_1')$, 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 a 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 [12] propose two key-switch keys types to control this error term:

- *Type 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 increase the number of keys, and therefore the number of operations, by an amount proportional to $Q/||\mathbf{w}||$.
- *Type 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. This also affects the overall performance or the homomorphic capacity.

Han and Ki [19] 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 enables 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 much affecting the homomorphic capacity.

Although 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. However, due to its non-linearity, such a decomposition cannot be used directly when dealing with coefficients in the RNS representation. 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:

$$a \equiv \sum [a]_{q_i} \frac{Q}{q_i} \Big[\Big(\frac{Q}{q_i}\Big)^{-1} \Big]_{q_i} \mod Q. \tag{2}$$

Therefore, 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 [3, 15]. It is also 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 complexity derivations of the algorithms used in our work for the key-switch (Algorithm 3 in Section 4.1) and hoisted rotation (Algorithm 4 in Section 4.2).

C.1   Key-Switch

We analyse the the complexity of a homomorphic multiplication with a key-switch (Algorithm 3 in Section 4.1) in term of the required number of modular multiplication in $\mathbb{Z}_p$, and we compare it with the results reported in [19].

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 $\mathsf{ct}' = (c_0', c_1')$ mod Q_ℓ.

Step 1: *Tensoring.* We compute the tensor product of the ciphertexts (of degree 1): $(\hat{c}_0, \hat{c}_1, \hat{c}_2) \leftarrow (c_0 c_0',\ c_0 c_1' + c_1 c_0',\ c_1 c_1')$ mod Q_ℓ. Intuitively, 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 therefore $4 \cdot N \cdot (\ell + 1)$.

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

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

1. We are given an input vector of $\ell + 1$ elements that we take modulo q_{α_i}, thus reducing its size to α elements. This first operation is free as $q_{\alpha_i} | Q_\ell$. We then extend this vector to a vector of size $(\ell + 1) + \alpha$, for which we already know α elements, hence the complexity is $\alpha \cdot (\ell + 1 + \alpha - \alpha) = \alpha \cdot (\ell + 1)$. We also have to run α pre-computations on the fly. As 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 $d \in R_{Q_\ell}$, hence the total number of NTT is reduced to $\ell + 1$, and 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 change the basis of $d_i \in R_P$ 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. Therefore, 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 is a subtraction and a multiplication by 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: $\mathsf{ct_{mul}} = (d_0 + \hat{c}_0, d_1 + \hat{c}_1) \bmod Q_\ell$.

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

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

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

The complexity reported in [19] is

$$N \cdot \left((\ell+1)^2 + (\ell+1) \cdot (\alpha + 2\beta + 6) + 3 + \log(N) \cdot \left((\ell+1) \cdot (\alpha + \beta + 5) + 3 \right) \right).$$

Table 8 compares both expressions by using the same parameters as the original table of [19] with $\ell = 23$ and a variable $\#p_j = \alpha$. The size of the moduli of Q and P is 45 bits, and q_0 is 55 bits. Our algorithm has the same asymptotic complexity, but it introduces a change in the constants, which is enough to induce a non-negligible difference. The number of NTT operations, which is the dominant term, is $(\ell+1) \cdot (\alpha + \beta + 5) + 3$ in [19], whereas it is $(\ell+1) \cdot (\beta + 3) + 2\alpha$ in our work. The number of NTTs in our algorithm decreases much faster for larger α than those of Han and Ki, e.g., for $\alpha = 6$ it already shows a factor-of-two difference. As the number of NTTs is the dominant term of the key-switch, this translates into a non-negligible difference in the final complexity.

C.2   (Hoisted) Rotations

In this section, we analyse the complexity of hoisted rotations (Algorithm 4 in Section 4.2) in term of their number of modular multiplications in $\mathbb{Z}_p$. 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, then

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: $\beta \cdot N \cdot (\alpha + (\ell+1) \cdot (\log(N) + \alpha))$.

Table 8: Comparison of the homomorphic multiplication complexity in log.

