Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-Sparse Keys

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

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

Abstract. We present a bootstrapping procedure for the full-RNS variant of the approximate homomorphic-encryption scheme of Cheon et al., CKKS (Asiacrypt 17, SAC 18). Compared to the previously proposed procedures (Eurocrypt 18 & 19, CT-RSA 20), our bootstrapping procedure is both more precise and more efficient, in terms of CPU cost and number of consumed levels. 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 for parameters that are 128-bit-secure under more recent attacks on sparse R-LWE secrets.
We achieve this by introducing two novel contributions applicable to the CKKS scheme: (i) We propose a generic algorithm for homomorphic polynomial-evaluation that is scale-invariant and optimal in level consumption. (ii) We optimize the key-switch procedure and propose a new technique for performing rotations (*double hoisting*); it significantly reduces the complexity of homomorphic matrix-vector products.
Our scheme improvements and bootstrapping procedure are implemented in the open-source Lattigo library. For example, bootstrapping a plaintext in $\mathbb{C}^{32768}$ takes 17 seconds, with an output coefficient modulus of 505 bits and a mean precision of 19.2 bits. Thus, we achieve an order of magnitude improvement in bootstrapped throughput (plaintext-bit per second) with respect to the previous best results, while ensuring 128-bit of security.

Keywords: Fully Homomorphic Encryption, Bootstrapping, Implementation

1 Introduction

Homomorphic encryption (HE), as it enables computing over encrypted data without decrypting them first, 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 [22, 20, 23], and some of them are now proposed as an industry standard [2].

Cheon et al. [10] introduced a *leveled* encryption scheme for approximate arithmetic (CKKS) that enables the homomorphic evaluation of arbitrary polynomial functions over encrypted complex-number vectors. Although the family of *leveled* cryptosystems enables only a finite multiplicative depth, each multiplication *consuming* one level, the CKKS scheme enables the homomorphic re-encryption of an exhausted ciphertext into an almost *fresh* one. This capability, 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 [18] that operates on the more recent full-RNS (residue number systems) version of CKKS, the bootstrapping efficiency has improved by several orders of magnitude. However, the bootstrapping operation would still be a bottleneck for its potential applications, and improving this procedure is crucial for the adoption of the scheme. The performance of the bootstrapping can be improved by following two approaches: (i) adapting its the circuit representation by using HE-friendly numerical methods. (ii) optimizing the scheme operations themselves, which also impacts the scheme performance in general.

All bootstrapping approaches so far [9, 18, 5] rely 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 [25, 8]. The lack of stability in the security of sparse keys has lead the standardization initiatives to exclude sparse secrets hence also the bootstrapping operation from the currently proposed standards [2]. This raises the question about the practicality of a bootstrapping procedure that would not require the use of sparse secret-keys. In this work, we present a constructive proof to Proposition 1:

Proposition 1. *There exists a bootstrapping procedure for the full-RNS CKKS encryption scheme that does not require the use of sparse secret-keys and retains at least the same utility (Definition 1 in Section 7) as current methods.*

1.1 Our Results

We propose a bootstrapping procedure for the full-RNS CKKS scheme that is not constrained to sparse secret-keys. To achieve that, we make the following contributions: (i) a scale-invariant procedure for the homomorphic evaluation of polynomial functions, that prevents the precision loss due to the approximate rescale, and (ii) a faster key-switch procedure that, when applied to the homomorphic matrix-vector product, significantly reduces the cost of computing linear transformations over ciphertexts. Even though we present these contributions as building blocks of our bootstrapping procedure, they are generic for the CKKS cryptosystem and have applications beyond the bootstrapping.

We show that our bootstrapping with standard "dense" keys has a throughput (in bootstrapped plaintext bits per second) 4.4 larger than the most recent work from Han and Ki [18] that uses sparse keys (on equivalent hardware). Furthermore, when using the sparse-key-adjusted parameters of Curtis and Player [11] for sparse-secrets, our throughput is 15 larger. We now briefly describe the high-level ideas behind our contributions.

Scale-Invariant Homomorphic Polynomial Evaluation The full-RNS variant of the CKKS scheme restricts the rescale 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 the number theoretic transform (NTT), the rescale cannot be done by a power of two (as in the original CKKS scheme with power-of-two moduli) and it introduces a small scale deviation in the process. For complex circuits, such as the polynomial-approximation evaluation required by the bootstrapping procedures, additions between ciphertexts of different scales will eventually occur and will introduce errors. As a partial solution, one could first scale both ciphertexts to a common scale, before adding them together, but this would lead to a sub-optimal level-consumption. For this reason, current state-of-the-art approaches have to trade off between precision and optimal level-consumption.

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

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

Faster Matrix $\times$ Ciphertext Products The most expensive homomorphic operation of the CKKS scheme, regardless of its variant, is the key-switch operation (that we detail later). 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), hence minimizing the number of key-switch and/or their complexity will have a significant effect on bootstrapping.

Given an $n \times n$ plaintext matrix M and an encrypted vector $\mathbf{v}$, all recent previous works use a baby-step giant-step (BSGS) algorithm to compute the encrypted product $M\mathbf{v}$ [9, 18, 5] in $O(\sqrt{n})$ rotations. These works treat the key-switch procedure as a black-box and focus on reducing the number of times it is executed.

We improve this BSGS algorithm by proposing a new format for rotation keys and a modified key-switch procedure (Section 4) that enables us to add different levels of *hoisting*[1]. 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 procedure, it reduces the cost of the CoeffsToSlots and SlotsToCoeffs steps (Section 5.3) by roughly a factor of 3.

Improved Bootstrapping Procedure We integrate our proposed improvements in the bootstrapping circuit proposed by Cheon et al. [9], Chen et al. [5], Cheon et al.

[1]The general concept of *hoisting* [17] refers to the ability to take advantage of the commutativity, distributivity, and the redundancy of some operations in order to factor them out and to reduce the number of times they need to be computed.

[6] and Han and Ki [18]. We propose a new scale-invariant, high-precision and faster bootstrapping circuit with updated parameters that are 128-bit secure, even when considering the most recent attacks on sparse keys [8, 25]. We detail our approach for each step of the improved bootstrapping in Section 5.

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

2 Background and Related Work

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

2.1 The Full-RNS CKKS Scheme

We consider the CKKS encryption scheme [10] in its full-RNS variant [7], in which the polynomial coefficients are always represented in the RNS and NTT domains, thus enabling fast arithmetic. The conversion between the RNS and positional domain is costly due to multi-precision arithmetic, but it is only required during the encoding and decoding of plaintext values. This yields an overall acceleration across all operations of one order of magnitude, compared to the original scheme with equivalent parameters.

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

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

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

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

Scheme **RNS-CKKS** { Basic Operations

$\mathsf{Setup}(N; h; b; \ )$: For a power-of-two ring degree N, a secret-distribution hamming-weight h, a standard deviation $$, and a modulus bit-size b: Select the moduli chains $\{q_0, \ldots, q_L\}$ and $\{p_0, \ldots, p_{\ 1}\}$ composed of pairwise different NTT-friendly primes close to powers of two such that $\log(\prod_{i=0}^{L} q_i \prod_{j=0}^{\ 1} p_j) \approx b$. In other words, primes of the form $q_i = 2^{s_i} + k_i 2N + 1$ (hence, $q_i \equiv 1 \mod 2N$) for s_i a positive integer size and k_i a small integer such that q_i is a prime. Set $Q_L = \prod_{i=0}^{L} q_i$, $P = \prod_{j=0}^{\ 1} p_j$.
Define the following distributions over R: $_{\mathsf{key}}$ where the coefficients are uniformly distributed over $\{\ 1; 0; 1\}$ and where exactly h coefficients are non-zero. $_{\mathsf{pkenc}}$ where the coefficients are distributed over $\{\ 1; 0; 1\}$ with respective probabilities $\{1/4; 1/2; 1/4\}$. $_{\mathsf{err}}$ where the coefficients are distributed according to a discrete Gaussian distribution with standard deviation $$ and truncated to $[\ b6\ c; b6\ c]$.

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

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

$\mathsf{SecKeyGen}(\ )$: Sample $s \leftarrow_{\mathsf{key}}$ and return the secret key s.

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

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

$\mathsf{Enc}(\mathsf{fpt}; Q_\ell; \ g; s)$: Sample $a \in_u R_{Q_\ell}$ and $e \leftarrow_{\mathsf{err}}$, set $\mathsf{ct} = (\ as + e; a) + (\mathsf{pt}; 0)$ and return $\mathsf{fct}; Q_\ell; \ g$.

$\mathsf{PubEnc}(f\mathsf{pt}; Q_\ell; \quad g; \mathsf{pk})$: Sample $u \quad _{\mathsf{pkenc}}$ and $e_0; e_1 \quad _{\mathsf{err}}$, set:
$\mathsf{ct} = \mathsf{SwitchKey}(u; \mathsf{pk}) + (\mathsf{pt} + e_0; e_1)$ and return $f\mathsf{ct}; Q_\ell; \quad g$.

$\mathsf{SwitchKey}(d; \mathsf{swk}_{s \,!\, s'})$: For $d \; 2 \; R_{Q_\ell}$ a polynomial[2], decompose d base $\mathbf{w}$ such that $d = \sum b_{\mathbf{w}}^{(i)} w^{(i)}$ and return $(d_0; d_1) = b P^{\;\;1} \sum b_{\mathbf{w}}^{(i)} \mathsf{swk}_{s \,!\, s'}^{(i)} e \bmod Q_\ell$, for $P^{\;\;1} \; 2 \; \mathbb{R}$.

$\mathsf{Dec}(f\mathsf{ct}; Q_\ell; \quad g; s)$: For $\mathsf{ct} = (c_0; c_1)$, return $f\mathsf{pt} = c_0 + c_1 s; Q_\ell; \quad g$.

Given the vectors $\mathbf{m}; \mathbf{a} = f(\mathbf{m})$ and $\mathbf{b} = \mathsf{Decode}(\mathsf{Decrypt}(f(\mathsf{Encrypt}(\mathsf{Encode}(\mathbf{m})))))$ 2 $\mathbb{C}^n$, with $f(\,)$ a polynomial function, we denote $\log(1/\,_{\mathrm{real}})$ the precision of $\mathbf{b}$ relative to $\mathbf{a}$ (the negative log of the average of their absolute difference), i.e. $\log(1/\,_{\mathrm{real}})$ and $\log(1/\,_{\mathrm{imag}})$ where $\;= \frac{1}{n} \sum_{i=0}^{n\;\;1} j a^{(i)} \quad b^{(i)} j$. For convenience, we detail the homomorphic operations of CKKS in Supplementary material 1.

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 $L \quad k > 0$ (where k is the depth of the bootstrapping circuit) such that $Q_{L \quad k} \quad Q_0$ and $[c_0' + c_1' s]_{Q_{L \quad k}} \quad \mathsf{pt}$. Since $[c_0 + c_1 s]_{Q_L} = \mathsf{pt} + Q_0 \;\; I$, where I is an integer polynomial [9], bootstrapping is equivalent to an extension of the CRT basis followed by an homomorphic reduction of each coefficient of $\mathsf{pt} + Q_0 \;\; I$ 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:

$$\underbrace{\mathrm{FFT}^{\;\;1}(\mathsf{pt} + Q_0 \;\; I) = \mathsf{pt}'}_{\text{(i) } \mathsf{Encode}(\mathsf{pt} + Q_0 \;\; I)} \quad \rightarrow \quad \underbrace{\frac{Q_0}{2} \sin \frac{2 \;\; \mathsf{pt}'}{Q_0} = \mathsf{pt}''}_{\text{(ii) } \mathsf{EvalSine}(\mathsf{pt}')} \quad \rightarrow \quad \underbrace{\mathrm{FFT}(\mathsf{pt}'') \quad \mathsf{pt}}_{\text{(iii) } \mathsf{Decode}(\mathsf{pt}'')}.$$

The complexity of the resulting bootstrapping circuit is influenced by two parameters: The first one is the secret-key hamming weight h, which directly impacts the depth of the bootstrapping circuit. Indeed, Cheon et al. show that $jjIjj \quad O(\sqrt{h})$ with very high probability. A denser key will therefore require evaluating a larger-degree polynomial, with a larger depth. The second parameter is the number of plaintext slots n, which has a direct impact on the complexity of the circuit (but not on its depth). By scaling down the values to compress them closer to the origin, Cheon et al. are able to evaluate the sine function by using a low-degree Taylor polynomial approximation of the complex exponential and then use repeated squaring (the double angle formula) to obtain the correct result. In their approach, the sine evaluation dominates the depth of the bootstrapping circuit while the homomorphic evaluation of the encoding and decoding algorithms dominates the circuit's width. In fact, the linear transformations are the main bottleneck of the bootstrapping because the number of operations grows

[2] $\mathsf{SwitchKey}$ 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.

linearly with the number of plaintext slots. For example, bootstrapping a ciphertext that packs $n = 2^{15}$ plaintext slots would take more than a day, with less than a minute devoted to the evaluation of the scaled sine function.

