Large-Precision Homomorphic Sign Evaluation using FHEW/TFHE Bootstrapping*

Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov

Duality Technologies

September 17, 2022

Abstract

A comparison of two encrypted numbers is an important operation needed in many machine learning applications, for example, decision tree or neural network inference/training. An efficient instantiation of this operation in the context of fully homomorphic encryption (FHE) can be challenging, especially when a relatively high precision is sought. The conventional FHE way of evaluating the comparison operation, which is based on the sign function evaluation using FHEW/TFHE bootstrapping (often referred in literature as *programmable bootstrapping*), can only support very small precision (practically limited to 4-5 bits or so). For higher precision, the runtime complexity scales linearly with the ciphertext (plaintext) modulus (i.e., exponentially with the modulus bit size). We propose sign function evaluation algorithms that scale logarithmically with the ciphertext (plaintext) modulus, enabling the support of large-precision comparison in practice. Our sign evaluation algorithms are based on an iterative use of homomorphic floor function algorithms, which are also derived in our work. Further, we generalize our procedures for floor function evaluation to arbitrary function evaluation, which can be used to support both small plaintext moduli (directly) and larger plaintext moduli (by using a homomorphic digit decomposition algorithm, also suggested in our work). We implement all these algorithms using the PALISADE lattice cryptography library, introducing several implementation-specific optimizations along the way, and discuss our experimental results.

*This work was funded primarily by Duality Technologies. This material is partially based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112090102.

Contents

1 Introduction

The ability to compare two encrypted numbers is required in many real-world applications, and often these applications need to combine comparisons with arithmetic operations, such as additions or multiplications (e.g., neural network or decision tree inference/training [3, 25]). The main non-interactive method for performing these computations in a privacy-preserving manner is fully homomorphic encryption (FHE), a powerful cryptographic primitive that enables performing computations over encrypted data without having access to the secret key.

The FHE schemes are generally broken down into three classes: the FHEW/TFHE schemes for evaluating boolean circuits, which are best suited for comparisons and decision diagram computations [16, 19, 29]; Brakerski-Gentry-Vaikuntanathan (BGV) and Brakerski/Fan-Vercauten (BFV) schemes for evaluating modular arithmetic over finite fields, which are also often applied for small-integer computations [9, 10, 20]; and Cheon-Kim-Kim-Song (CKKS) scheme for approximate computations over real and complex numbers [14].

One of the open challenges is that although the CKKS scheme can efficiently support additions, multiplications, and more generally, polynomial function evaluation, with relatively high precision, the current FHE capabilities of evaluating the encrypted comparison is limited. One method to resolve this problem is to use scheme switching between CKKS and FHEW/TFHE, first introduced in the CHIMERA paper by Boura at. al [8], and later improved in the PEGASUS paper by Lu et. al [3]. However, after switching to FHEW/TFHE the comparison capability for these "high-precision" numbers is very limited. For instance, we show in Section 7 that a single FHEW/TFHE bootstrapping, a typical way to perform an encrypted comparison in FHE, can efficiently support at most 4 bits of precision for encrypted comparison using typical parameters as in [29], which is also close to the precision used in [3]. Any further precision improvement for this method makes the encrypted comparison highly inefficient. Therefore, there is a significant interest in developing methods for large-precision comparison of encrypted numbers that would scale significantly better (both asymptotically and practically) with input precision.

The comparison of two encrypted numbers is equivalent to computing the difference of these numbers followed by the evaluation of the sign function. As evaluating the difference is trivial for any additively homomorphic encryption scheme, the difficulty lies in the sign function computation. In the rest of the paper, we will focus on the sign function, assuming that all our results for the sign function readily apply to encrypted comparison.

The sign function evaluation is closely related to the main idea of FHEW/TFHE bootstrapping, where we need to find the most significant bit (MSB) of an encrypted number. Hence, one could directly apply the FHEW/TFHE bootstrapping to find the sign. However, this approach only works for a very limited precision (up to 4 bits, as pointed out above) for the parameters currently used for efficient Boolean circuit evaluation [1, 29]. The complexity of the FHEW/TFHE bootstrapping procedures scales linearly with the ciphertext modulus Q, i.e., exponentially with the bit-size of Q. This implies that already for 10 bits of precision, one would need to increase the runtime by a factor of $2^6 = 64$, as compared to the current results for Boolean arithmetic. Clearly, this approach is not viable for practical applications that require 10 or even more bits of precision.

A major goal of our work is to develop a sign function evaluation procedure that scales logarithmically with Q. We also use the central idea of our sign evaluation algorithm to derive efficient general functional bootstrapping procedures, which support the evaluation of arbitrary functions. Note that functional bootstrapping is often also called *programmable bootstrapping* [18].