		$\log(\#\text{Mul in } \mathbb{Z}_p)$	
α	$\log(QP)$	Work in [19]	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

Step 3: MultSum. For each rotation r_k we compute the dot product between $\mathbf{d}$ and $\mathsf{rot}_{k,q_{\alpha_i}}$: $2k\beta \cdot N \cdot (\ell + 1 + \alpha)$.

Step 4: ModDown. For each rotation r_k we rescale a and b by P and reduce the RNS basis from PQ_ℓ 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 rotation r_k we apply the automorphism ϕ_k on $c_0 + a$ and b: there is no multiplication.

Hence, the total complexity for r hoisted rotations is

$$N \cdot \left(\log(N) \cdot \Big((\ell+1) \cdot (\beta + 1 + 2r) + 2r\alpha \Big) + (\ell+1) \cdot \Big(\beta\alpha + 2r \cdot (\alpha + \beta + 1) \Big) + \alpha \cdot \Big(\beta + 2r \cdot (\beta + 1) \Big) \right).$$

Figure 4 compares the complexity of regular and hoisted rotations for a varying number of rotations r and ciphertext level ℓ. It shows that using hoisted rotations 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, as the first step of this procedure is a linear transformation computed at the maximum ciphertext level.

D Performance of Basic Operations

Table 9 reports the single-thread performance in ms of Lattigo v2.1.1 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 is not changed when α varies). The timings for the $\mathsf{ct} \times \mathsf{ct}$ multiplication are reported without the relinearization (key-switch). The benchmarks were conducted single threaded on an i5-6600k at 3.5 GHz with 32Gb of RAM running Windows 10 and Go 1.15.6, GOARCH=amd64, GOOS=windows.
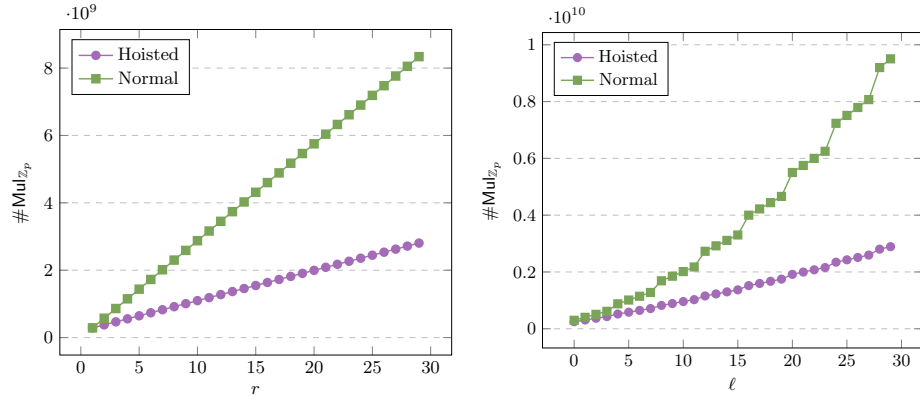
(a) Varying number of rotations r. Parameters: $\{N=2^{16}, \ell=21, \alpha=4, \beta=\lceil(\ell+1)/\alpha\rceil\}$.

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

Fig. 4: Complexity of hoisted rotations vs. non-hoisted rotations in terms of the number of multiplications in $\mathbb{Z}_p$.

Table 9: Performance of basic operations in Lattigo. Timings are in [ms]

lvls	α	$\mathsf{Enc}_{\mathsf{pk}}$	$\mathsf{Enc}_{\mathsf{sk}}$	Dec	Add	$\mathsf{Mul}_{\mathsf{pt}}$	$\mathsf{Mul}_{\mathsf{ct}}$	ϕ	KeySwitch	Rescale
29	1	205	124	9	4	8	19	7	1121	72
28	2	208	124	8	4	8	18	6	692	58
27	3	208	118	8	4	8	17	6	496	55
25	5	215	113	7	3	7	16	6	349	50
24	6	214	106	7	3	7	16	5	311	49
20	10	210	92	6	3	6	12	4	233	41
15	15	204	72	4	2	4	10	3	181	30

## E Bootstrapping Stability Experiments

This section reports on the following checks: the mean precision across all the slots against the number of slots (Supplementary material E.1), the probability of each slot to fall under some given precision (Supplementary material E.2), and the mean precision across all the slots after each bootstrapping for 50 successive bootstrapping operations (Supplementary material E.3).