This issue was later addressed by Chen et al. They proposed an improved procedure to compute the homomorphic evaluation of the encoding and decoding transformations [5]. They find that these transformations are direct/inverse discrete Fourier transforms (DFTs) that can be efficiently computed by using the Cooley-Tukey algorithm. Chen et al. show that the corresponding homomorphic circuit has depth $\log(n)$, i.e., the number of iterations in the Cooley-Tukey algorithm, because each iteration of the algorithm has a multiplicative depth of one, with only two key-switch operations (see Section 5.3). Chen et al. merge several layers of the Cooley-Tukey algorithm to reduce the number of iterations (the circuit depth) at the cost of additional complexity per iteration, and they discuss the introduced trade-off. In a concurrent work, Cheon et al. [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] and [6] enabled the acceleration of the homomorphic evaluation of the encoding algorithms by two orders of magnitude. Chen et al. [5] also improved the approximation of the scaled sine function with a near optimal depth method by using a Chebyshev interpolant.

In a more recent work, Han and Ki port the bootstrapping procedure to the full-RNS variant of CKKS and propose several improvements to the bootstrapping circuit and to the CKKS scheme [18]. They propose a generalization of its key-switch procedure by using an intermediate RNS decomposition that enables a finer-grained selection of the scheme parameters and trade-offs between the complexity of the key-switch and the homomorphic capacity of a fresh ciphertext. They also provide an alternative way of computing the scaled sine function of the bootstrapping circuit, which takes the magnitude of the underlying plaintext into account and uses the cosine function along with the double angle formula. Their technique requires an amount of non-scalar multiplications smaller than the technique of Chen et al. [5]. They show that these changes, combined, lead to an acceleration factor of 2.5 to 3, compared to the work of Chen et al. [5].

Regarding the implementation, both [6] and [5] were implemented with HEAAN [19], but the code of only the former was published. The work of [18] was implemented using SEAL [24], but the code has not been published yet.

2.3 Security of Sparse Keys

One commonality between all the aforementioned works is the use of a sparse secret-keys with a hamming weight $h = 64$. A key with a small number of non-zero coefficients is convenient, as it enables a low-depth bootstrapping circuit, hence was essential for its practicality. However, recent advances in the cryptanalysis of the R-LWE problem have demonstrated that hybrid attacks specifically targeting such sparse keys can severely affect its security [25, 8]. Curtis and Player [11] discussed the practicality and usability of sparse keys. In light of the most recent attacks, they estimate that, for a sparse key with $h = 64$ and a ring degree $N = 2^{16}$, 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 $fN = 2^{16}$, $\log(Q) = 2480$, $h = 64$, $= 3:2g$ and estimate the security of these parameters to 80 bits. In their work, Han and Ki [18] propose new parameter sets, one of which

they claim has 128-bit of security: $fN = 2^{16}$, $\log(Q) = 1450$, $h = 64$, $= 3{:}2g$. However, these estimates are based on results obtained using Albrecht's estimator [1] that, at the time, did not take into account the most recent attacks on sparse keys. In more recent work from Son and Cheon, the security of the parameter set $fN = 2^{16}$, $\log(Q) = 1250$, $h = 64$, $= 3{:}2g$ is estimated at 113 bits. This sets a loose upper bound to security of the parameters (which have a 1450-bit modulus) proposed by Han and Ki. 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. The induced performance penalty is, however, more than compensated by the gain associated with our proposed procedures of Sections 3, 4 and 5.

3 Scale-Invariant Polynomial Evaluation

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

Approximate Rescale The main disadvantage of the full-RNS variant of the CKKS scheme stems from its rescale operation 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 of the form $2^s + k2N + 1$ where s is a positive integer, k a non-zero integer and N the ring degree. Under this constraint, the power-of-two rescale of the original CKKS scheme can only be approximated. In order to avoid introducing errors, the exact ciphertext scale needs to be tracked, and the rescale needs to be treated as an exact division. As a result, ciphertexts at the same level can have slightly different scales (depending on the previous homomorphic operations and the level at which they were previously rescaled). The exact scale resulting from the homomorphic sum between two ciphertexts at different scales is undefined, thus forcing the user to reconcile these scales.

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

Early rescale: Rescale the result of ct_1 ct_2, then add ct_3. As the additions between two ciphertexts of different scales is not defined, we choose to set the scale to the maximum of the two different scales (assuming that their difference is less than 1). This option ensures an optimal level-consumption, as the resulting level will be $\min(\min(`_1; `_2) \quad 1; `_3)$. However, it introduces an error proportional to the difference between the scale of $\mathsf{rescale}(\mathsf{ct}_1 \quad \mathsf{ct}_2)$ and ct_3.
Late rescale: Scale ct_3 upward to (exactly) match the scale of ct_1 ct_2, add ct_3, then rescale the result. As the two scales match, this option results in no error; but it will not ensure an optimal level consumption, as the resulting level will be $\min(\min(`_1; `_2) \quad 1; `_3 \quad 1)$.

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

Input: $p(u) = \sum_{i=0}^{d} c_i T_i(u)$, a degree-$d$ polynomial and a point t.
Output: The evaluation of $p(u)$ at the point $u = t$.

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

Dynamic rescale: Apply *Early rescale* if $\min(\ell_1, \ell_2) > \ell_3$, and apply *Late rescale* otherwise. This option ensures an optimal level-consumption, while minimizing the number of additions between ciphertexts at different scales (yet, not completely preventing them).

The choice of the scale-matching strategy significantly affects the output precision and level-consumption when homomorphically evaluating a polynomial of large degree with the baby-step-giant-step approach (described below). This is highly relevant for our bootstrapping procedure, where relaxing the constraint on sparse keys requires a polynomial evaluation of degree ≥ 250.

Baby-Step Giant-Step (BSGS) Algorithm In order to minimize the number of ciphertext-ciphertext multiplications in their bootstrapping circuit, Han and Ki [18] adapt a generic baby-step giant-step polynomial-evaluation algorithm polynomials in Chebyshev basis. Algorithm 1 gives a high-level description of the procedure.

For a polynomial $p(t)$ of degree d, with $m = \lceil \log(d+1) \rceil$ and $l = \lfloor m/2 \rfloor$, the algorithm first decomposes $p(t)$ into a linear combination of $u_i(t) = \sum_{j=0}^{2^l-1} c_{i;j} T_j(t)$, with $c_{i;j} \in \mathbb{C}$ and $T_{0 \le j \le 2^l}$ a pre-computed power basis, such that $p(t) = \sum_{i=0}^{\lfloor d/2^l \rfloor} u_i(t) T_{i \cdot 2^l}(t)$. We denote $u_{\lfloor d/2^l \rfloor}(t)$ as $u_{\max}$. The BSGS algorithm then recursively combines the monomials $u_i^0(t) = u_{i+1}(t) \cdot T(t) + u_i(t)$ in a tree-like manner by using a second pre-computed power basis $T_{2^{l \cdot i \cdot m}}(t)$ to minimize the number of non-scalar multiplications. The algorithm requires $2^l + m - l - 3 + d(d+1)/2^l$ non-scalar products and has, in the best case, depth m.

We implemented Algorithm 1 for each scale-matching strategy. Table 1 reports the empirical effect of each strategy on the output precision for the homomorphic evaluation of the function $f(x) = \cos(2\pi(x - 0.25)/2^r)$ followed by r evaluations of the double angle formula $\cos(2x) = 2\cos^2(x) - 1$. 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). We observe that the dynamic approach successfully guarantees an optimal level-consumption and provides a close-optimal precision, but it cannot prevent a scale

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

			log(1/) for $(K; d; r)$				
	Rescale		(12, 34, 2)	(15, 40, 2)	(17, 44, 2)	(21, 52, 2)	(257, 250, 3)
Alg. 1 ([18])	*Early*	$2^{31:44}$	30.36	30.05	29.73	29.19	25.00
	Late	$2^{28:48}$	37.52	37.80	37.40	37.05	29.41
	Dynamic	$2^{28:84}$	37.52	37.75	36.24	31.06	29.43
Alg. 2 (ours)	*Early*	0	37.37	37.16	37.15	37.04	29.46
	Dynamic	0	37.63	37.75	37.50	37.05	29.43

deviation of $\ 2^{30}$ (when normalized). This deviation is significant, with respect to the output precision of 2^{37}, and represents a loss of $\ 7$ bits in subsequent additions.

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

3.1 Scale-Invariant Polynomial Evaluation

Our solution to the problem of the rescale error for the evaluation of a polynomial $p(t)$ is to scale each of the coefficients of the $u_i(t)$ terms of the decomposed polynomial $p(t)$ by some value $_i$ such that, during all the steps of the polynomial evaluation, the rescales are exact and the additions are done between ciphertexts with matching scales. From the output scale $_0$ we compute each intermediary scales $_i$ as the polynomial $p(t)$ is recursively split into $p(t) = q(t)T_{2^i}(t) + r(t)$. Note that, for each recursion, $_{i+1}$ can be computed from $_i$ knowing which T_{2^i} and which q_i for the rescaling will be used. Hence, at the end of the recursion, we know the scale $_i$ that each $u_i(t) = \sum_{j=0}^{2^{l\ 1}} c_{i;j}T_j(t)$ must have, and we can thus scale the coefficients $c_{i;j}$ by the appropriate $_{i;j}$. We illustrate our approach with a toy example $p(t)$ of degree $d = 15$ in Figure 1.

Algorithm 2 is our proposed algorithm for polynomial evaluation: it integrates our scale-propagation technique to the recursive decomposition of $p(t)$ into $q(t)$ and $r(t)$. The final scale of the ciphertext, after the polynomial evaluation, either remains unchanged or matches a desired scale. For this reason, we denote this method *scale-invariant*.

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

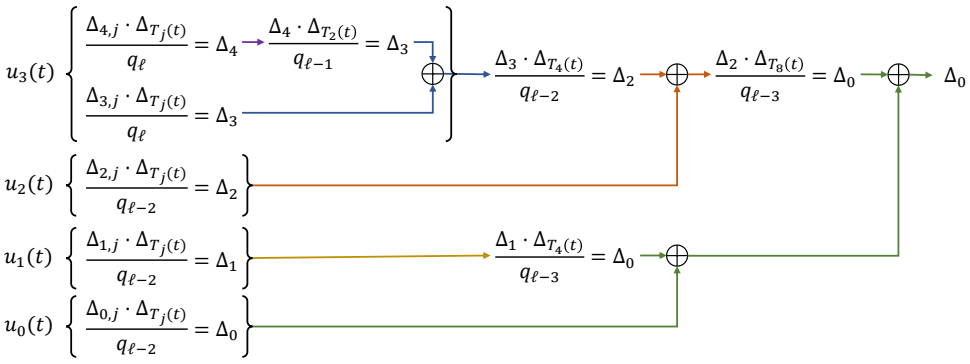


Fig. 1: Scale propagation for the evaluation of $p(t) = u_0(t) + u_1(t) \quad T_4(t) + (u_2(t) + u_3(t) \quad T_4(t)) \quad T_8(t)$. The evaluation of $u_3(t)$ is different, to ensure an optimal depth-consumption (see Section 3.2). The symbol denotes an addition between two (matching) scales.

3.2 Optimal Level Consumption

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

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

Due to the multiplications with $c_{max;j}$, the depth to evaluate $u_{max}(t)$ is $l+1$, which brings it one level below the first power-of-two basis T_{2^l} (i.e., one additional level is consumed). However, as long as the multiplication with the successive power-of-two basis T_{2^i} has at least one "gap" (one of the T_{2^i} is not used), this additional level can be "absorbed by the gap", and the resulting depth remains m. But this is not the case when $d > 2^m \quad 2^{l-1}$, because all the T_{2^i} are used, and the depth becomes $m+1$. We solve this discrepancy in Algorithm 2, by adding an additional recursive branching (lines 2-5).

Theorem 1. *Algorithm 2 has an optimal level-consumption of $m = \lceil \log(d+1) \rceil$ levels for any d.*

Proof. (sketch) We first observe that the number of levels needed to evaluate $p(t)$ using Algorithm 2 is equal to the number of levels needed to evaluate $u_{\lfloor d/2^l \rfloor}(t) \quad T_{2^{m-2^l}}(t)$. Indeed, the term $u_{\lfloor d/2^l \rfloor}(t)$ is the result of the decomposition of the highest degree terms of $p(t)$ and its degree is upper-bounded by 2^{l-1}. Also, the term $T_{2^{m-2^l}}(t)$ is the product of all the elements of the second power basis given as input. Then, we observe that Algorithm 2 decomposes $u_{\lfloor d/2^l \rfloor}(t) \quad T_{2^{m-2^l}}(t)$ using repeated squaring. The decomposition of the term $T_{2^{m-2^l}}(t)$ is trivial as it is given as input to the algorithm.

Algorithm 2: EvalRecurse

Input: A target scale , an upper-bound m, a stop factor l, a degree-d polynomial $p(t) = \{c_0, \ldots, c_d\}$, 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(ct).

```
if d < 2^l then
    if p(t) = u_max(t) and l > 2^m - 2^(l-1) and l > 1 then
        m ← ⌈log(d + 1)⌉
        l ← ⌊m/2⌋
        return EvalRecurse( ; p(t); m; l; T)
    else
        ct ← ⌊c_0 q_{T_d}⌉
        for i = d; i > 0; i = i - 1 do
            ct ← Add(ct; MultConst(T_i; ⌊(c_i q_{T_d}) = T_i⌉))
        end
        return Rescale(ct)
    end
end
m ← m - 1
Express p(t) as q(t) T_{2^m} + r(t)
ct_0 ← EvalRecurse((q_{T_{m-1}}) = T_m; q(t); m; l; T)
ct_1 ← EvalRecurse( ; r(t); m; l; T)
ct_0 ← Mul(ct_0; T_{2^m})
if level(ct_0) > level(ct_1) then
    ct_0 ← Add(Rescale(ct_0); ct_1)
else
    ct_0 ← Rescale(Add(ct_0; ct_1))
end
return ct_0
```

The decomposition of the term $u_{bd=lc}(t)$ is done by treating it as a new polynomial $p'(t)$ and recursively calling Algorithm 2 on it. At the end of the recursion, the highest-degree coefficients of the original $p(t)$ are fully decomposed terms of degree at most 1. The evaluation is then done using a pre-computed power-of-two basis and is equivalent to the evaluation of the monomial $a \; T_{2^m - 1}(t) = T_{2^m}(t)$ if $a = T_1(t)$ by using a square-and-multiply algorithm, which consumes m levels.