Our Contributions More concretely, the contributions of our work can be summarized as follows:

- We propose a novel procedure for large-precision homomorphic sign evaluation using FHEW/TFHE bootstrapping: a large-precision ciphertext is broken down into digits, and then the homomorphic floor function is executed sequentially to clear each digit, starting from the least significant one. After each digit is cleared, the ciphertext is scaled down to work with a smaller ciphertext modulus Q, until at the last iteration the current modulus becomes small enough to evaluate the fast FHEW/TFHE bootstrapping procedure (with the same parameters as used for Boolean arithmetic).

- We develop two algorithms for the homomorphic floor function. The first algorithm requires two invocations of FHEW/TFHE bootstrapping and has a specific constraint for the input noise. The second algorithm requires three invocations of FHEW/TFHE bootstrapping, but has no constraint on the input noise.

- We use the central idea of the homomorphic floor function algorithms to develop a general functional bootstrapping procedure, which supports arbitrary functions for small plaintext spaces (up to 4 bits in practical settings). Our general functional bootstrapping procedure has asymptotically smaller noise than other recent works.

- We derive a homomorphic digit decomposition algorithm based on the sign-evaluation algorithm to extend the general functional bootstrapping procedure to larger plaintext spaces.

- We implement all these capabilities using the PALISADE lattice cryptography library, introducing several implementation–specific optimizations. Our comparison of the two algorithms for floor function evaluation implies that the method based on two invocations of bootstrapping is always more efficient in practice. We also demonstrate an application of our method in the context of a CKKS-based computation.

Techniques We describe a method to compute the sign of an encrypted value using bootstrapping techniques. The input is the encryption of a numerical value $m \in \mathbb{Z}$, usually a signed integer, or a fractional number in fixed-point, binary, two's complement representation. We assume the input is presented as an LWE ciphertext, i.e., a vector of elements in $\mathbb{Z}_Q$. The message m is an integer modulo Q/α. We assume that $\alpha = 2^l$ and $Q = 2^h$ are powers of 2, so that the message m can also be interpreted as a $(h-l)$-bit integer. The problem is to compute an encryption of the most significant bit of m, i.e., $\lfloor m/2^{h-l-1} \rfloor$. If $m \in \mathbb{Z}_{2^{h-l}}$ is the standard (two's complement) representation of a signed integer, this bit is the sign of m, i.e., it equals 1 if and only if m represents a negative number.

We treat FHEW/TFHE bootstrapping as a black box, implying that any of the bootstrapping functions described in [16, 19, 29] can be used interchangeably. For conciseness, we refer to this function as FHEW bootstrapping in the rest of the paper.

FHEW supports functional/programmable bootstrapping for negacyclic functions, i.e., functions $f\colon \mathbb{Z}_Q \to \mathbb{Z}$ satisfying $f(x + Q/2) = -f(x)$. If we add $\alpha/2$ to the LWE ciphertext, yielding a modified message $m' = \alpha m + e + \alpha/2$, where e is the noise, and define a sign function $\gamma\colon \mathbb{Z}_Q \to \{-1, +1\}$, mapping $\gamma(x) = +1$ for $x \in \{0, \ldots, Q/2 - 1\}$ and $\gamma(x) = -1$ for $x \in \{-Q/2, \ldots, -1\}$, we can directly apply the FHEW bootstrapping procedure for the evaluation

function γ (it is easy to observe that γ is already negacyclic). The problem is that the complexity of the FHEW bootstrapping procedure (in particular, the size of the FHEW accumulators) is linear in the ciphertext modulus Q. So, while conceptually the sign computation can be performed directly using the FHEW procedure, the resulting algorithm would be terribly inefficient, both in theory (exponential in the bit size of the input) and in practice.

To circumvent this problem, we "break down" the ciphertext modulo Q into multiple digits, each working internally with a much smaller modulus q, which enables the use of efficient FHEW bootstrapping. For each digit, we evaluate a homomorphic floor function that can be used to clear the least significant digit from the ciphertext. As soon as the current least significant digit is cleared, the ciphertext is scaled down using modulus switching from Q to $\alpha Q/q$. This iterative procedure is repeated until Q becomes less than or equal to q. At that point, efficient FHEW bootstrapping for $\gamma(x)$ can be used directly to evaluate the sign function. Conceptually, this algorithm corresponds to the "schoolbook" long division algorithm. The main challenge in this long division algorithm is associated with evaluating the floor function, which is not negacyclic and hence cannot be directly evaluated using FHEW bootstrapping.

The idea of our first floor function algorithm is to first evaluate the sign function $\gamma(x)$ to clear the MSB of each digit (first bootstrapping) and then subtract the remaining bits in the digit using the second invocation of FHEW bootstrapping. Both of these evaluation functions are negacyclic, enabling us to use FHEW bootstrapping. If we had a perfect (noiseless) bootstrapping procedure, this would take care of clearing all the bits of the digit. But FHEW bootstrapping (just like any lattice-based bootstrapping procedure) is noisy. In order to accommodate for the bootstrapping noise, this method requires the introduction of a constraint on the noise of the input ciphertext: $\beta \leq \alpha/4$, where $|e| < \beta$. This floor function algorithm can clear up to q/α bits.