E.1 Precision vs. Slots

We plot in Figure 5 the mean precision for different values n (slots) for the parameter sets I, II, III, IV and V of Table 5 in Section 6.3. The plaintext values are of the form $a + bi$, for a, b random reals between -1 and 1. The comparison is made against a non-encoded plaintext vector, hence these results also include the error of the encoding algorithms. The parameter sets are optimized to give

the best performance for $n = 2^{15}$ slots, hence a smaller amount of slots will not necessarily imply a better precision. To get a better precision with a smaller n, the moduli of the linear transforms and the ratio $Q_0/||m||$ should be increased.
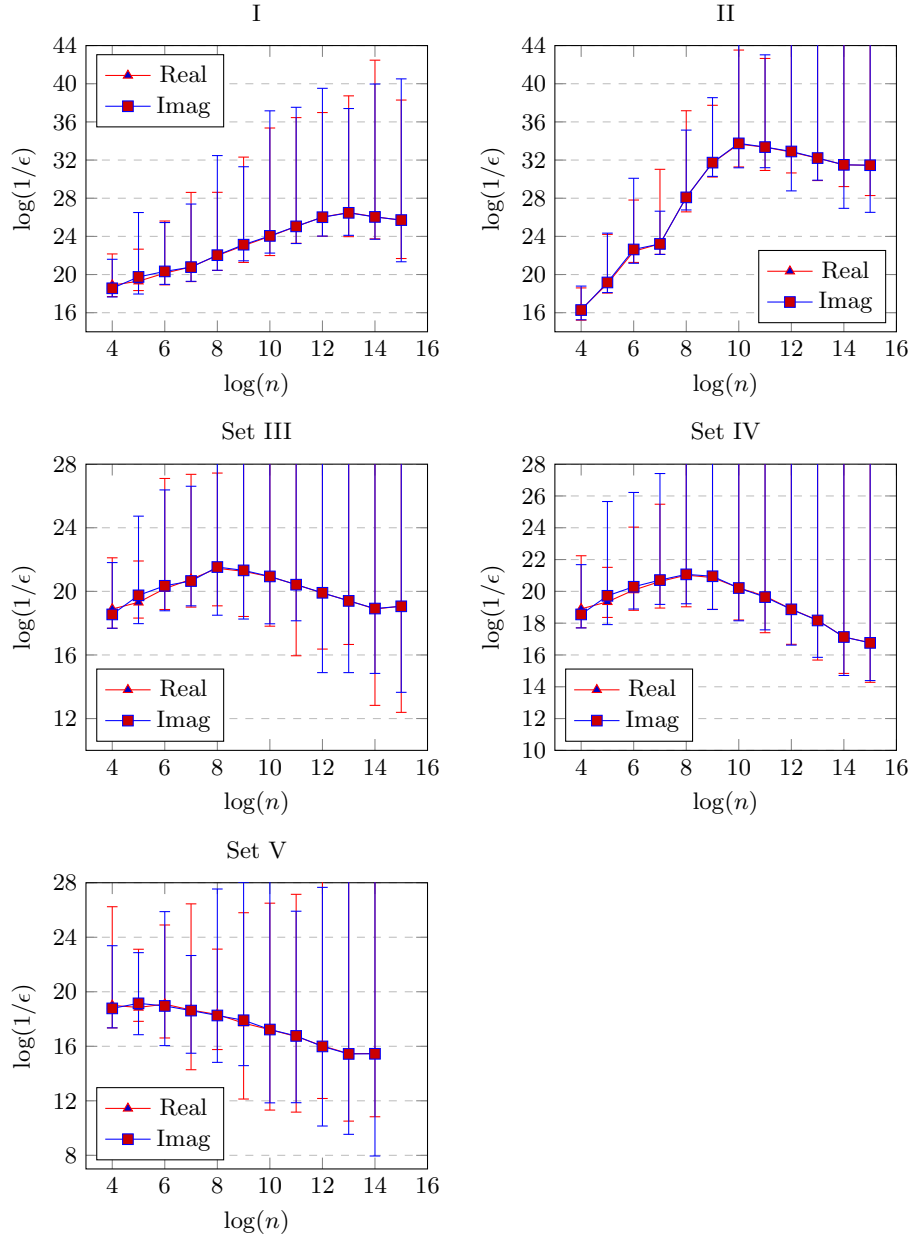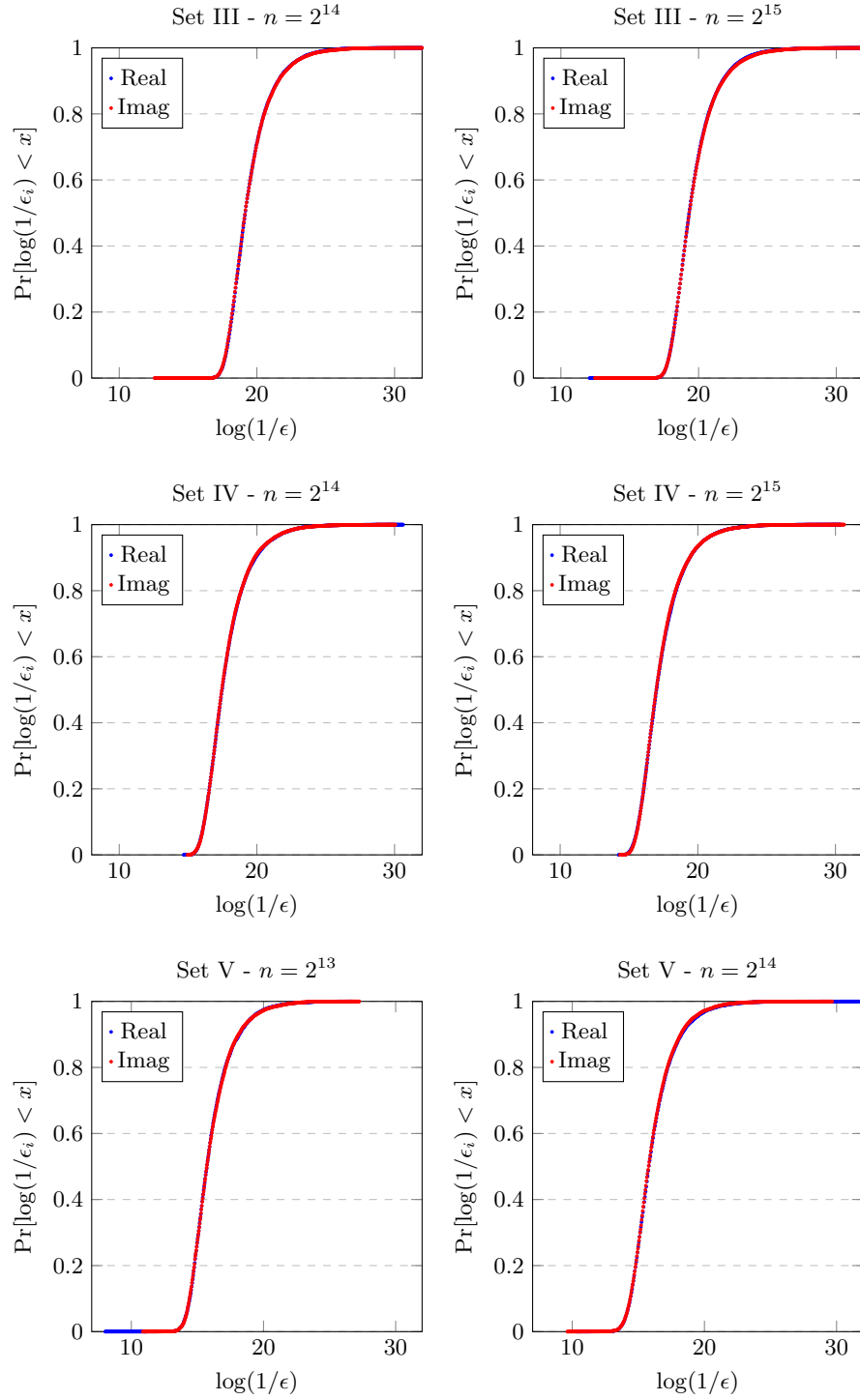