The additional recursions of Algorithm 2 add $d\log(d + 1 \quad (2^m \quad 2^{l-1}))e$ non-scalar multiplications, but enable the systematic evaluation of any polynomial by using exactly m levels, as illustrated in the toy example of Figure 1.

3.3 Conclusions

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

Our proposed algorithm also offers the possibility to choose the output scale and to guarantee that rescales and additions throughout the entire polynomial evaluation are exact, hence preventing the precision loss related to the scale deviation and making the

procedure easier to use. In Section 5, we show that these features are highly relevant for an efficient bootstrapping procedure. As linear transforms and constant multiplications can already be made scale-invariant, our scale-invariant polynomial evaluation is the remaining building block for enabling scale-invariant circuits of arbitrary depth.

4 Key-switch and Improved Matrix-Vector Product

The key-switch procedure is the generic public-key operation in the CKKS scheme. By generating specific public *switching-keys* derived from secret keys s' and s, it is possible to enable public re-encryption of ciphertexts from key s' to s. Beyond the public encryption-procedure (switching from $s' = 0$ to s), key-switch is required by most homomorphic operations to cancel the effect of encrypted arithmetic on the decryption circuit, thus ensuring the compactness of the scheme. More specifically, 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 associated with 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 impact on the overall efficiency of the scheme.

We propose an optimized switching-key format and an associated key-switch algorithm (Section 4.1). We then apply them to the specific case of rotation-keys and further improve the hoisted-rotation technique (Section 4.2), as introduced by Halevi et al. [17]. Finally, we propose a modified procedure for matrix-vector multiplications over packed ciphertexts (Section 4.3): it features a novel *double-hoisting* optimization. For each procedure, we provide mathematical notes (*Remarks* 1-4) to enable an efficient implementation.

4.1 Improved Key-switch Keys and Procedure

Given a ciphertext modulus $Q_L = \prod_{j=0}^{L} q_j$, we use a basis $\mathbf{w}$ composed of products among the q_j, as described by Han and Ki [18]. We also include the entire base $\mathbf{w}$ in the keys, as done by Bajard et al. and Halevi et al. [14, 3]; this saves one constant multiplication during the key-switch and enables a simpler key-switch keys generation. Therefore, we propose a simpler and more efficient hybrid between these previous approaches (Supplementary material 2 provides an overview of these approaches).

Specifically, our basis is $w^{(i)} = \frac{Q_L}{q_i}[(\frac{Q_L}{q_i})^{-1}]_{q_i}$ with $q_i = \prod_{j=}^{\min(\ (\ +1)\ -1;L)} q_j$ for $0 \quad i \quad , \quad = d(L+1)/e$ and $\quad$ a positive integer. Thus, the key-switch keys have the following format:

$$\mathsf{swk}^0_{q_i} = [\ -a_i s + s'\ P\ \frac{Q_L}{q_i}\ [(\frac{Q_L}{q_i})^{-1}]_{q_i} + e_i]_{Q_L P};$$
$$\mathsf{swk}^1_{q_i} = [a_i]_{Q_L P}.$$

We set $P = \prod_{j=0}^{-1} p_j$, and the bit-size of P such that $q_i \quad P; \forall i$. As shown by Han and Ki [18], this leads to a negligible error introduced by the key-switch operation. Algorithm 3 describes the associated key-switch procedure. As the decomposition basis is already included in the key, there is no need to multiply it to each decomposed components as a part of the final ciphertext re-composition, and there is no need to store additional constants.

Algorithm 3: key-switch

Input: $c \in R_{Q_\cdot}$, the switching key $\mathsf{swk}_{s \to s'}$ (both in the NTT domain).
Output: $(a', b') \in R^2_{Q_\cdot}$.

$d \leftarrow \mathsf{Decompose}(c)$ // (Algo. 10 in Supp. material 4)
$(a, b) \leftarrow \sum_i d^{(i)} \cdot \mathsf{swk}_{q_i}$ // $d^{(i)} = [c]_{q_i \cdot Q \cdot P}$
$a' \leftarrow \mathsf{ModDownNTT}(a)$ // $a' = \lfloor aP^{-1} \rceil_{Q_\cdot}$ (Algo. 11 in Supp. material 4)
$b' \leftarrow \mathsf{ModDownNTT}(b)$ // $b' = \lfloor bP^{-1} \rceil_{Q_\cdot}$
return (a', b')

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

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

4.2 Improved Hoisted-Rotations

The slot-rotation operation in CKKS is defined by the automorphism $\tau_k : X \to X^{5^k} \pmod{X^N + 1}$. Its effect is to rotate the message slots by k positions to the left[3]. Rotations are extensively used by circuits such as matrix multiplications. After a rotation, the secret under which the ciphertext is encrypted is changed from s to $\tau_k(s)$. Thus, a key-switch $\tau_k(s) \to s$ must be applied to go back to the original key.

Halevi et al. [17] show that as τ_k is an automorphism, it distributes over addition and multiplication, and commutes with the power-of-two base decomposition. As τ_k acts individually on the coefficients by permuting them without changing their norm (the modular reduction by $X^N + 1$ will at most induce a sign change), it also commutes with the special RNS decomposition (see Eq. (1) in Supplementary material 2): $[\tau_k(a)]_{q_i} = \tau_k([a]_{q_i})$. If we view a polynomial of R_{Q_L} as an $(L+1) \times N$ matrix, the effect of τ_k is to permute its columns; the coefficient-wise modular reduction (and RNS basis extension) independently acts on each column, and both operations commute.

Hence, when several rotations have to be applied on the same ciphertext, $[a]_{q_i}$ can be pre-computed and re-used for each subsequent rotation: $\sum_i \tau_k([a]_{q_i}) \cdot \mathsf{rot}_{k;q_i}$. Whereas the procedure proposed by Halevi et al. requires the computation of the automorphism for each of the a_{q_i}, this technique significantly reduces the number of NTTs and CRT reconstructions, thus reducing the overall complexity (see Supplementary material 3.1).

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

Algorithm 4: Optimized Hoisting-Rotations

Input: $\mathsf{ct} = (c_0, c_1) \in R_Q^2$ and a set of r rotation keys rot_{r_k} (both in the NTT domain).

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

$\mathbf{d} \leftarrow \mathsf{Decompose}(c_1)$ // (Algo. 10 in Supp. material 4)

foreach r_k **do** // in parallel

  $(a, b) \leftarrow \sum_i d^{(i)} \cdot \mathsf{rot}_{r_k;q_i}$ // $d^{(i)} = [c]_{q_i \cdot Q \cdot P}$

  $a' \leftarrow \mathsf{ModDownNTT}(a)$ // $a' = aP^{-1} \cdot ae$ (Algo. 11 in Supp. material 4)

  $b' \leftarrow \mathsf{ModDownNTT}(b)$ // $b' = bP^{-1} \cdot be$

  $\mathbf{v}_{r_k} \leftarrow \{\tau_{r_k}(c_0 + a'), \tau_{r_k}(b')\}$

end

return $\mathbf{v}$

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

$$\mathsf{rot}^0_{k;q_i} = [\tau_k^{-1}(s) + s \cdot P \cdot \tfrac{Q_L}{q_i} \cdot [(\tfrac{Q_L}{q_i})^{-1}]_{q_i} + e_i]_{Q_L P};$$

$$\mathsf{rot}^1_{k;q_i} = [a_i]_{Q_L P}.$$

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

$$\sum_i \tau_k([a]_{q_i}) \cdot \mathsf{rot}_{k;q_i} = \tau_k \Big(\sum_i [a]_{q_i} \cdot \mathsf{rot}_{k;q_i}\Big).$$

Our improved algorithm for hoisted rotations is detailed in Algorithm 4. We minimize the number of evaluations of τ in Algorithm 4 thus make it constant-time, regardless of the basis decomposition.

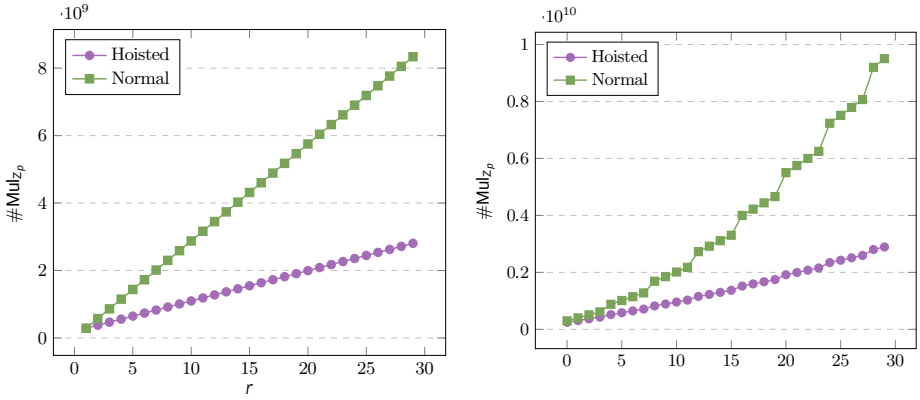
Let r be the number of rotations, N the ring degree, ℓ the current level and $\alpha = (\ell+1)/\beta$, with β a positive integer. The total complexity of this algorithm in number of integer multiplications (derived in Supplementary material 3.2), that we denote $\#\mathsf{Mul}_{z_p}$, is

$$\begin{aligned}\#\mathsf{Mul}_{z_p} = {} & N \log(N) (\ell+1)(\alpha+1+2r) + 2r \\ & + N(\ell+1) + 2r(\alpha+\beta+1) \\ & + N + 2r(\alpha+1).\end{aligned}$$

Figure 2 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.

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



(a) Varying number of rotations r. Parameters: fN=2^{16}; `=21; =4; =d(`+1)= eg.

(b) Varying input level for r=20 rotations. Parameters: fN=2^{16}; =4; =d(`+1)= eg.

Fig. 2

generic linear transformations to encrypted vectors is pivotal for most applications. In particular, the homomorphic computation of the CKKS encoding and decoding procedures, which carry a prohibitive cost in the original bootstrapping procedure, are linear transformations.

Halevi and Shoup propose to express an n n matrix M in diagonal form and to use a baby-step giant-step (BSGS) algorithm to evaluate the matrix product in $O(\sqrt{n})$ rotations [15, 16]. At the time of this writing, all the existing bootstrapping procedures are based on this approach. We now break down the cost of the BSGS algorithm, analyze its components and, using our observations, we present our improvements to this approach.

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

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

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

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

Algorithm 5: BSGS Algorithm For Matrix Vector Multiplication

Input: ct a ciphertext encrypting $\mathbf{m}$ 2 $\mathbb{C}^n$, $\mathbf{M}_{diag}$ the diagonal rows of $\mathbf{M}$ a n n matrix with $n = n_1 n_2$.

Output: The evaluation $\mathsf{ct}' = \mathsf{M}$ ct.

1 **foreach** $i = 0; i < n_1; i = i + 1$ **do**
2 | ct_i Rotate$_i$(ct)
3 **end**
4 ct' $(0; 0)$
5 **foreach** $j = 0; j < n_2; j = j + 1$ **do**
6 | $\mathbf{r}$ $(0; 0)$
7 | **foreach** $i = 0; i < n_1; i = i + 1$ **do**
8 | | $\mathbf{r}$ Add($\mathbf{r}$; Mul(ct_i; Rotate$_{n_1 j}(M_{diag}^{(n_1 j+i)})$))
9 | **end**
10 | ct' Add(ct'; Rotate$_{n_1 j}(\mathbf{r})$)
11 **end**
12 ct' Rescale(ct')
13 **return** ct'

to which we must also add $2n_2n_1$ multiplications in R_Q (line 8 of Algorithm 5). We denote *inner loop* and *outer loop* the loops that depend, respectively, on the value n_1 and n_2. Figure 3 shows the weight of each of the four steps within the total complexity of ` rotations (see Supplementary material 3.2 for the underlying complexity analysis). The complexity of the steps *MultSum* and *Permute* is close to 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 greatly reduces the complexity related to the inner-loop rotations and consists in two hoisting levels. The *first level* applies to the inner-loop rotations (lines 7 to 10 of Algorithm 5), as proposed by Halevi et al. [17]. This renders the computation dedicated to *Decompose* independent of the value n_1, so the complexity is reduced to

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

The *second level* introduces an additional hoisting for the inner-loop rotations, by observing that the *ModDown* step is a coefficient-wise operation. Thus, this operation, similarly to the *Decompose* step, commutes with the *Permute* step and the ciphertext-plaintext multiplications (line 8 of Algorithm 5). Therefore, we apply it only once, after the entire inner loop of n_1 rotations, thus reducing the number of key-switch operations from $n_1 + n_2$ to $n_2 + 1$. Applying the same reasoning for the *ModDown* step of the outer-loop rotations, the complexity is reduced to

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

Our double-hoisting BSGS matrix vector is described in Algorithm 6.