We also propose an alternative floor function, which does not have the input noise constraint, but requires an extra invocation of FHEW bootstrapping. The first invocation of FHEW bootstrapping is used to clear the second-most significant bit in the digit. Intuitively, this first invocation has the effect of enforcing the $\beta \leq \alpha/4$ constraint of the first floor computation algorithm. So, we can proceed with another invocation of FHEW bootstrapping that clears the MSB, and, finally, the remaining bits in the digit are cleared using the third invocation of FHEW bootstrapping. In other words, the main difference between the two floor function algorithms is in the first bootstrapping operation, which clears the second-most significant bit. In practice, the alternative floor function evaluation algorithm gains one extra bit of precision compared to the first algorithm, but has a cost of an additional invocation of FHEW bootstrapping.

Then, we generalize the algorithms for homomorphic floor function to arbitrary function evaluation for small plaintext moduli, i.e., restricting the ciphertext modulus to q that supports efficient FHEW bootstrapping. Consider the generalization of our first floor function algorithm as an example. We first extend the ciphertext from modulus q to $2q$. This introduces, as a byproduct, a random MSB modulo $2q$. Then we evaluate the $\gamma(x)$ function modulo $2q$ to clear this MSB. Finally, we invoke the desired function for the remaining bits unaffected by noise. Compared to the homomorphic floor function, we loose just one bit of precision.

Finally, we derive a homomorphic digit decomposition algorithm that can be combined with the general functional bootstrapping for small-precision ciphertexts to achieve the evaluation of arbitrary functions over large-precision ciphertexts, i.e., evaluate large lookup tables. The digit decomposition algorithm is closely related to the homomorphic sign evaluation algorithm: it basically performs the same sequence of applications of the homomorphic floor function evaluation, while

keeping track of the (encrypted) digits produced by each invocation.

Note that most of the homomorphic encryption schemes support the efficient extraction of LWE ciphertexts. So the methods described here can be applied to those schemes by first extracting an LWE representation of the input, and then applying the main algorithm. For details on the algorithms for efficient extraction of LWE ciphertexts, we refer the reader to [12, 25].

1.1 Related Works

Related Concurrent Works Two concurrent and independent works [18, 27] propose algorithms for homomorphic evaluation of arbitrary functions for small plaintext moduli. Table 1 summarizes the results of the comparison between our main algorithm for arbitrary function evaluation with their algorithms. An expanded comparison with concrete parameters is presented in Section 8.

Table 1: Comparison of noise growth and complexity of our method for arbitrary function evaluation with other recent works; here, β is the FHEW functional bootstrapping noise (see details in Section 6.5), N is the ring dimension used for functional bootstrapping, p is the plaintext modulus, Q' is the underlying RLWE ciphertext modulus, q is the output LWE ciphertext modulus, and $d'_g \geq 2$ is the number of digits for gadget decomposition specific to functional bootstrapping in [27].

	Noise Growth	# of bootstrappings
[18]	$\beta \cdot O(Np)$	2
[27]	$\beta \cdot O(\sqrt{Nd'_g}Q'^{1/d'_g})$	$d'_g + 1$
Our work	β	2

The main idea of both works is to use the fact that $-1 \cdot (-m) = m$ and extract the MSB as part of their procedures by invoking FHEW/TFHE bootstrapping. Both approaches hence require one multiplication operation, which increases the noise requirements. This also implies that the main homomorphic encryption scheme should support both additions and multiplications. Our approach does not require any multiplications, and can be applied to any additively homomorphic encryption scheme, similar to the Boolean circuit construction in the original FHEW paper [19].

The approach in [18] executes two bootstrapping operations (one to extract the MSB and another to evaluate the desired function), and then multiplies the results using a multiplication operation similar to the one in Brakerski's and BFV schemes [9, 20]. As a result, the noise increases by $O(Np)$, which implies that the cost of the bootstrapping operations in this method is higher than in ours. Our analysis in Section 8 predicts that the runtime complexity will be at least two times higher for practical parameters.

The method in [27] applies the same blueprint, but instead of performing a BFV-like multiplication, initially uses a multiplication by a GSW ciphertext, and then further optimizes it to replace it with a cheaper multiplication by an LWE$'$ ciphertext (i.e., a vector of LWE ciphertexts, see details in [29]). This approach requires at least $d'_g + 1$ bootstrapping operations, where d'_g is a design parameter. Note that the value of d'_g also affects the noise growth. If the noise cost is minimized (a larger d'_g is chosen), then the number of bootstrapping invocations increases. It is clear that the method in [27] is always at least 1.5x slower than ours as $d'_g \geq 2$, and it also substantially increases the noise unless d'_g is much larger than 2.

Both methods [18, 27] can be extended to support large-precision sign evaluation (though this was not done in these works), but will have the same drawbacks (compared to our approach) as for

arbitrary function evaluation: asymptotically higher noise growth (both methods) and (for [27]) increased number of bootstrapping operations.

A recent paper [32] independently developed an arbitrary function evaluation method similar to ours. This work was published after our results became available and hence we do not examine it here.