Fig. 5: Precision vs. number of slots for the parameter sets I, II, III, IV and V of Table 5 in Section 6.3.

E.2 Precision Distribution

In this section, we plot in Figures 6 and 7 the CDF of the precision for the parameter sets I, II, III, IV and V of Section 6.3. The goal is to show that though we have a good mean precision, the overall distribution also behaves well and is not scattered. The plaintext values are of the form $a + bi$, for a, b random reals between -1 and 1. The comparison is made against a non-encoded plaintext vector, hence these results also include the error of the encoding algorithms. The shapes of all the plots show that the distribution is smooth across all the different parameter sets of Table 5 and that 99% of the values are within $\pm$ 2 bits of the mean.
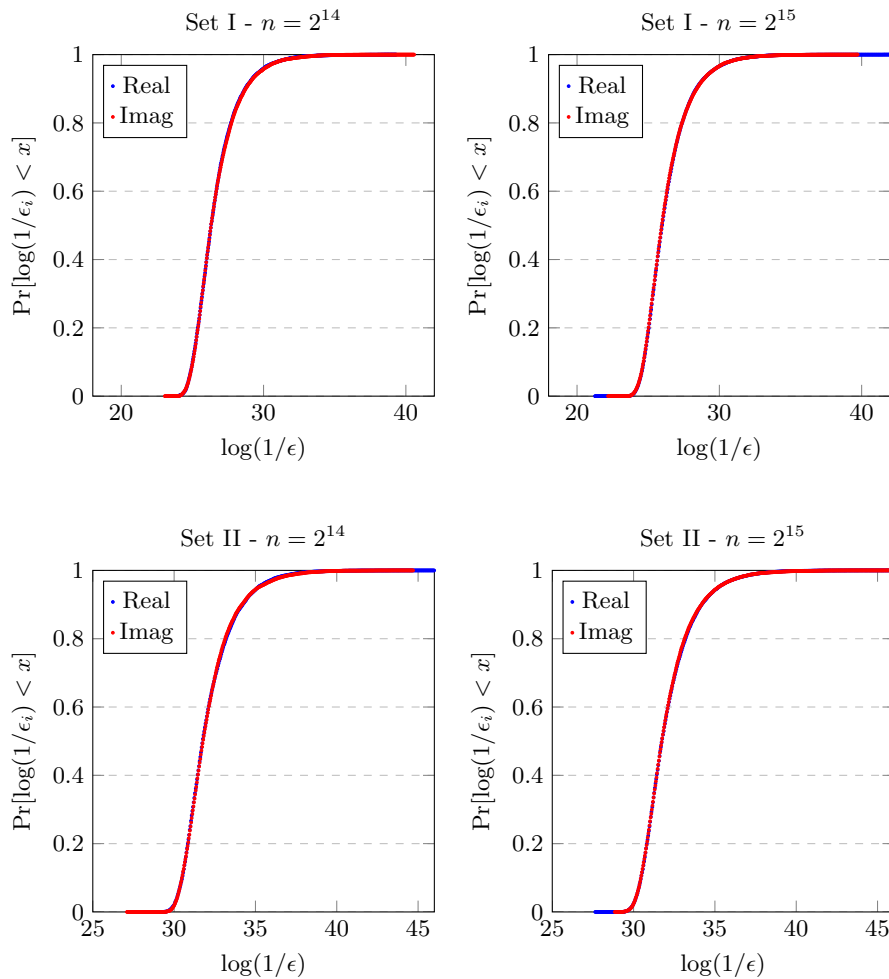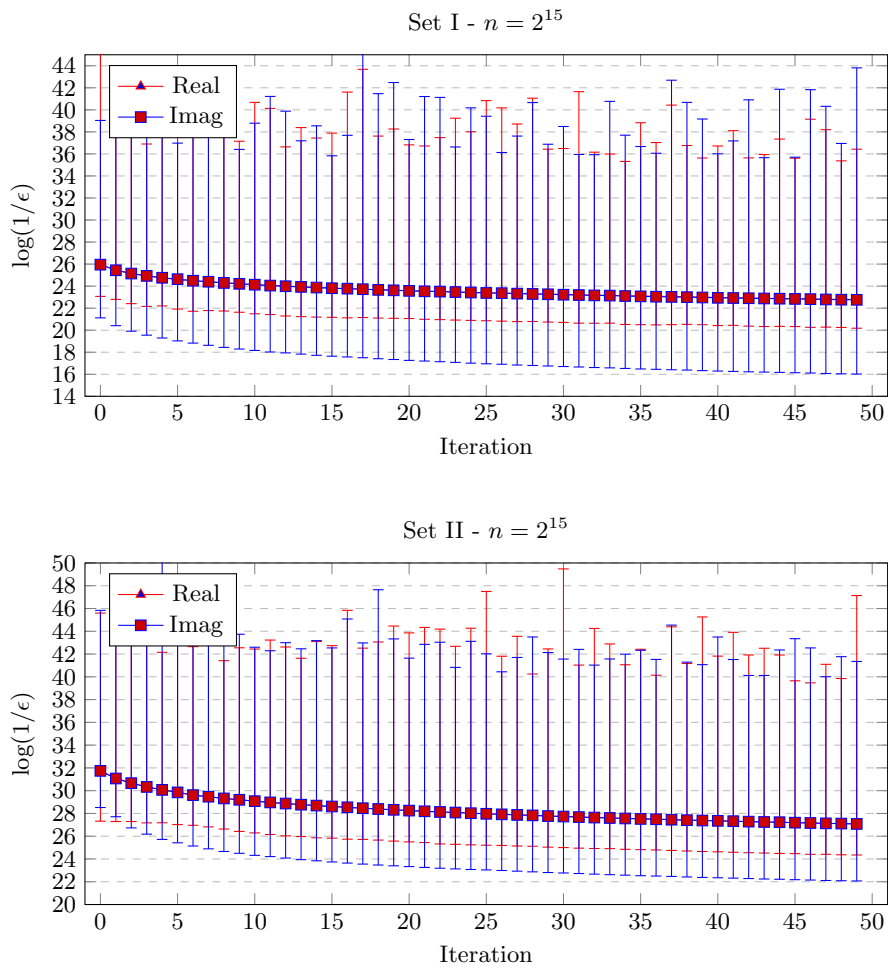