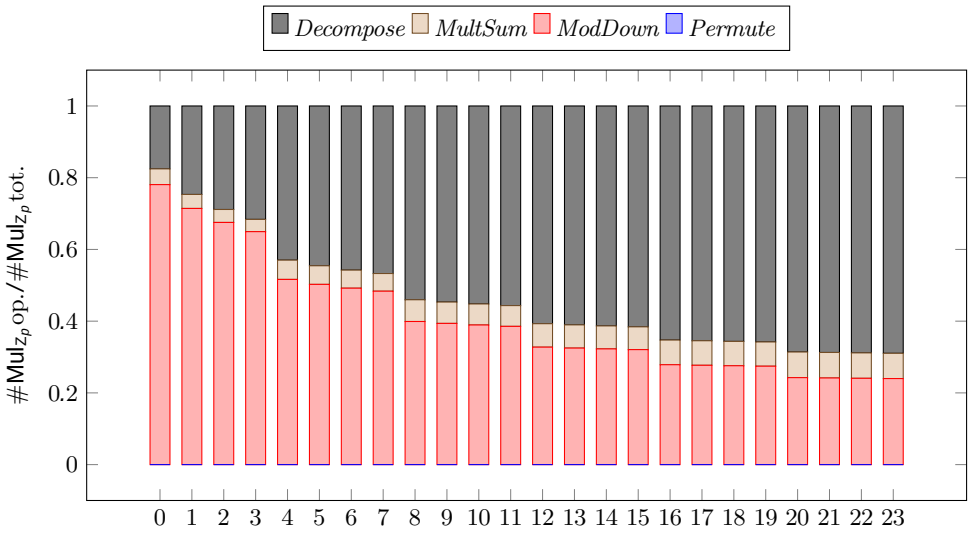


Fig. 3: Normalized complexity of each step (op.) of a rotation. The complexity for each operation was computed with $N = 2^{16}$, 0 ` 23 and = 4

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

Remark 4. Algorithm 6 is presented with M given in plaintext, but the adaptation to an encrypted M is trivial and would not significantly affect the complexity (as long as M does not need to be manipulated or modified). Indeed, hoisting can also be applied to the relinearization of the ciphertext-ciphertext multiplications by summing the results in $R^3_{Q\cdot P}$ in the *inner loop* and by applying the relinearization $R^3_{Q\cdot}$! $R^2_{Q\cdot}$ step only once per *outer loop*.

Discussion In addition to benefiting from our improved key-switch (Section 4.1) and rotation (Section 4.2) procedures, Algorithm 6 introduces a trade-off: The *ModDown* step in the *inner loop* is now dependent on the value n_2, and the *ModDown* step of the *outer loop* has to be performed only once. However, the $2n_1n_2$ multiplications and additions have to be performed in $R_{Q\cdot P}$ instead of $R_{Q\cdot}$. Hence, the complexity dependency on the value n_1 is significantly reduced at the cost of slightly increasing the complexity dependency on n_1n_2. Applying the *ModDown* step at the end of each loop has the additional positive effect of introducing the rounding error only once.

Table 2 compares the complexity of a non-hoisted (for reference), single-hoisted (Algorithm 5) and double-hoisted (Algorithm 6) BSGS, with the optimal ratio $n_1{=}n_2$ for each of the approaches. Our approach minimizes the complexity when 2^3 $n_1{=}n_2$ 2^4. This shows that the strategy of the previously proposed bootstrapping procedures [9, 5, 18], which targets n_1 n_2 to minimize the number of rotations, is not optimal anymore. The maximum gain occurs when n (the number of non zero diagonals) is

Algorithm 6: Double-hoisting BSGS matrix vector algorithm

Input: $\mathsf{ct} = (c_0, c_1) \in R_Q^2$, a ciphertext, $\mathbf{M}_i^{diag} \in R_{Q \cdot P}$ the set of pre-rotated and encoded diagonals of $\mathbf{M}$ a plaintext $n \times n$ matrix, $n_1 n_2 = n$, $\mathsf{rot}_i \in R_{Q \cdot P}^2$ the set of necessary rotations keys.

Output: The evaluation of $\mathbf{M} \times \mathsf{ct}$.

1 $\mathbf{d} \leftarrow \mathsf{Decompose}(c_1)$ // $Q \cdot \to Q \cdot P$ (Algo. 10 in Supp. material 4)
2 **foreach** k_{n_1} **do**
3 $\quad a_{k_{n_1}} \leftarrow \kappa_{n_1}(P \times c_0 + \sum d^{(i)} \times \mathsf{rot}^{(0)}_{k_{n_1}; q_i})$ // $\in R_{Q \cdot P}$
4 $\quad b_{k_{n_1}} \leftarrow \kappa_{n_1}(\sum d^{(i)} \times \mathsf{rot}^{(1)}_{k_{n_1}; q_i})$ // $\in R_{Q \cdot P}$
5 **end**
6 $r_0, r_1, r_2 \leftarrow (0), (0), (0)$
7 **foreach** k_{n_2} **do**
8 $\quad u_0, u_1 \leftarrow (0), (0)$
9 $\quad$ **foreach** k_{n_1} **do**
10 $\quad\quad u_0 \leftarrow u_0 + a_{k_{n_1}} \times \mathbf{M}^{diag}_{k_{n_1}}$ // $\in R_{Q \cdot P}$
11 $\quad\quad u_1 \leftarrow u_1 + b_{k_{n_1}} \times \mathbf{M}^{diag}_{k_{n_1}}$ // $\in R_{Q \cdot P}$
12 $\quad$ **end**
13 $u_0^0 \leftarrow \mathsf{ModDownNTT}(u_0)$ // $u_0^0 = \lfloor P^{-1} \times u_0 \rceil \in R_{Q \cdot}$
14 $u_1^0 \leftarrow \mathsf{ModDownNTT}(u_1)$ // $u_1^0 = \lfloor P^{-1} \times u_1 \rceil \in R_{Q \cdot}$
15 $\quad \mathbf{d} \leftarrow \mathsf{Decompose}(u_1^0)$ // $Q \cdot \to Q \cdot P$
16 $\quad r_0 \leftarrow r_0 + \kappa_{n_2}(\sum d^{(i)} \times \mathsf{rot}^{(0)}_{k_{n_2}; q_i})$ // $\in R_{Q \cdot P}$
17 $\quad r_1 \leftarrow r_1 + \kappa_{n_2}(\sum d^{(i)} \times \mathsf{rot}^{(1)}_{k_{n_2}; q_i})$ // $\in R_{Q \cdot P}$
18 $\quad r_2 \leftarrow r_2 + \kappa_{n_2}(u_0^0)$ // $\in R_{Q \cdot}$
19 **end**
20 $r_0^0 \leftarrow \mathsf{ModDownNTT}(r_0)$ // $r_0^0 = \lfloor P^{-1} \times r_0 \rceil \in R_{Q \cdot}$
21 $r_1^0 \leftarrow \mathsf{ModDownNTT}(r_1)$ // $r_1^0 = \lfloor P^{-1} \times r_1 \rceil \in R_{Q \cdot}$
22 **return** $(r_0^0 + r_2, r_1^0)$

around 128 and then falls again for smaller values. This behavior can be exploited by factorizing the linear transforms, which are used during the bootstrapping, into several sparse matrices (see Section 5.3).

Note that whereas Algorithm 6 reduces the overall complexity of matrix vector products, it also induces an increase in the number of used rotation keys. Therefore, it introduces a time-memory trade-off. For example, increasing the ratio from $n_2/n_1 = 1$ to $n_2/n_1 = 16$ in our bootstrapping parameters (Section 6) increases the number of keys by a factor around 1.6 and reduces the computation time by 20%.

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] (Section 5.1) and adapt each step by relying on the techniques proposed in Sections 3 and 4.

The purpose of the CKKS bootstrapping [9, 5, 18], in contrast with the BFV [12] and BGV [4] bootstrapping, is not to reduce the error. Instead, it is meant to reset the ciphertext modulus to a higher level in order to enable further homomorphic

Table 2: Complexity comparison between the original Algorithm 5, Algorithm 5 with single hoisted rotations and Algorithm 5 with double hoisted rotations (Algorithm 6). M is a 2^{15} 2^{15} matrix with $n = n_1 n_2$ non zero diagonals. The used parameters are $N = 2^{16}$, 2^{15} slots, $\grave{} = 18$ and $= 4$. The speed-up factor is the ratio between the $\#\mathsf{Mul}_{\mathsf{Z}_p}$ of the original algorithm (1-hoisting was not used in the bootstrapping procedures of [9, 18, 5]) and our double hoisted algorithm.

	Original		1-hoisted [17]		2-hoisted [Ours]		Speed-
n	n_1=n_2	$\log(\#\mathsf{Mul}_{\mathsf{Z}_p})$	n_1=n_2	$\log(\#\mathsf{Mul}_{\mathsf{Z}_p})$	n_1=n_2	$\log(\#\mathsf{Mul}_{\mathsf{Z}_p})$	up
32768	2	37.276	2	36.913	8	36.813	1.378
16384	1	36.500	4	36.114	16	35.903	1.512
8192	2	35.865	2	35.364	8	35.055	1.753
4096	1	35.152	4	34.648	16	34.205	1.927
2048	2	34.597	2	33.981	8	33.446	2.219
1024	1	33.927	4	33.337	16	32.672	2.386
512	2	33.422	2	32.732	8	32.014	2.653
256	1	32.769	4	32.137	16	31.318	2.733
128	2	32.282	2	31.568	8	30.753	2.886
64	1	31.614	4	30.992	16	30.127	2.804
32	2	31.112	2	30.430	8	29.637	2.779
16	1	30.375	4	29.842	16	29.311	2.090
8	2	29.792	2	29.248	2	29.116	1.596

multiplications. The approximate nature of the CKKS scheme, due to the plaintext and ciphertext error being mixed together, implies that each homomorphic operation decreases the output precision. As a result, all the currently proposed bootstrapping circuits only approximate the ideal bootstrapping operation, and their output precision determines their practical utility.

5.1 Circuit Overview

Let $f\mathsf{ct} = (c_0; c_1); Q_0;\ \ g$ be a ciphertext that encrypts an n-slot message under a secret-key s with hamming weight h, such that $\mathsf{Decrypt}(\mathsf{ct}; s) = c_0 + sc_1 = b\ \ m(Y)e + e\ 2\ \mathbb{Z}[Y]=(Y^{2n}+1)$, where $Y = X^{N=2n}$. The bootstrapping operation outputs a ciphertext $f\mathsf{ct}^0 = (c_0^0; c_1^0); Q_L\ \ _k;\ \ g$ such that $c_0^0 + sc_1^0 = b\ \ m(Y)e + e^0\ 2\ \mathbb{Z}[Y]=(Y^{2n}+1)$, where $k < L$ is the number of levels consumed by the bootstrapping and $jje^0jj\ \ jjejj$ is the error that results from the combination of the initial error e, homomorphic operations, rounding during the rescale and encoding, and function approximations.

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

1. ModRaise: ct is raised to the modulus Q_L by applying the CRT map $R_{q_0}\ \ !\ \ R_{q_0}$ R_{q_1} R_{q_L}. This yields a ciphertext $f\mathsf{ct}; Q_L;\ \ g$ for which

$$[c_0 + sc_1]_{Q_L} = Q_0\ \ I(X) + b\ \ m(Y)e = m^0;$$

where $Q_0\ \ I(X) =\ \ [sc_1]_{Q_0} + sc_1\ _{Q_L}$ is an integer polynomial for which $jjI(X)jj$ is $O(\sqrt{p}\overline{h})$ [9]. The next four steps remove this unwanted $Q_0\ \ I(X)$ 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 $Q_0 \; I(X) + b \quad m(Y)e$ to $(N/2n) \; (Q_0 \; \tilde{I}(Y) + b \quad m(Y)e)$, a polynomial in Y [9].
3. CoeffToSlots: The message $m' = Q_0 \; \tilde{I}(Y) + b \quad m(Y)e$ is in the *coefficient* domain, i.e. $m' = \quad b(Q_0/ \;) \; \tilde{I}(Y) + m(Y)e$, 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 $4n \; N$, these ciphertexts can be repacked into one. Otherwise, the next step is applied separately on both ciphertexts.
4. EvalSine: The modular reduction $f(x) = x \bmod 1$ is homomorphically evaluated on the ciphertext(s) encrypting $\mathsf{Encode}(m')$. This function is approximated by $\frac{Q_0}{2} \quad \sin \quad \frac{2 \quad x}{Q_0}$, which is tight when $Q_0/ \quad \; jjm(Y)jj$. As the range of x is determined by $jj\tilde{I}(Y)jj$, the approximation needs to account for the secret-key density.
5. SlotsToCoeffs: This step homomorphically evaluates the DFT on the ciphertext(s) encrypting $f(\mathsf{Encode}(m'))$. It returns a ciphertext at level $Q_L \; _k$ that encrypts $\mathsf{Decode}(f(\mathsf{Encode}(m'))) \quad f(m') \quad b \quad m(Y)e$, which is a close approximation of the original message.