Other Approaches for Evaluating Sign Function Although we focus on the approaches to evaluating the comparison/sign functions based on FHEW/TFHE bootstrapping, other methods have also been considered in literature.

We note that all of the methods described below have their own merits and method selection is application-dependent. For instance, the FHEW/TFHE-based method is preferred when only a small number of comparisons are needed or a small number of levels are available for the comparisons. The CKKS-based method may work better when a large number of comparisons are needed in parallel and a sufficient multiplicative depth or CKKS bootstrapping are available (see Section 8.2 for details). The desired precision of comparison is also an important factor. A comprehensive comparison of these methods is outside the scope of this paper and is suggested as a topic for future work.

One approach is based on evaluating special interpolation polynomials over finite fields using the BGV or BFV scheme (see [24] for an extensive review of these techniques). This approach does not typically require bootstrapping but involves a complicated encoding of interpolation polynomials into the native polynomial space of BGV and BFV. Although high efficiency can be achieved (this method may even have a smaller complexity than the techniques considered in our work), this approach is somewhat special-purpose and becomes challenging when the comparison operations need to be combined with multiplications and additions. The main advantage of our approach is the ability to combine comparisons with regular arithmetic operations, resulting in a more general functionality.

Another approach is based on minimax or other polynomial approximations using the CKKS scheme (see [15, 28] for recent results). This approach can be very efficient for relatively small precision, and takes full advantage of CKKS packing. However, the input numbers typically have to be within a specific known range, and the runtime complexity may sharply increase with precision or minimum difference allowed between two numbers. In contrast, the computational complexity of our approach is guaranteed to scale linearly with the number of precision bits, and does not depend on how close two numbers are to each other, i.e., how close the value of the sign function input is to zero. We provide a high-level comparison between the CKKS method and our approach in Section 8.2.

A leveled bit-wise version of TFHE (without bootstrapping) was also previously considered. For example, Chillotti et al. showed that two $(\log p)$-bit numbers can be compared by evaluating a deterministic automaton made of $5 \log p$ CMux gates [17]. Though this comparison complexity is much smaller than for the approach considered in our paper, it has the drawback of requiring the input to be encrypted in a bit-wise fashion. So, their approach will quickly become inefficient in scenarios where comparisons need to be combined with additions and multiplications, as these operations are very expensive in bit-wise representation. Note that our main motivation for developing the general comparison capability based on FHEW/TFHE bootstrapping was to support mixed computations involving additions, multiplications, or, more generally, polynomial evaluation, as well as comparisons.

Another potentially promising approach is based on a limited form of functional bootstrapping supported by BFV/BGV. Chen at al. show how BFV bootstrapping can be used to compute the sign function [13]. It is not clear whether the BFV/BGV approach can be extended to arbitrary functions (look-up tables), but it is certainly an interesting research problem.

1.2 Organization

The rest of the paper is organized as follows. In Section 2 we provide the necessary background on FHEW bootstrapping. Section 3 describes our algorithms for homomorphic sign and floor evaluation. Section 4 shows how our homomorphic floor algorithms can be generalized to arbitrary function evaluation. Section 5 introduces homomorphic digit decomposition algorithms based on our sign evaluation algorithms. Section 6 discusses how parameters should be selected, and introduces some optimizations. Section 7 describes our implementation and presents experimental results, and Section 8 compares our algorithms with other concurrent works. Section 9 discusses an application of large-precision comparison. Section 10 concludes the paper.

2 Background

All logarithms are expressed in base 2 if not indicated otherwise. Vectors are indicated in bold, e.g., $\mathbf{a}$. We choose the ring dimension N as a power of two for efficiency reasons.

2.1 FHEW Functional/Programmable Bootstrapping

In this section we recall the definition of LWE ciphertexts [30], and the properties of the FHEW [19] "functional" bootstrapping procedure needed by our algorithms.

The LWE cryptosystem [30] is parametrized by a plaintext modulus p, ciphertext modulus q, and secret dimension n. The LWE encryption of a message $m \in \mathbb{Z}_p$ under (secret) key $\mathbf{s} \in \mathbb{Z}^n$ is a vector $(\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$ such that

$$b = \langle \mathbf{a}, \mathbf{s} \rangle + (q/p) \cdot m + e \pmod q$$

where e is a small error term, $|e| < q/(2p)$. The message m is recovered by first computing the approximate LWE decryption function

$$\mathsf{Dec}_{\mathbf{s}}(\mathbf{a}, b) = b - \langle \mathbf{a}, \mathbf{s} \rangle \pmod q = (q/p) \cdot m + e$$

and then rounding the result to the closest multiple of (q/p).

The ciphertext modulus of LWE ciphertexts can be changed (at the cost of a small additional noise proportional to the secret key size) simply by scaling and rounding its entries, as described in the following lemma.