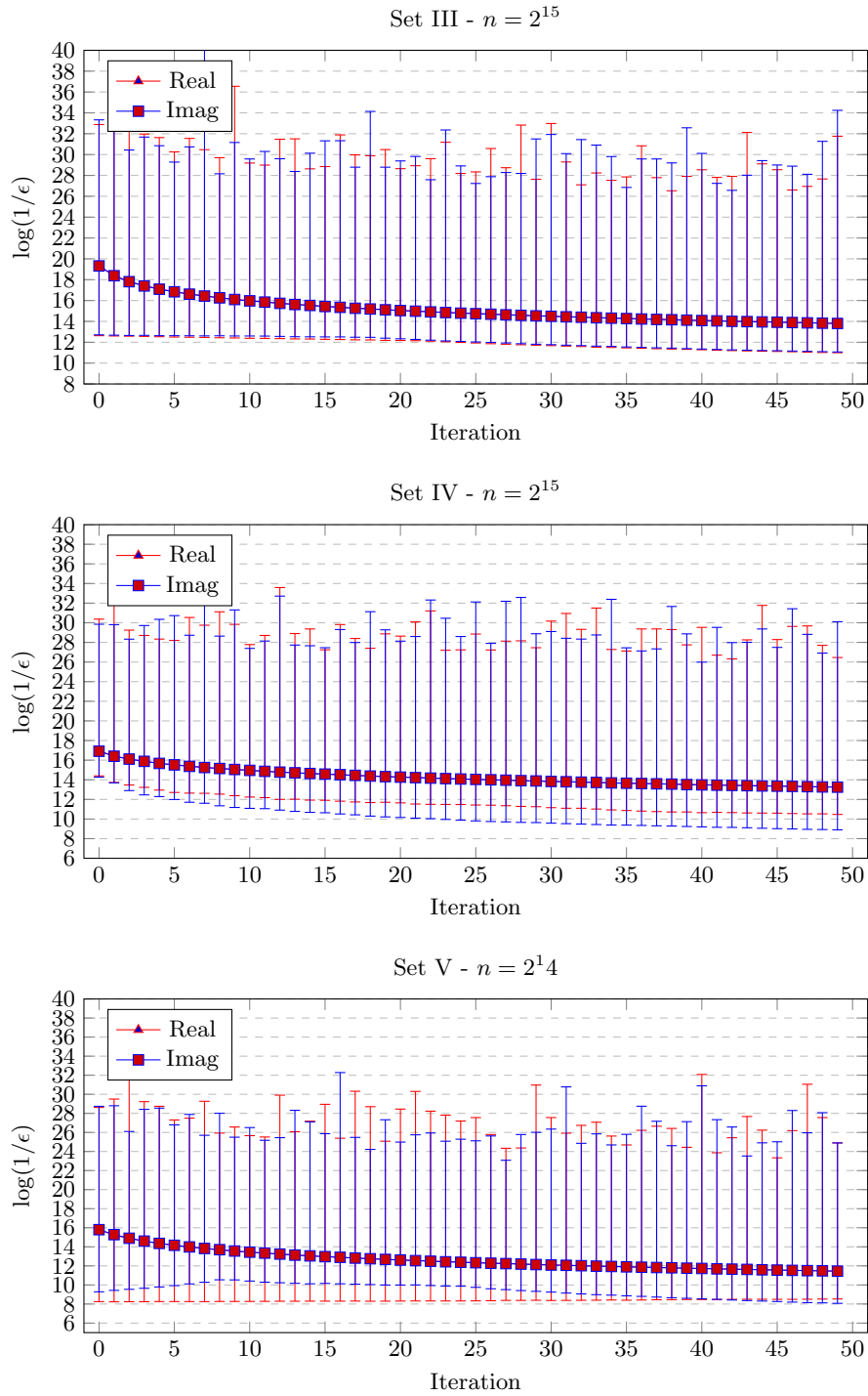**Fig. 6:** CDF of the precision for the parameters sets I and II of Table 5 in Section 6.3.