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

5.2 ModRaise and SubSum

We base the ModRaise and SubSum operations directly on the initial bootstrapping of Cheon et al. [9] and provide their respective implementations in Algorithm 12 and Algorithm 13 in Supplementary material 4. The SubSum step multiplies the encrypted message by a factor $N/2n$ that needs to be subsequently cancelled. We take advantage of the following CoeffToSlot step, that is 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 CoeffToSlots and SlotsToCoeffs

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

Overview Let n be a power-of-two integer such that $1 \quad n < N$; the following holds for any two vectors $\mathbf{m}, \mathbf{m}' \; 2 \; \mathbb{C}^n$ due to the convolution property of the DFT

$$\mathsf{Decode}_n(\mathsf{Encode}_n(\mathbf{m}) \quad \mathsf{Encode}_n(\mathbf{m}')) \quad \mathbf{m} \quad \mathbf{m}',$$

where and respectively denote the nega-cyclic convolution and Hadamard multiplication. In other words, the encoding and decoding algorithms define an isomorphism between $\mathbb{R}[Y]/(Y^{2n} + 1)$ and $\mathbb{C}^n$ [10]. The goal of the CoeffToSlots and SlotsToCoeffs steps is to homomorphically evaluate this isomorphism on a ciphertext.

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

$$\mathrm{SF}_n = \begin{matrix} 2 \\ 1 \\ 6 1 \\ 6 1 \\ 6 1 \\ 6 \\ 6 : \\ 6 : \\ 6 1 \\ 6 1 \\ 6 1 \\ 4 1 \\ 1 \end{matrix} \begin{matrix} 3 \\ 7 \\ 7 \\ 7 \\ 7 \\ 7 \\ 7 \\ 7 \\ 7 \\ 7 \\ 7 \\ 5 \end{matrix} ;$$

$$\begin{matrix} & : : : & n \ 2 & n \ 1 \\ 5 & : : : & (n \ 2) \ 5 & (n \ 1) \ 5 \\ 5^2 & : : : & (n \ 2) \ 5^2 & (n \ 1) \ 5^2 \\ & & & \\ : & \ddots & : & : \\ 5^{n \ 3} & : : : & (n \ 2) \ 5^{n \ 3} & (n \ 1) \ 5^{n \ 3} \\ 5^{n \ 2} & : : : & (n \ 2) \ 5^{n \ 2} & (n \ 1) \ 5^{n \ 2} \\ 5^{n \ 1} & : : : & (n \ 2) \ 5^{n \ 1} & (n \ 1) \ 5^{n \ 1} \end{matrix}$$

and its inverse $\mathrm{SF}_n{}^1 = \frac{1}{n}\overline{\mathrm{SF}}_n^T$ [6], and their homomorphic evaluation can be expressed in terms of plaintext matrix vector products:

1. $\mathsf{Coe\ sToSlots}(\mathbf{m}) : t_0 = \frac{1}{2} \ \mathrm{SF}_n{}^1 \quad \mathbf{m}+\overline{\mathrm{SF}_n{}^1 \quad \mathbf{m}} \ ; t_1 = \quad \frac{1}{2} i(\mathrm{SF}_n{}^1 \quad \mathbf{m} \quad \overline{\mathrm{SF}_n{}^1 \quad \mathbf{m}}$
2. $\mathsf{SlotsToCoe\ s}(t_0; t_1) : \mathbf{m} = \mathrm{SF}_n \quad (t_0 + i \quad t_1)$.

DFT Evaluation In their initial bootstrapping proposal, Cheon et al. in [9] homomorphically compute the DFT as a matrix-vector product, using a baby-step giant-step (BSGS) approach: Given SF_n represented in diagonal form and two integers n_1 and n_2 such that $n = n_1 n_2$, Algorithm 5 evaluates $\mathrm{SF}_n \quad \mathbf{m}$ in $n_1 + n_2 \quad 2\sqrt{n}$ rotations and n plaintext multiplications and additions. This algorithm consumes only one level and is much more efficient than the naïve one, which would require n rotations. However, it still remains prohibitive for large n, as it involves a large number of rotations.

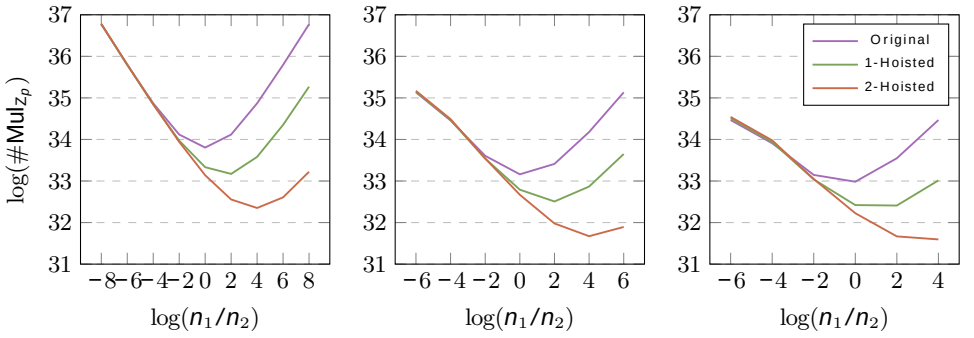
To reduce the complexity of Algorithm 5, two recent works from Cheon et al. [6] and Chen et al. [5] exploit the structure of the equivalent FFT algorithm by recursively merging its iterations. In matrix form, both approaches find appropriate factorizations of SF_n (and its inverse)

$$\mathrm{SF}_n = \underbrace{\mathrm{M}_0 \quad \mathrm{M} \quad {}^1}_{\text{partially factorized}} \ [6, 5] \ \underbrace{\mathrm{M}_0 \quad \mathrm{M}_{\log(n)} \quad {}^1}_{\text{fully factorized}}$$

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

We use our variant of the BSGS matrix-vector multiplication approach (see Section 4.3 and Algorithm 6) and combine it with level-merging [6, 5]. We pre-compute the full factorization of SF_n (and $\mathrm{SF}_n{}^1$) into sparse matrices. From the full factorization into $\log(n)$ matrices, we first split the matrices into groups of similar size then apply the FFT iteration-merging by multiplying among those groups. This ensures that the total complexity is evenly distributed across all the final matrices.

Figure 4 shows the effect of our algorithm on the $\mathsf{Coe\ sToSlots}$ step, compared with the original BSGS algorithm, for different factorizations corresponding to $_{\mathrm{SF}_n{}^1} =$



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

Fig. 4: Theoretical complexity of CoeffToSlots for different DFT factorizations using Algorithm 5 with no hoisting (original), single hoisting, and double hoisting (Algorithm 6).

$f2; 3; 4g$. The complexity is computed as the number of products in $\mathbb{Z}_p$, with parameters $N = 2^{16}$, a target $` = 17$ (the level after CoeffToSlots) and $n = 2^{15}$ slots.

It can be seen that each level of hoisting reduces the total complexity by a noticeable amount. The first level of hoisted rotations, as proposed in Section 4.2, changes the parameters $n_1; n_2$ for which the minimum complexity is achieved to $n_1 \quad 2^2 n_2$, instead of $n_1 \quad n_2$. Using a second level of hoisted rotations further shifts the minimum complexity to $n_1 \quad 2^4 n_2$. On average, our method reduces the complexity of the linear transformations in the bootstrapping by a factor of 2:8 . We will show that, for our best performing parameters, we are able to reduce the run-time of the CoeffToSlots and SlotsToCoeffs down to equal or lower that of the EvalSine (see Section 7).

Efficient Repacking of Sparse Plaintexts. The first part of the CoeffToSlots step is a DFT that outputs a vector of $\mathbb{C}^n$ values; the second part of the CoeffToSlots step applies the map $\mathbb{C}^n \ ! \ \mathbb{R}^{2n}$ to this vector. The real values of $\mathbb{C}^n$ are put in the first n slots of the vector in $\mathbb{R}^{2n}$, whereas the imaginary values of $\mathbb{C}^n$ are multiplied by i and located in the last n slots. During the decoding, the inverse mapping $\mathbb{R}^{2n} \ ! \ \mathbb{C}^n$ is used. We have to apply these operations in the encrypted domain before and after the EvalSine.

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

We observe that decoding a plaintext $\mathbf{m} \ 2 \ \mathbb{C}^n$ 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 $\mathbf{m}$:

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

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

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

$$\begin{matrix} I_n & i \cdot I_n \\ I_n & -i \cdot I_n \end{matrix};$$

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

5.4 EvalSine

EvalSine implements the homomorphic modular reduction of the message $\lfloor q_0/q_m \rceil m(Y) + \tilde{I}(Y) + m(Y)$ modulo q_0. The modular reduction is approximated by the function

$$f(x) = \frac{q_0}{q_m}\frac{1}{2\pi}\sin\left(2\pi x \frac{q_m}{q_0}\right) \approx \frac{q_0}{q_m} \frac{q_m}{q_0} x \mod 1;$$

which scales the message down to $\lfloor q_m/q_0 \rceil m(Y) + \tilde{I}(Y) + (q_m/q_0)\, m(Y)$, removes the $\tilde{I}(Y)$ polynomial by taking the message modulo 1, and scales the message up to $\approx \lfloor q_m/q_0 \rceil m(Y)$. Because $\tilde{I}(Y)$ mostly determines the range and degree of the approximation, the EvalSine step has to take the secret-key density h into account. More specifically, the range of approximation $(-K; K)$ is chosen as $K = \lceil c\sqrt{h} \rceil$ such that $\Pr[||\tilde{I}(Y)|| > K] < 2^{-\lambda}$. We elaborate more on how we choose K with respect to h and λ in Section 6.2.

Previous Work Chen et al. [5] directly approximate the function $\frac{1}{2\pi}\sin(2\pi x)$ by using a standard Chebyshev interpolant of degree $d = 119$ in an interval of $(-K; K)$ for $K = 12$ (using a sparse key with $h = 64$). The multiplications by q_0/q_m and q_m/q_0 are performed, respectively, before and after the evaluation of the Chebyshev interpolant. Han and Ki [18] choose to approximate $\cos(2\pi\frac{1}{2^r}(x - 0.25))$ followed by r iterations of the double angle formula $\cos(2x) = 2\cos(x)^2 - 1$ to approximate $\sin(2\pi x)$,

then they multiply the result by 1=2 . 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, thus further reducing the degree of the approximation. In their work, they use an interpolant of degree 30 with a scaling factor $r = 2$ (they also use a sparse key with $h = 64$). This results in a much smaller Chebyshev-interpolant degree that reduces the total amount of steps to evaluate the polynomial and that leads to a faster evaluation. However, their interpolant has a minimum degree of $2K$ 1 and is not always numerically stable, especially if $r = 0$.

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

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

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

Our technique includes the 1=2 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.

The ciphertext must be divided by $Q_0/\ m$ before the polynomial evaluation and multiplied by $Q_0/\ m$ after. However, Q_0 is an NTT-friendly prime and the value $Q_0/\ m$ is not an integer (we assume that $\ m$ is a power of two). Hence, to maximize the resulting precision after the bootstrapping, extra care must be taken during these scalings. We solve this problem in three steps. We first pre-multiply the ciphertext by a correcting factor $Q_0/2^{b\log(Q_0)e}$ during the Coe sToSlots step, then we perform the multiplication and division by $bQ_0/\ me$ by manipulating the scale during the EvalSine step and, finally, we correct back the ciphertext with $2^{b\log(Q_0)e}/Q_0$ during the SlotsToCoe s step (see Section 5.5 for further details). With this approach the division is exact, and the shape of Q_0 can therefore be chosen to be small and with relatively loose bounds, as long as the ratio between Q_0 and the message permits a good approximation of the modular reduction by the sine function.

We also observed that, when considering dense keys, the specialized interpolation method of Han and Ki leads to an interpolant degree higher than the standard Chebyshev interpolation. When K is large, the minimum degree of Han and Ki's interpolant for a given precision grows slower than the minimum degree of $2K$ 1 imposed by their method, which is further amplified when using a scaling factor r. Hence, we use the standard Chebyshev interpolation technique when d $2K$ 1 and, otherwise, the modified technique of Han and Ki.

Algorithm 7 details our implementation of the EvalSine procedure. Due to the change of variable and the division by 2^r, the input ciphertext is pre-multiplied by

Algorithm 7: EvalSine

Input: $f\mathsf{ct}; Q`; \quad g$ a ciphertext, $p(t)$ a Chebyshev interpolant of degree d of $f(x) = x \mod 1$, K the range of interpolation, r a scaling factor.

Output: The scale invariant evaluation $\mathsf{ct}^0 = bQ_0 = \ _m e \ \ p(bQ_0 = \ _m e^{\ 1} \ \ \mathsf{ct})$.

```
1  bQ_0= _m e // Division by bQ_0= _m e
2  T_0; T_1  1; ct
3  T_1  AddConst(T_1;  0:5=(2^{r+1} K))
4  m  dlog(d + 1)e
5  l  bm=2c
6  T  fT_0; T_1; : : : ; T_{2^l}; : : : ; T_{2^m _1} g // Compute the power basis
7  for i = 0; i < r; i = i + 1 do
8  |  p q_L  CtS depth  EvalSine depth  r+i // Compute the appropriate
9  end
10 ct^0  EvalRecurse( ; m; l; p(t); T) (Algorithm 2) // Scale invariant
11    (1=2 )^{1=2^r}
12 for i = 0; i < r; i = i + 1 do
13 |  ^2
14 |  ct^0  Mul(ct^0; ct^0)
15 |  ct^0  Add(ct^0; ct^0)
16 |  ct^0  AddConst(ct^0;  )
17 |  ct^0  Rescale(ct^0) //  ^2=q_L  CtS depth  EvalSine Depth  i
18 end
19  bQ_0= _m e^{ 1} // Multiplication by bQ_0= _m e
20 return ct^0
```

$1=(2^r K)$ during the CoeffsToSlots step (see Section 5.5). The of the ciphertext after polynomial evaluation (EvalRecurse) takes into account the subsequent evaluations of the double-angle formula such that it remains unchanged after the EvalSine step.