Lemma 1 (Modulus Switching)**.** *Let $(\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$ be an LWE encryption of a message $m \in \mathbb{Z}_p$ under secret key $\mathbf{s} \in \mathbb{Z}^n$ with ciphertext modulus q and noise bound $|\mathsf{Dec}_{\mathbf{s}}(\mathbf{a}, b) - (q/p)m| < \beta$. Then, for any modulus q', the rounded ciphertext $(\mathbf{a}', b') = \lceil (q'/q) \cdot (\mathbf{a}, b) \rfloor$ is an encryption of the same message m under $\mathbf{s}$ with ciphertext modulus q' and noise bound $|\mathsf{Dec}_{\mathbf{s}}(\mathbf{a}', b') - (q'/p)m| < (q'/q)\beta + \beta''$, where $\beta'' = \frac{1}{2}(\|\mathbf{s}\|_1 + 1)$.*

In practice, when the input ciphertext is sufficiently random, or when modulus switching is performed by *randomized* rounding, it is possible to replace the additive term β'' with a smaller probabilistic bound $O(\|\mathbf{s}\|_2)$. For uniformly random ternary keys $\mathbf{s} \in \{0, 1, -1\}^n$, this is $\beta'' \approx O(\sqrt{n})$.

A key feature of FHEW is that it allows to perform certain homomorphic computations (described by an "extraction" function) on ciphertexts during bootstrapping at no additional cost. We will use (a slight generalization of) the FHEW [19] bootstrapping procedure, and its optimized variants for binary [16] and ternary secrets [29], as implemented in PALISADE. The bootstrapping algorithm is parametrized by

- a dimension n and (input ciphertext) modulus q, where q is a power of 2,
- a secret key $\mathbf{s} \in \mathbb{Z}^n$, which must be a short vector. Here we assume $\mathbf{s} \in \{0, 1, -1\}^n$,
- a large ciphertext modulus Q' used internally to the bootstrapping procedure, and which is not required to be a power of 2,
- an output ciphertext modulus Q, which we set to a power of 2 possibly different from q, and
- an extraction function $f \colon \mathbb{Z}_q \to \mathbb{Z}$ which must satisfy the negacyclic constraint

$$f(x + q/2) = -f(x). \tag{1}$$

The bootstrapping procedure also uses a bootstrapping key, which is computed from $\mathbf{s}$, but can be made public. Since this bootstrapping key is only used internally by the bootstrapping procedure, we omit it from the notation.

We remark that, since $\mathbf{s}$ is a small vector (e.g., with ternary entries $\{0, 1, -1\}$), it can be used as a key both modulo q, and modulo Q' or Q. On input an LWE ciphertext $(\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$, the FHEW bootstrapping procedure first computes an LWE ciphertext $(\mathbf{c}', d') \in \mathbb{Z}_{Q'}^{n+1}$ such that

$$\mathsf{Dec}_{\mathbf{s}}(\mathbf{c}', d') = f'(\mathsf{Dec}_{\mathbf{s}}(\mathbf{a}, b)) + e' \pmod{Q'},$$

where the noise bound $|e'| \leq \beta'$ depends only on the computation performed during bootstrapping (and not the input ciphertext), and

$$f'(x) = \left\lceil \frac{Q'}{Q} \cdot f(x) \right\rfloor$$

is a scaled version of f still satisfying the negacyclic condition (1). Then, modulus switching is applied to $(\mathbf{c}', d')$ to obtain a ciphertext $(\mathbf{c}, d) = \left\lceil \frac{Q}{Q'}(\mathbf{c}', d') \right\rfloor \in \mathbb{Z}_Q^{n+1}$ modulo Q such that

$$\mathsf{Dec}_{\mathbf{s}}(\mathbf{c}, d) = f(\mathsf{Dec}_{\mathbf{s}}(\mathbf{a}, b)) + e \pmod{Q}$$

where $|e| < \beta = (Q/Q')\beta + \beta''$ is the noise bound from Lemma 1.

For the sake of comparison, we recall that in the original FHEW bootstrapping procedure:

- the input LWE ciphertext $(\mathbf{a}, b)$ uses plaintext modulus $p = 4$, so that messages $m \in \{0, 1, 2, 3\}$ are encoded as multiples of $\alpha = q/4$, i.e., $\mathsf{Dec}_{\mathbf{s}}(\mathbf{a}, b) = (q/4) \cdot m + e$ for some error $|e| < q/8$;

- the output modulus $Q = q$ is the same as the input modulus, so that bootstrapping operations can be composed into arbitrary circuits;
- the extraction function f maps the interval $[-q/8, 3q/8) \subset \mathbb{Z}_q$ to $q/8$ and (necessarily, to satisfy (1)) the interval $[3q/8, 7q/8)$ to $-q/8$. Moreover, the output ciphertext is modified to $(\mathbf{c}, d + q/8)$, so that the final output is either an encryption of $q/8 + q/8 = q/4 = 1 \cdot \alpha$ (i.e., an encoding 1) when $m \in \{0, 1\}$, or an encryption of $-q/8 + q/8 = 0 \cdot \alpha$ (i.e., an encoding of 0) when $m \in \{2, 3\}$. This allows to evaluate the NAND of two input bits $m_0, m_1 \in \{0, 1\}$ as $f(m_0 + m_1 \bmod 4)$.