Fig. 7: CDF of the precision for the parameters sets III, IV and V of Table 5 in Section 6.3.

44

E.3   Successive Bootstrapping Operations

In this section, we plot in Figures 8 and 9 the plaintext precision values after
each bootstrapping with 50 iterations for the parameter sets I, II, III, IV and V
of Section 6.3. The plaintext values are of the form $a + bi$, for a, b random reals
between -1 and 1. The plots show the mean precision, along with the absolute
upper and lower precision bound (no value had a larger or smaller precision).
Note that the comparison is made against a non-encoded plaintext vector, hence
these results also include the inherent error of the encoding algorithms. We
observe a logarithmic decrease in the precision loss, that is consistent with an
additive error.

Set I - $n = 2^{15}$

Set II - $n = 2^{15}$

Fig. 8: Mean precision after successive bootstrapping operations for the param-
eter sets I and II of Table 5 in Section 6.3.

Set III - $n = 2^{15}$

Set IV - $n = 2^{15}$

Set V - $n = 2^{1}4$

Fig. 9: Mean precision after successive bootstrapping operations for the parameter sets III, IV and V of Table 5 in Section 6.3.

F   Bootstrapping Diagram

We include here a graphical description (Figure 10) of the implementation of our bootstrapping circuit in the form of a flow diagram.
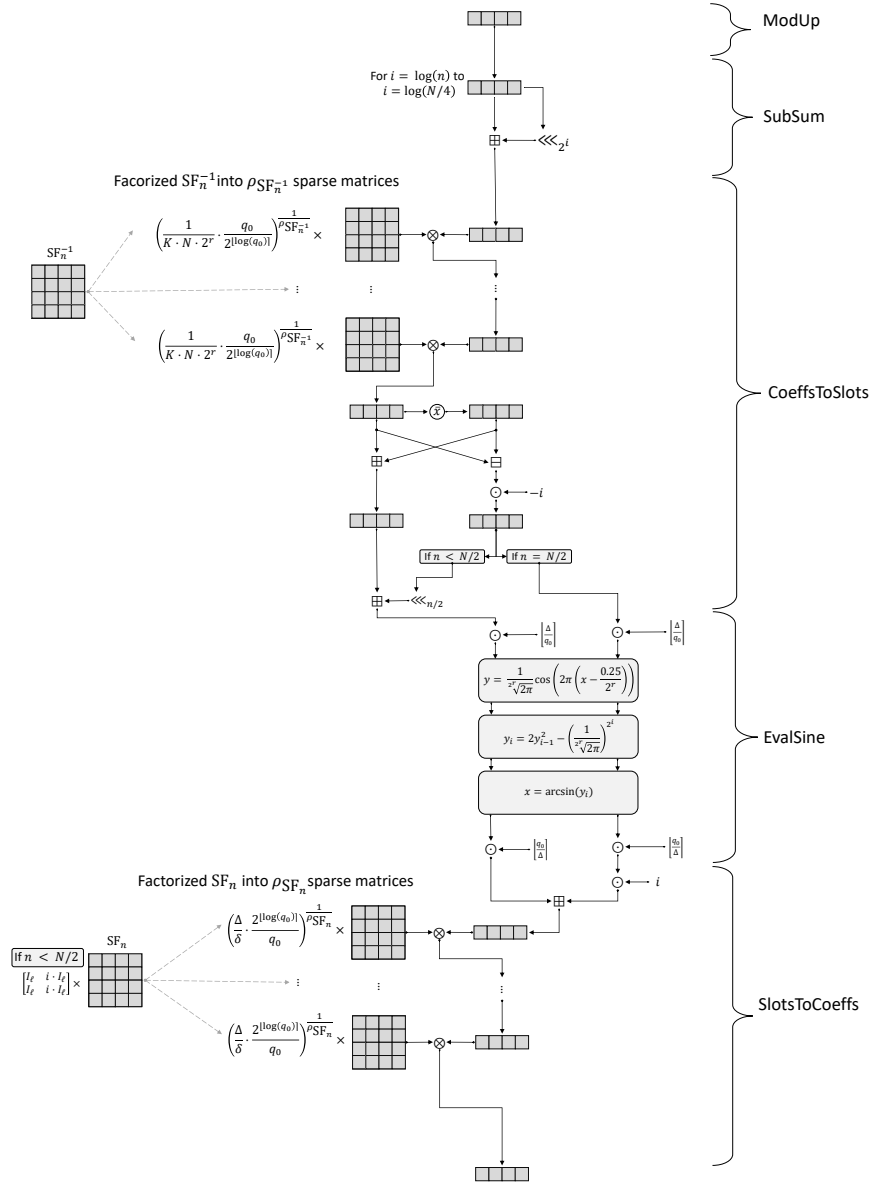

Fig. 10: Bootstrapping circuit diagram.