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 $\mathrm{SF}_n{}^1$ 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 $2=(2^r(b \ \ a)) = 1=(2^r K)$ for the scaling by $1=2^r$ and change of variable for the polynomial evaluation in Chebyshev basis, and (iii) by $Q_0=2^{d\log(Q_0)e}$ to compensate for the error introduced by the approximate division by $bQ_0 = \ _m e$. Therefore, the matrices resulting from the factorization of $\mathrm{SF}_n{}^1$ are scaled by

$$\mathsf{CtS} = \left(\frac{1}{2^r K N} \ \ \frac{Q_0}{2^{b\log(Q_0)e}}\right)^{\frac{1}{\mathrm{SF}_n{}^1}}$$

where $_{\mathrm{SF}_n{}^1}$ is the degree of factorization of $\mathrm{SF}_n{}^1$. 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 $2^{d\log(q_0)e}=q_0$ to compensate for the error introduced by the approximate multiplication by $bQ_0 = \ _m e$, and (ii) by

= , where is the scale of the ciphertext after the EvalSine step and is the desired ciphertext output scale.

Therefore, the matrices resulting from the factorization of SF_n are scaled by

$$\mathrm{StC} = \left(\frac{2^{b\log(Q_0)e}}{Q_0} \right)^{\frac{1}{\mathrm{SF}_n}},$$

where $_{\mathrm{SF}_n}$ 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 through experimental processes for finding appropriate trade-offs between performance and the probability of decryption errors. In Sections 6.1–6.2, we discuss various constraints and inter-dependencies in the parameter selection. Then, we propose a generic procedure for finding appropriate parameter sets in Section 6.3.

6.1 Security

We base our parameters selection on the works of Curtis and Player [11], Cheon et al. [8], and Son et al. [25] for both the sparse- ($h = 128$ and $h = 196$) and the dense-keys ($h = 32768$) setting. For each case, we select a modulus size that ensures a security of 128 bits. All three works suggest slightly different parameters. Extrapolating the work of Cheon et al. [8], we deduce that the parameters ($N = 65536; \log(Q) = 1010; h = 64$) would achieve a security of about 128 bits. In a more recent work, Son et al. [25] report that parameters ($65536; 1250; 64$) provide a security estimated at 113 bits. Both works focus on showing that the parameters that are currently commonly used for the bootstrapping do not meet the security standards, but they do not propose, or show how to select, updated parameters. Conversely, the work of Curtis and Player takes a more general approach and proposes a systematic way to extrapolate the security for large rings by taking the key density into account. We base the choice of the maximum modulus size for 128-bit security on their work. These values are shown in Table 3 for several choices of h. These are slightly more conservative for $h = 64$ than those of Cheon et al. [8] and Son et al. [25].

6.2 Choosing K for EvalSine

The previous works on bootstrapping [9, 5, 6, 18] use a sparse key with $h = 64$ and $K = 12$. This value $K = 12$ was experimentally determined by Cheon et al. by using the heuristic assumption that $jj\tilde{I}(Y)jj \quad O(\sqrt{h})$ [9]. In practice, this value works well as the coefficients of $\tilde{I}(Y)$ rarely ever go above 10. However, this is no longer true for larger values of h, for which new heuristic bounds must be found. For this purpose, we conducted the following experiment: For each $h \in [64; 96; 128; 192; 16384]$ we sampled 10^5 encryptions of zero in $Z_{Q_0}[X]/(X^{2^{16}}+1)$, each was encrypted with a freshly sampled secret key; we then decrypted the result in $Q_L > h \cdot Q_0^2$ and recorded the distribution of $\tilde{I}(Y)$.

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

h	$\log(QP)$				K		
	$\log(QP; N);$	$= 128$	$N = 2^{15}$	$N = 2^{16}$	K	$\Pr[\lvert x_i \rvert \quad K]$	$K/\sqrt{h}$
64	0:015121N	8:248756	496	982	12	$2^{-38:8}$	1.500
96	0:018896N	3:671642	619	1234	15	$2^{-40:5}$	1.531
128	0:021370N	3:601990	699	1396	17	$2^{-39:4}$	1.502
192	0:023448N	3:611940	767	1533	21	$2^{-41:3}$	1.515
N=2	[11]		881	1782	257	$2^{-40:1}$	1.42

Table 4: The sets of parameters of the full-RNS variant of CKKS used to evaluate the performance of our bootstrapping implementation. + denotes the concatenation in the chain and $a \quad b$ denotes the consecutive concatenation of a distinct moduli of size b. A fractional a indicates that the modulus is only partially used by the step it is allocated to.

Parameters																
Set	**CKKS**										**Bootstrapping**					
	h	N		$\log(QP)$	L	$\log(q_i)$				$\log(p_j)$	StC & StC		Sine		
						$q_0 \quad i \quad (L \quad k)$	StC	Sine	CtS		$\mathrm{SF}_n{}^{-1}$	SF_n	K	d	r
I	192	2^{16}	2^{45}	1521	24	$55 + 10 \quad 45$	$56 + 28$	$8 \quad 55$	$4 \quad 53$	$5 \quad 56$	4	3	21	52	2
II	192		2^{30}	1553	21	$55 + 7{:}5 \quad 60$	$1{:}5 \quad 60$	$8 \quad 55$	$4 \quad 53$	$5 \quad 61$	4	3	21	52	2
III	32768		2^{45}	1782	25	$55 + 9 \quad 45$	$56 + 28$	$11 \quad 60$	$4 \quad 53$	$6 \quad 61$	4	3	257	250	3
IV	192	2^{15}	2^{25}	768	14	$35 + 50 + 25$	60	$8 \quad 50$	$2 \quad 49$	$2 \quad 50$	2	2	21	52	2

Using a quadratic polynomial approximation, we extrapolated the probability distribution obtained by this experiment in Table 3. Our extrapolation shows that, for $h = 64$ and $K = 12$, the overflow probability per plaintext coefficient is approximately $2^{-38:8}$. Using the same technique as for $(h = 64; K = 12)$, we validated other values $(h; K)$ that give a similar failure probability. We observe that taking $K \quad b1{:}5\sqrt{h}e$ is a good approximation for a probability of decryption failure of about 2^{-40} per plaintext coefficient.

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.

Selected Parameters We propose four reference parameter sets, each resulting from following Algorithm 8. The parameter sets were selected for their performance and similarity with those in previous works, thus enabling a comparison. For each set,

Algorithm 8: Heuristic Parameter Selection

Input: a security parameter.
Output: The parameters $(N; n; h; Q; P; \ ; d; r; \ _{\mathrm{SF}_n 1}; \ _{\mathrm{SF}_n})$.

1 Select n, N and h and derive $\log(QP)$ according to .
2 Select $_m$ the plaintext scale, the bootstrapping output precision and Q_0 such that Q_0 $_m$.
3 Compute K from h and find d and r such that the polynomial of the EvalSine step in the interval ($K{=}2^r; K{=}2^r$) and degree d gives a precision of about $\log(Q_0{=}\ _m) +$ bits.
4 Select $_{\mathrm{SF}_n 1}$ and $_{\mathrm{SF}_n}$ (the depth of the Coe sToSlots and SlotsToCoe s steps).
5 Allocate the q_j of the Coe sToSlots, EvalSine and SlotsToCoe s steps, with the maximum possible bit-size for all q_j.
6 Select and allocate $P = \prod_{j=0}^{1} p_j$, ensuring that P $jjq \ _i jj$.
7 Run the bootstrapping and find the minimum bit-size for the q_j of the EvalSine such that the output reaches the desired precision or until it plateaus.
8 Run the bootstrapping and find the minimum bit-size for the q_j of the Coe sToSlots such that the output precision is not affected.
9 Run the bootstrapping and find the minimum bit-size for the q_j of the SlotsToCoe s such that the output precision is not affected.
10 Allocate the rest of the moduli of Q such that $\log(QP)$ ensures a security of and check again line 6.
11 If additional residual homomorphic capacity is needed or the security cannot be achieved:
   1. Reduce , $_{\mathrm{SF}_n 1}$ and/or $_{\mathrm{SF}_n}$ and check again line 6.
   2. Increase h to increase $\log(QP)$ and restart at line 1.
   3. Increase N to increase $\log(QP)$ and restart at line 1.

return $(N; n; h; Q; P; \ ; d; r; \ _{\mathrm{SF}_n 1}; \ _{\mathrm{SF}_n})$

Table 4 shows the parameters related to the CKKS scheme and to the bootstrapping circuit.

7 Evaluation

The improved algorithm of Sections 3 and 4, along with the bootstrapping procedure of Section 5 have been implemented in the Lattigo library [21]. 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] and CKKS schemes in Golang [26]. All experiments were conducted single threaded on an i5-6600k at 3.5 GHz with 32 GB of RAM running Windows 10 (Go version 1.14.2, GOARCH=amd64, GOOS=windows).

The Bootstrapping Utility Metric 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, its precision, and the remaining multiplicative depth. In order to evaluate our bootstrapping procedure against the existing ones, we will use the *bootstrapping utility metric*, as introduced by Han and Ki [5] for the same purpose.

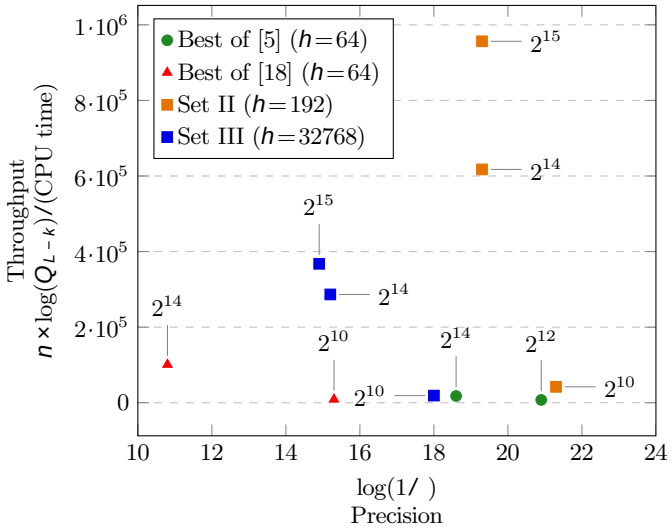


Fig. 5: Bootstrapping utility comparison. We plot the results for our best performing parameter set against the state of the art. Nodes are labeled with n, the number of plaintext slots.

Definition 1 (Bootstrapping Utility). *For n a number of plaintext slots,* $\log(1/\ )$ *the output precision,* $\log(Q_{L\ \ k})$ *the output coefficient-modulus size after the bootstrapping (remaining homomorphic capacity) and* $\mathsf{complexity}$ *a measure of the computational cost (in CPU time), the bootstrapping utility is defined as:*

$$\frac{n \quad \log(1/\ ) \quad \log(Q_{L\ \ k})}{\mathsf{complexity}}.$$

The *bootstrapping* utility can also be interpreted as *bootstrapping throughput* in bits/sec.

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

7.1 Results

We run our benchmarks and compute the bootstrapping utility for each parameter set of Table 4 (Section 6.1) and compare them with the previous works of Chen et al. [5] and Han and Ki [18]. Unfortunately, the implementations of these works have not been publicly released. Consequently we were not able to reproduce their results on our own hardware for a totally fair comparison[4]. The results are summarized in Table 5. Supplementary material 6 reports on several experiments that demonstrate the numerical stability of our bootstrapping procedure.

[4]Chen et al. [18] use an i9-9820X @ 3.3GHz, single-threaded (5-10% more instructions per cycle than our bench). Han and Ki [5] use an i7 @ 2.8 GHz, single-threaded.

Table 5: Performance comparison of prior bootstrapping in [5, 18] and our proposed bootstrapping for the full-RNS variant of CKKS with parameter sets I, II, III, IV. MU, SS, CtS, StC designate ModUp, SubSum, SlotstoCoe s, Coe stoSlots. "-" indicates that the prior work did not report on the value.