In this paper, we make extensive use of the FHEW bootstrapping procedure, but for a larger output modulus Q, where $q \leq Q < Q'$, and a number of different (but still negacyclic) extraction functions f.

We write

$$\mathsf{Boot}[f](\mathbf{a}, b)$$

for the result of invoking this bootstrapping procedure for a given function f. We will make blackbox use of Boot, so that the internal workings of the bootstrapping procedure are not important for the rest of the paper, and Boot can be implemented either using the original FHEW bootstrapping procedure [19] or the optimized versions proposed in [16, 29]. The properties of the Boot function described in this section and needed in the rest of the paper are summarized in the following theorem.

Theorem 1. *For any LWE ciphertext $(\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$ and function $f \colon \mathbb{Z}_q \to \mathbb{Z}_Q$ such that $f(x + q/2) = -f(x) \pmod{Q}$, the bootstrapping procedure $\mathsf{Boot}[f](\mathbf{a}, b)$ outputs a ciphertext $(\mathbf{c}, d) \in \mathbb{Z}_Q^{n+1}$ such that*

$$\mathsf{Dec}_{\mathbf{s}}(\mathbf{c}, d) = f(\mathsf{Dec}(\mathbf{a}, b)) + e \pmod{Q}$$

for some $|e| < \beta$, where β is a noise bound that depends only on the operations performed by Boot, but not on the input ciphertext $(\mathbf{a}, b)$.

For simplicity of presentation, we round β up to a power of 2.

3 Large-Precision Homomorphic Sign Evaluation

In this section we describe our main algorithms to homomorphically compute the sign of an encrypted value.

Let $(\mathbf{c}, d) \in \mathbb{Z}_Q^{n+1}$ be an LWE ciphertext with (large) ciphertext modulus Q and plaintext modulus Q/α. Specifically, assume $\mathsf{Dec}(\mathbf{c}, d) = \alpha m + e$, for some plaintext message $m \in \mathbb{Z}_{Q/\alpha}$ and noise bound $|e| < \beta \leq \alpha/2$. (Later we may set β to a bound strictly smaller than $\alpha/2$.) We assume that Q and α are powers of 2, so that the message m and the decryption $\mathsf{Dec}(\mathbf{c}, d)$ can both be interpreted as signed integers, in two's complement notation, and the sign of m is given by the MSB of m's binary representation. The goal is to homomorphically compute this sign bit.

By adding β to the ciphertext, the error $e + \beta$ becomes a positive value in the range $(0, 2\beta) \subseteq (0, \alpha)$. Hence the sign bit is also the same as the MSB of

$$m' = \mathsf{Dec}(\mathbf{c}, d + \beta) = \alpha m + (e + \beta).$$

$$
\begin{aligned}
f_0(x) &= \begin{cases} -q/4 & \text{if } 0 \le x < q/2 \\ +q/4 & \text{otherwise} \end{cases} \\
f_1(x) &= \begin{cases} x & \text{if } x < q/2 \\ q/2 - x & \text{otherwise} \end{cases} \\
f_2(x) &= \begin{cases} -q/4 & \text{if } 0 \le x < q/4 \\ +q/4 & \text{if } q/2 \le x < 3q/4 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 1: Negacyclic functions used by our homomorphic sign computation algorithms. The value of $f_1(x) = q/2 - x$ for $x \ge q/2$ is not relevant for our algorithms, and added here only to satisfy the negacylic constraint.

At this point, since we only care about the MSB of m', it does not matter which bits of m' are considered "message" bits and which are "noise" bits, and one may think of m' simply as an arbitrary integer modulo Q.

We compute the MSB of m' following the approach outlined in the introduction, using FHEW's functional bootstrapping algorithm Boot with a relatively small modulus q to *clear* the least significant bits of m' in small chunks, until only the MSB is left. We present two algorithms: the first algorithm requiring only two invocations of Boot per chunk, but under the assumption that $|e|$ is smaller than $\alpha/4$ and the second algorithm that works for ciphertexts with an arbitrary error e, but requires three invocations of Boot for each chunk. Although the approach based on two invocations of Boot is more efficient in practice for the large-precision sign evaluation, the approach with three invocations is more general and is of independent interest for evaluating the homomorphic floor function on arbitrary ciphertexts, e.g., noisy ciphertexts in the CKKS scheme.

In both algorithms, we instantiate the bootstrapping procedure as follows:

- We fix the modulus q to an appropriate value that can be efficiently supported by FHEW.
- We set the output modulus to Q by picking an internal modulus Q' larger than Q. (Usually, Q' is not a power of two, in order to support NTT.) We recall that the complexity of FHEW is linear in $\log Q'$, and exponential only in $\log q$. Hence one can use a relatively large Q'.
- We use Boot with one of three possible extraction functions f_0, f_1, f_2 shown in Figure 1. It can be easily checked that all three functions satisfy the negacyclic requirement (1).