Bootstrapping Performance														
Set	h	N	n	Timings (s)						Data (GB)		$\log(Q_{L\ \ k})$	**log (1/ϵ)**	**log (bits/s)**
				MU	SS	CtS	StC	Sine	Total	Keys	DFT			
[5]	64	2^{16}	2^{14}	119.8				38.5	158.3	-	-	172	18.6	18.33
[5]	64		2^{12}	127.5				40.4	167.9	-	-	301	20.9	17.22
[18]	64	2^{16}	2^{14}	-	-	-	-	-	52.8	-	-	370	10.8	20.24
[18]	64		2^{10}	-	-	-	-	-	37.6	-	-	370	15.3	17.23
I	192	2^{16}	2^{15}	0.07	0	7.0	3.5	11.5	21.9	15.9	2.8	550	19.8	23.95
I	192		2^{14}	0.07	0.4	7.0	3.8	5.8	16.0	14.9	2.6	550	20.2	23.43
I	192		2^{10}	0.07	2.1	4.8	2.3	5.8	14.9	6.9	1.1	550	**23.2**	19.74
II	192	2^{16}	2^{15}	0.07	0	5.5	2.4	9.3	17.3	14.3	2.5	505	19.2	**24.13**
II	192		2^{14}	0.07	0.4	5.5	2.6	4.7	13.1	13.4	2.2	505	19.2	23.53
II	192		2^{10}	0.07	1.8	4.2	1.5	4.7	12.3	6.22	1.0	505	21.3	19.77
III	32768	2^{16}	2^{15}	0.08	0	8.0	3.2	29.7	41.0	17.5	2.9	460	14.9	22.39
III	32768		2^{14}	0.08	0.5	7.7	3.2	14.8	26.3	16.4	2.7	460	15.2	22.05
III	32768		2^{10}	0.08	2.4	5.7	1.9	14.8	25.0	7.6	1.2	460	18.0	18.37
[5]	64	2^{15}	2^{10}	28.8				9.5	38.3	-	-	150	6.9	14.75
[5]	64		2^{8}	16.9				9.2	26.0	-	-	75	10.03	12.85
[18]	64	2^{15}	2^{2}	-	-	-	-	-	7.5	-	-	185	15.0	10.53
[18]	64		2^{1}	-	-	-	-	-	7.0	-	-	185	16.8	9.79
IV	192	2^{15}	2^{14}	0.02	0	3.7	0.7	2.6	7.1	7.7	2.1	110	15.1	21.87
IV	192		2^{10}	0.02	0.4	1.6	0.4	1.3	3.9	5.1	0.6	110	16.6	18.87

We observe that, for our best performing parameter set (Set II), we achieve a bootstrapping throughput 15 larger than the best result reported in the work of Han and Ki that was implemented using the SEAL library [24] and conducted on similar hardware. It also provides a throughput 55:9 larger than the best result reported in the work of Chen et al. [5] that was implemented using the library HEAAN [19]. HEAAN does not implement the full-RNS variant of CKKS, hence this second comparison shows the significant performance gains that can be achieved by combining optimized algorithms with a full-RNS implementation. Figure 5 plots our best performing instances against those of the previous works.

We observe that the best results of our bootstrapping consistently occur when the number of slots is set to the maximum (fully-packed ciphertexts). The reason is that even though using 2^{15} slots requires the evaluation of two EvalSine compared to 2^{14} slots, the complexity of the linear transformations remains nearly the same because no repacking involving a matrix multiplication is needed. Taking SubSum into account, we observe that there is in fact no significant difference for the computational time of the linear transformation across all the different slot values. Consequently, at a computational cost that is only that of an additional EvalSine and without significantly affecting the precision, we are able to bootstrap twice the number of slots.

8 Conclusion

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

To achieve this, we have proposed a generic algorithm for the homomorphic evaluation of polynomials that is both scale-invariant and optimal in level consumption. In addition to the increase in precision and efficiency, this also improves the usability of the full-RNS variant of CKKS (for which managing a changing scale in large circuits is known to be a difficult task). We also proposed improved key-switch procedures and applied them to the homomorphic matrix vector multiplication. Our novel *double-hoisted* algorithm reduces the evaluation time of the Coe sToSlots and SlotsToCoe s by roughly a factor of three compared to the 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 will enable much more efficient homomorphic circuits (e.g., in the training of convolutional neural networks).

The measured utility of our bootstrapping procedure with "dense" secret-keys ($h = N/2$) is up to 4.4 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 utility 15 larger than the previous work that uses a sparse key with $h = 64$.

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

References

[1] 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.

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

[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- rst annual ACM symposium on Theory of computing*. 2009, pp. 169–178.

[14] 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.

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

[16] 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.

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

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

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

[20] 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`.

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

[22] 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.

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

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

[25] 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.

[26] *The Go Programming Language*. Online: https://golang.org/. 2020.

Supplementary Material

1 CKKS Homomorphic Operations

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

Add($\mathsf{fct}; Q_\ell; \quad g; \mathsf{fct}'; Q_{\ell'}; \quad {}^0g$) : Scale ct and ct' to max($\quad ; \quad {}^0$) and return $\mathsf{fct}+\mathsf{ct}'$, $\min(Q_\ell, Q_{\ell'})$, max($\quad ; \quad {}^0$)g.

AddPlain($\mathsf{fct}; Q_\ell; \quad g; \mathsf{fpt}; Q_{\ell'}; \quad {}^0g$) : Scale ct and pt to max($\quad ; \quad {}^0$) and return $\mathsf{fct} + (\mathsf{pt}; 0)$, $\min(Q_\ell, Q_{\ell'})$, max($\quad ; \quad {}^0$)g.

AddConst($\mathsf{fct}; Q_\ell; \quad g; a + bi \ 2 \ \mathbb{C}$): Return $\mathsf{fct} + (b \quad (a + b \quad X^{N=2})e, 0), Q_\ell, \quad g$.

Multi($\mathsf{fct}; Q_\ell; \quad g$): Returns $\mathsf{fct} \quad X^{N=2}, Q_\ell; \quad g$, the homomorphic product by the imaginary unit.

MultConst($\mathsf{fct}; Q_\ell; \quad g; a+bi \ 2 \ \mathbb{C}; \quad {}^0$): Return $fb \quad {}^0ae \ \mathsf{ct}+b \quad {}^0be \ \mathsf{ct} \ X^{N=2}; Q_\ell; \quad {}^0g$.

Mul($\mathsf{fct}; Q_\ell; \quad g; \mathsf{fct}'; Q_{\ell'}; \quad {}^0g$): For $\mathsf{ct} = (c_0; c_1)$ and $\mathsf{ct}' = (c_0'; c_1')$, compute $(d_0; d_1; d_2) = (c_0c_0'; c_0c_1' + c_1c_0'; c_1c_1')$, and return $\mathsf{fct}_{\mathsf{mul}} = (d_0; d_1) + \mathsf{SwitchKey}(d_2; \mathsf{rlk})$, $\min(Q_\ell, Q_{\ell'}); \quad {}^0g$:

MulPlain($\mathsf{fct}; Q_\ell; \quad g; \mathsf{fpt}; Q_{\ell'}; \quad {}^0g$): For $\mathsf{ct} = (c_0; c_1)$, return $f(c_0 \ \mathsf{pt}; c_1 \ \mathsf{pt})$, $\min(Q_\ell, Q_{\ell'}); \quad {}^0g$:

Rescale($\mathsf{fct}; Q_\ell; \quad g$): Return $fbq_\ell^{\ 1} \quad \mathsf{cte}; Q_{\ell \ 1}; \quad =q_\ell g$, for $q_\ell^{\ 1} \ 2 \ \mathbb{R}$.

DropLevel($\mathsf{fct}; Q_\ell; \quad g; k$): Return $\mathsf{fct}; Q_{\ell \ k}; \quad g$.

Rotate($\mathsf{fct}; Q_\ell; \quad g; k$): For $\mathsf{ct} = (c_0; c_1)$, return $\mathsf{fct}_{\mathsf{rot}_k} = (c_0^{5^k}; 0) + \mathsf{SwitchKey}(c_1^{5^k}; \mathsf{rot}_k); Q_\ell; \quad g$.

Conjugate($\mathsf{fct}; Q_\ell; \quad g$): For $\mathsf{ct} = (c_0; c_1)$, return $\mathsf{fct}_{\mathsf{conj}} = (c_0^{\ 1}; 0) + \mathsf{SwitchKey}(c_1^{\ 1}; \mathsf{conj}); Q_\ell; \quad g$.

2 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 $(c_0; c_1) = (\quad as^0 + m + e; a)$ that decrypts under s^0, the most efficient approach to switch it to s would be to generate public key-switch keys of the form $\mathsf{swk} = (\quad bs + s^0 + e^0; b)$, and perform re-encryption from s^0 to s as

$$(c_0; 0) + c_1 \quad \mathsf{swk} = (\quad abs + ae^0 + m + e; ab):$$

However, the term ae^0 would introduce too much error for the ciphertext to be correctly decrypted. Fan and Vercauteren [12] propose two switching-keys types to control this error term:

1. Use $\mathsf{swk}^{(i)} = (\ b_i s + w^{(i)} s^0 + e_i^0; b_i)$ for a base $\mathbf{w}$ with the reconstruction formula $x = \sum_{\mathbf{w}} x_{\mathbf{w}}^{(i)} w^{(i)}$, decompose c_1 under base $\mathbf{w}$ and compute $(c_0; 0) + \sum_{\mathbf{w};1} c_{\mathbf{w};1}^{(i)} \mathsf{swk}^{(i)} = (\ a^0 s + \sum_{\mathbf{w}} a_{\mathbf{w}}^{(i)} e_i^0 + m + e; a^0)$. This solution is highly inefficient if the target is to make $\sum_{\mathbf{w}} a_{\mathbf{w}}^{(i)} e_i^0$ small because it will increases the number of keys, and therefore the number of operations, by an amount proportional to $Q/\|\mathbf{w}\|$.
2. Use $\mathsf{swk} = (\ bs + P\ s^0 + e^0; b)$, for P a large integer, and compute $(c_0; 0) + bP^{-1}\ c_1\ \mathsf{swk}e = (\ a^0 s + bP^{-1}\ ae^0 e + m + e; a^0)$. If $P \gg \|ae^0\|$ then the added error is negligible. This solution is more efficient than the type I, but the modulus of the keys is multiplied by P, so the size of the ring degree must be increased or the ciphertext modulus reduced to compensate for the security loss, thus also affecting the overall performances or the homomorphic capacity.

Han and Ki [18] propose a hybrid version that combines both approaches and uses keys of the form $\mathsf{swk}^{(i)} = (\ b_i s + w^{(i)}\ P\ s^0 + e_i^0; b_i)$. Similarly to the type II, if $\|\mathbf{w}\| \ll P$, 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, $a = \sum_{\mathbf{w}} a_{\mathbf{w}}^{(i)} w^{(i)}$, the RNS reconstruction is also a linear operation over a vector, i.e., a sum of products:

$$a \equiv \sum_{[a]_{q_i}} \frac{Q}{q_i} \left[\left[\frac{Q}{q_i} \right]_{q_i}^{-1} \right]_{q_i} \bmod Q. \tag{1}$$

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, 14]. 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.

3 Complexity Analysis

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

3.1 Key-Switch

We analyse the the complexity of a homomorphic multiplication with a key-switch by using our Algorithm 3 (Section 4.1) in term of its number of modular multiplication in $\mathbb{Z}_p$, and we compare it with the results reported in [18].

We assume that the inputs and outputs of the procedure are both in the NTT domain. We set $= \#p_j$ and $= d(\ell+1) =$ e, $\mathsf{ct} = (c_0; c_1)$ and $\mathsf{ct}' = (c_0'; c_1')$ mod Q_ℓ.

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

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

Step 3: *MultSum*. We decompose d' base q_i, multiply it with evk and sum. So for $0 < i <$,

1. We apply Decompose to $d' \in R_{Q_\ell}$: We are given an input vector of $\ell+1$ elements that we take modulo q_i, thus reducing its size to elements. This first operation is free as $q_i \mid Q_\ell$. Using the ModUp algorithm, we then extend this vector to a vector of size $(\ell+1)+$, but for which we already know elements, hence the complexity is $(\ell+1+$ $) =$ $(\ell+1)$. We also have to run pre-computations on the fly. As we have to run this for N values, the total complexity is N $($ $+$ $(\ell+1))$.
2. We switch $d'_{q_i} \in R_{Q_\ell P}$ back to the NTT domain : We need to compute $(\ell+1)+$ NTT, but we already have 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 $\log(N)$ $(\ell+1)$.
3. We multiply $d_{q_i} \in R_{Q_\ell P}$ with $\mathsf{evk}^j_{q_i}$ for $j \in 0; 1$. The complexity is 2 N $(\ell+1+$ $)$.

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

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

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

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

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

$$N \quad (\ell+1) \quad \log(N) \quad (3+ \quad) + \quad (\quad +2) + 2 \quad +6 \quad + \quad (2 \quad \log(N) + 2 + 3 \quad) \ :$$

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

Whereas, in [18], they report a complexity of

$$N \quad (\grave{} + 1)^2 + (\grave{} + 1) \quad (\quad + 2 \quad + 6) + 3 + \log(N) \quad (\grave{} + 1) \quad (\quad + \quad + 5) + 3 \quad .$$

Table 6 compares both complexities by using the same parameters as the original table of [18] with $\grave{} = 23$ and a variable $\#p_j = \quad$. The size of the moduli of Q and P is of 45 bits and q_0 is 55 bits. Our tweaked algorithm has the same asymptotic complexity but introduces a change in the constants, which is enough to induce a non-negligible difference. The number of NTT, which is the dominant term, is $(\grave{} + 1) \quad (\quad + \quad + 5) + 3$ in [18], whereas it is $(\grave{} + 1) \quad (\quad + 3) + 2 \quad$ in our work. The number of NTT in our algorithm decreases much faster for larger than those of Han and Ki, e.g., for $= 6$ it already shows a factor of two difference. As the number of NTT is the dominant term of the key-switch, this translates into a non-negligible difference in the final complexities.