3.1 Homomorphic Floor Function using Two Invocations of Boot

The core of the algorithm is a procedure $\mathsf{HomFloor}$ that on input a ciphertext $(\mathbf{c}, d) \in \mathbb{Z}_Q^{n+1}$ encrypting a message $m \in \mathbb{Z}_{Q/\alpha}$ with noise bounded by

$$|\mathsf{Dec}(\mathbf{c}, d) - \alpha \cdot m| < \beta \le \alpha/4$$

outputs another ciphertext $(\mathbf{c}', d') \in \mathbb{Z}_Q^{n+1}$ encrypting the floored message

$$r(m) = \left\lfloor \frac{\alpha}{q} \cdot m \right\rfloor \cdot \frac{q}{\alpha} \tag{2}$$

subject to the same noise bound β, i.e., such that $|\mathsf{Dec}(\mathbf{c}', d') - \alpha \cdot r(m)| < \beta$. Notice that this has precisely the same effect as zeroing the $\log_2(q/\alpha) = \log_2 q - \log_2 \alpha$ least significant bits of m. In particular, the MSB of m is the same as the MSB of $r(m)$.

The main algorithm HomSign uses the HomFloor subroutine to clear the least significant bits of the message until only the sign bit is left, as we describe next. Notice that after the application of HomFloor, the resulting ciphertext

$$\mathsf{Dec}(\mathbf{c}', d') = \alpha \cdot r(m) + e = q \cdot \tilde{m} + e \pmod{Q}$$

can be interpreted as an encryption of the message

$$\tilde{m} = \frac{\alpha}{q} \cdot r(m) = \left\lfloor \frac{\alpha}{q} \cdot m \right\rfloor \in \mathbb{Z}_{Q/q}$$

with noise $|e| < \beta$ much smaller than q. Since $r(m)$ is a multiple of q/α, the MSB of $\tilde{m}$ is the same as the MSB of $r(m)$ and m. So, we can switch to a smaller modulus $(\alpha/q) \cdot Q$ using Lemma 1 to obtain an encryption of $\tilde{m}$ with a scaling factor α, and repeat. After $\lceil (\log Q - \log q)/\log(q/\alpha) \rceil$ iterations, the modulus Q will be at most q, and the sign of the message can be computed directly using Boot.

The pseudocode of HomFloor and HomSign is given in Algorithm 1. In the rest of this subsection we analyze the correctness of the algorithm. We first analyze the correctness of HomFloor.

Algorithm 1 Algorithm for Homomorphic Sign Computation

procedure $\mathsf{HomFloor}(Q, (\mathbf{c}, d))$
  $d \leftarrow d + \beta$
  $(\mathbf{a}, b) \leftarrow (\mathbf{c}, d) \bmod q$
  $(\mathbf{c}, d) \leftarrow (\mathbf{c}, d) - \mathsf{Boot}[f_0](\mathbf{a}, b) \pmod{Q}$
  $d \leftarrow d + \beta - \frac{q}{4}$
  $(\mathbf{a}, b) \leftarrow (\mathbf{c}, d) \bmod q$
  $(\mathbf{c}, d) \leftarrow (\mathbf{c}, d) - \mathsf{Boot}[f_1](\mathbf{a}, b) \pmod{Q}$
  return $(\mathbf{c}, d)$
end procedure
procedure $\mathsf{HomSign}(Q, (\mathbf{c}, d))$
  while $Q > q$ **do**
    $(\mathbf{c}, d) \leftarrow \mathsf{HomFloor}(Q, (\mathbf{c}, d))$
    $(\mathbf{c}, d) \leftarrow \left\lceil \frac{\alpha}{q} \cdot (\mathbf{c}, d) \right\rfloor$
    $Q \leftarrow \alpha Q/q$
  end while
  $d \leftarrow d + \beta$
  $(\mathbf{a}, b) \leftarrow (q/Q) \cdot (\mathbf{c}, d)$
  $(\mathbf{c}, d) \leftarrow (-\mathsf{Boot}[f_0](\mathbf{a}, b)) \pmod{Q}$
  return $(\mathbf{c}, d)$
end procedure

Lemma 2. *For any Q, q, m and $\beta \leq \alpha/4$, the procedure* HomFloor *in Algorithm 1, on input a ciphertext $(\mathbf{c}, d) \in \mathbb{Z}_Q^{n+1}$ such that $|\mathsf{Dec}(\mathbf{c}, d) - \alpha \cdot m| < \beta$ outputs a ciphertext $(\mathbf{c}', d') \in \mathbb{Z}_Q^{n+1}$ such that $|\mathsf{Dec}(\mathbf{c}', d') - \alpha \cdot r(m)| < \beta$, where $r(x)$ is the rounding function defined in (2).*