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

		log(#Mul in $\mathbb{Z}_p$)	
	log(QP)	Work in [18]	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

3.2 (Hoisted) Rotations

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

Step 2 : Decompose + NTT. We decompose c_1^0 mod each $q \quad _i$, extend the RNS basis from $Q\grave{}$ to $Q\grave{}P$ and switch back the result in the NTT domain : $\quad N \; (\quad +(\grave{}+1) \; (\log(N)+ \quad))$.

Step 3 : MultSum. For each k rotation we multiply d_i^k with $\mathsf{rot}_{k;q \quad _i}$ and sum : $2k \quad N$ $(\grave{} + 1 + \quad)$.

Step 4 : ModDown. For each k rotation we rescale a and b by P and reduce the RNS basis from $Q\grave{}P$ back to $Q\grave{}$: $2k \quad N \quad ((\grave{} + 1) \quad (\quad + \log(N) + 1) + \quad (\log(N) + 1))$.

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

Hence, the total complexity for k hoisted rotations is

$$N \log(N) \quad (\grave{}+1)\,(\ +1+2k)+2k \quad +N\ (\grave{}+1) \qquad +2k\,(\ +\ +1)\ +N \qquad +2k\,(\ +1)\ :$$

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

4 Algorithms

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

Exact base conversion using floating point arithmetic from Halevi et al. [14]:

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

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

$$v = \left\lfloor \sum_{j=0}^{\grave{}-1} \frac{[a\ \ \hat{q}_j\ ^{-1}]_{q_j}}{q_j} \right\rfloor :$$

Algorithm 9: ModUp

Input: $[a]_{Q\cdot}$ a polynomial with N coefficients, P.
Output: $[a]_P$.

```
for 0    i < N do
    [a^(i)]_P = Conv^exact_{Q` ! P}([a^(i)]_{Q`})
end
return [a^(i)]_P
```

Algorithm 10: Decompose

Input: $a \in R_{Q_\ell}$ in the NTT domain, $Q_\ell = \prod_{j=0}^{\ell} q_j$, $q_{[i]} = \prod_{i=\ldots}^{\min(\ldots,\ell)} q_j$ for $0 \leq i < \ldots$, $beta = d(\ell+1)/e$ and $P = \prod_{j=0}^{k-1} p_j$.

Output: **d** $\in R_{Q_\ell P}$ in the NTT domain.

```
a' ← NTT^{-1}(a)
foreach q_[i] do
    d^(i) ← ModUp_{q_[i] → Q_ℓ P}([a']_{q_[i]})
    foreach q_j do
        if q_j | q_[i] then
            [d^(i)]_{q_j} ← [a]_{q_j}
        else
            [d^(i)]_{q_j} ← NTT([d^(i)]_{q_j})
        end
    end
    foreach p_j do
        [d^(i)]_{p_j} ← NTT([d^(i)]_{p_j})
    end
end
return d
```

Algorithm 11: ModDownNTT

Input: $a \in R_{Q_\ell P}$ in the NTT domain

Output: $\lfloor P^{-1} a \rceil \in R_{Q_\ell}$ in the NTT domain

```
[b]_P ← NTT^{-1}([a]_P)
[b]_{Q_ℓ} ← ModUp_{P → Q_ℓ}([b]_P + ⌊Q_ℓ/2⌋) − ⌊Q_ℓ/2⌋
c ← NTT([b]_{Q_ℓ})
[a]_{Q_ℓ} ← [a]_{Q_ℓ} − [c]_{Q_ℓ}
return [P^{-1}]_{Q_ℓ} · [a]_{Q_ℓ}
```

Algorithm 12: ModRaise

Input: p a polynomial of N coefficients in basis $Q_0 = q_0$.
Output: p the same polynomial in basis $Q = \prod_{i=0} q_i$, with $p[j][i]$ the i-th coefficient of the basis q_j.

```
p ← NTT^{-1}(p)
for 0 ≤ i < N do
    x ← p[0][i]
    for 1 ≤ j < ℓ do
        if x ≥ bq_0/2c then
            p[j][i] = q_j + (q_0 − x) mod q_j
        else
            p[j][i] = x mod q_j
        end
    end
end
p ← NTT(p)
return p
```

Algorithm 13: SubSum

Input: ct a ciphertext, n the number of slots of the encoded in the plaintext, N the degree of the ring.

```
for log(n) ≤ i < log(N) − 1 do
    tmp ← Rotate_{2^i}(ct)
    ct ← Add(ct, tmp)
end
return
```

Algorithm 14: FFT Algorithm For Evaluating $\mathrm{SF}_n{}^1$

Input: $\mathbf{w}$ 2 $\mathbb{C}^n$, $n > 1$ a power of 2 integer, a pre-computed table of $4n$-th roots of unity such that $^{(j)} = e^{i\ j=2n}$ for $0 \quad j \quad 2n$.

Output: $\mathrm{SF}_n{}^1$ $\mathbf{w}$.

```
1  for m = n; m   2; m = m=2 do
2  |  for i = 0; i < n; i = i + m do
3  |  |  for j = 0; j < m=2; j = j + 1 do
4  |  |  |  k = 4m   (5^j mod 4m)  (n=m)
5  |  |  |  u = w^(i+j) + w^(i+j+m=2)
6  |  |  |  v = w^(i+j)   w^(i+j+m=2)
7  |  |  |  w^(i+j) = u
8  |  |  |  w^(i+j+m=2) = v   (k)
9  |  |  end
10 |  end
11 end
12 bitReverse(w, n)
13 for i = 0; i < n; i = i + 1 do
14 |  w^(i) = n  1  w^(i)
15 end
16 return w
```

5 Basic Operations Performances

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

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

6 Bootstrapping Stability Experiments

We carried out several experiments to validate the stability of our bootstrapping procedure. Supplementary material 6 reports on the following checks: The mean precision across all the slots against the number of slots (Supplementary material 6.1), the probability of each slot to fall under some given precision (Supplementary material 6.2), and the mean precision across all the slots after each bootstrapping for 50 successive bootstrapping (Supplementary material 6.3).

6.1 Precision vs. Slots

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

(a) Set I

(b) Set II

(a) Set III

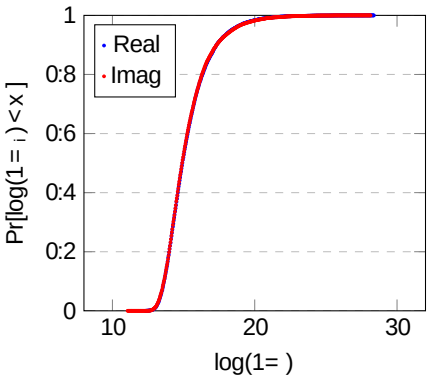
(b) Set IV

6.2 Precision Distribution

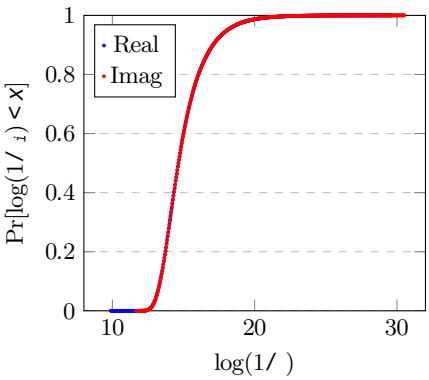
In this section, we plot the probability of each slot to fall under some given precision for the di erent parameters presented in Section 6.1. The goal is to show that though we have a good mean precision, the overall distribution also behaves well and is not scattered. The plaintext values that were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

(a) Set I - 2^{14} (b) Set I - 2^{15}

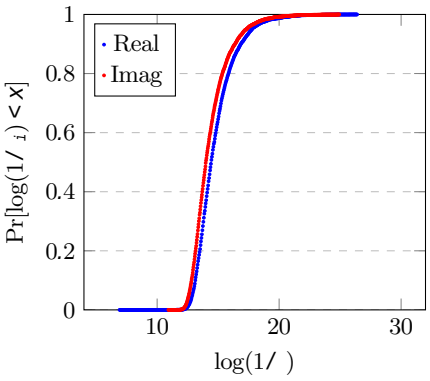
(a) Set II - 2^{14} (b) Set II - 2^{15}



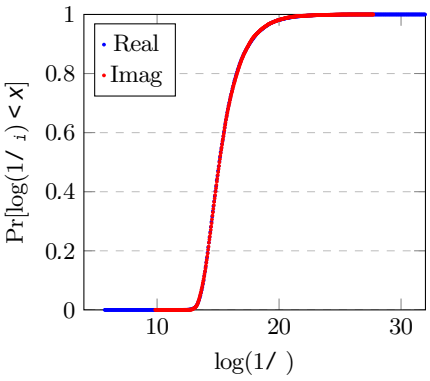
(a) Set III - 2^{14}



(b) Set III - 2^{15}



(a) Set IV - 2^{13}



(b) Set IV - 2^{14}

6.3 Successive Bootstrappings

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

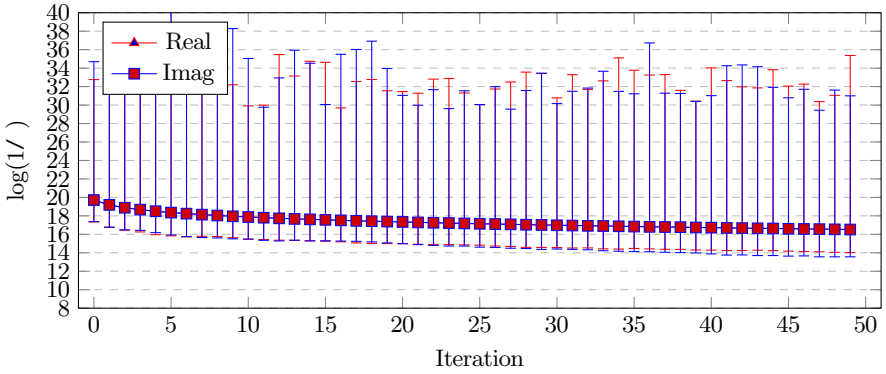


Fig. 12: Set I - 2^{15} slots

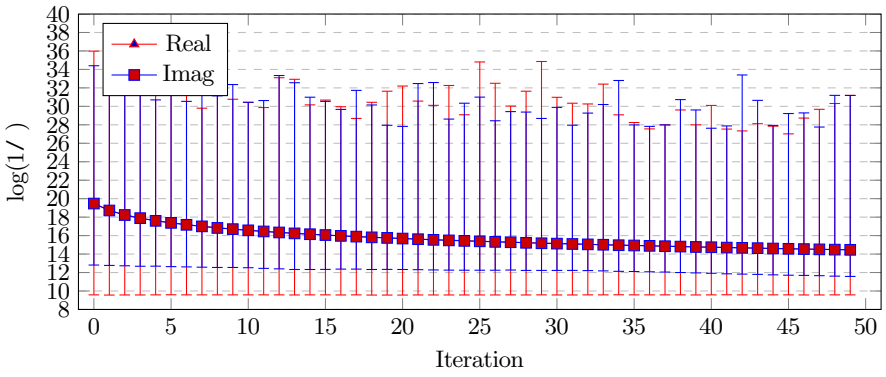


Fig. 13: Set II - 2^{15} slots

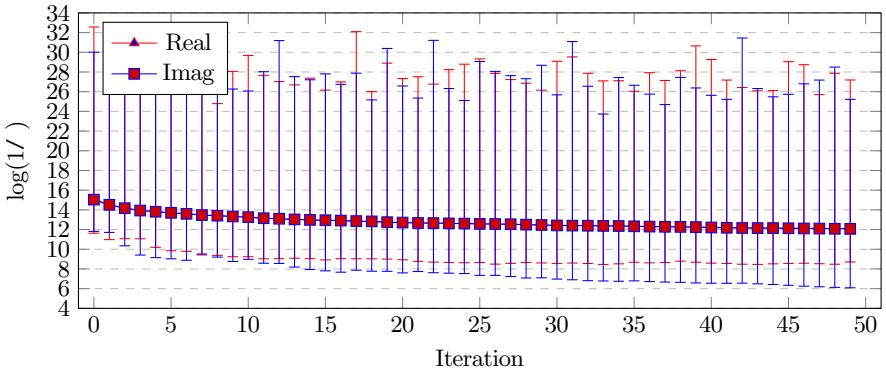


Fig. 14: Set III - 2^{15} slots

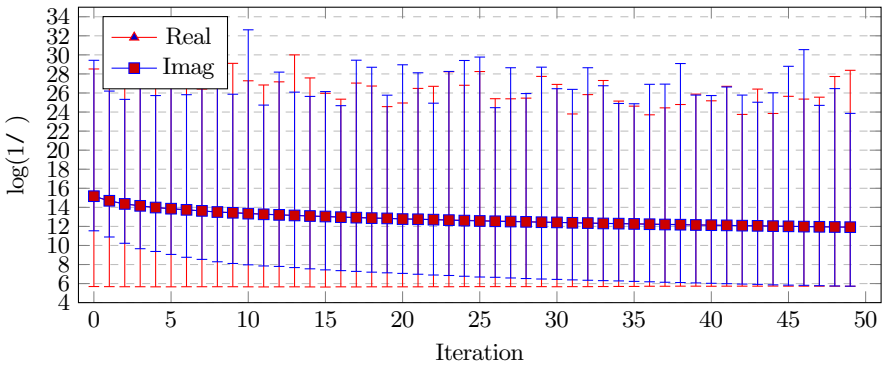


Fig. 15: Set IV - 2^{14} slots