Proof. Let $\mu = \mathsf{Dec}(\mathbf{c}, d) \in \mathbb{Z}_Q$ be the value encrypted by the input ciphertext $(\mathbf{c}, d)$. By assumption, $\mu = \alpha m + e$ for some $|e| < \beta$. We trace the value of μ and e through the execution of the algorithm. Adding β on line 2 makes the error positive $e \in (0, 2\beta)$. Line 3 computes an LWE ciphertext $(\mathbf{a}, b)$ that decrypts to $\mu' = \mathsf{Dec}(\mathbf{a}, b) = \mu \pmod q \in \mathbb{Z}_q$, that is, the $(\log_2 q)$ least significant bits of μ. Let $\tilde{m} = \lfloor \mu/q \rfloor = \lfloor (\alpha/q)m \rfloor$ be the remaining (most significant) bits, so that $\mu = \tilde{m} \cdot q + \mu'$.

Next, in order to analyze lines 4 and 5, we consider two cases, depending on the most significant bit of μ'. If the most significant bit of μ' is zero, then $\mathsf{Dec}(\mathsf{Boot}[f_0](\mathbf{a}, b)) = -q/4 + e_\beta$, where $|e_\beta| < \beta$. Subtracting $\mathsf{Boot}[f_0](\mathbf{a}, b)$ from $(\mathbf{c}, d)$ in line 4, and adjusting d in line 5, modifies μ by an additive term

$$-(-q/4 + e_\beta) + \beta - q/4 \in (0, 2\beta).$$

On the other hand, if the most significant bit of μ' is 1, then $\mathsf{Dec}(\mathsf{Boot}[f_0](\mathbf{a}, b)) = +q/4 + e_\beta$, and lines 4 and 5 modify μ by the additive term

$$-(q/4 + e_\beta) + \beta - q/4 = -q/2 + (0, 2\beta).$$

In either case, this clears the $(\log_2 q)th$ least significant bit of μ (corresponding to the most significant bit of μ') while increasing the error by at most 2β. Since the initial error is in $(0, 2\beta)$, the final error is in $(0, 4\beta) \subseteq (0, \alpha)$, and does not overflow into the most significant bits.

This shows that, even when accounting for the bootstrapping error, the value of $\mu = \mathsf{Dec}(\mathbf{c}, d)$ at line 6 has its $(\log_2 q)th$ least significant bit set to 0. In formulas, $\mu = q \cdot \tilde{m} + x$ for some $x = (\mu \bmod q) \in [0, q/2)$. The ciphertext $(\mathbf{a}, b)$ computed in line 6 encrypts this value x modulo q. Since $f_1(x) = x$ is the identity function for all $x \in [0, q/2)$, $\mathsf{Boot}[f_1]$ in line 7 returns an encryption of $x + e_\beta$. Subtracting this ciphertext from $(\mathbf{c}, d)$ on line 7, gives an encryption of

$$(q \cdot \tilde{m} + x) - (x + e_\beta) = q \cdot \tilde{m}x - e_\beta = \alpha \cdot r(m) - e_\beta$$

and hence

$$|\alpha \cdot r(m) - e_\beta - \alpha \cdot r(m)| < \beta,$$

as claimed in the lemma. □

The correctness of the main function $\mathsf{HomSign}$ easily follows, by repeatedly applying Lemma 2.

Theorem 2. *Let $\beta > 2$ be an upper bound on both the bootstrapping noise (from Theorem 1) and the size of the secret key*[1] *$\|\mathbf{s}\|_1 \leq \beta$. Let $\alpha \geq 4\beta$ be a power of 2. The procedure $\mathsf{HomSign}$ in Algorithm 1, on input an LWE ciphertext $(\mathbf{c}, d) \in \mathbb{Z}_Q^{n+1}$ encrypting a message $m \in \mathbb{Z}_{Q/\alpha}$ with error bounded by $|\mathsf{Dec}(\mathbf{c}, d) - \alpha \cdot m| < \beta$, computes an LWE encryption of the most significant bit of m, making at most $2\left\lfloor \frac{\log Q}{\log(q/\alpha)} \right\rfloor + 1$ calls to Boot.*

Proof. We need to show that the loop at lines 11-14 preserves the invariant that $(\mathbf{c}, d)$ encrypts a message with the correct MSB, and noise bounded by β. By Lemma 2, at each iteration, at line

[1]The weaker bound $\beta \geq O(\|\mathbf{s}\|_2) = O(\sqrt{n})$ suffices when using randomized modulus switching, or heuristically when assuming the input ciphertext is random. We use this weaker estimate for concrete parameters later in the paper.

12, HomFloor computes an encryption of a value of the form $\tilde{m}q + e$ with $|e| < \beta$, where $\tilde{m}$ has the correct MSB. Then, lines 13-14 switch the ciphertext modulus from Q to $(\alpha/q)Q$. By Lemma 1, the error of the resulting ciphertext is at most

$$(\alpha/q)\beta + (\beta+1)/2 \leq \beta/4 + \beta/2 + 1/2 < \beta,$$

taking into account the constraint $\beta > 2$. This proves the loop invariant. Upon exiting the loop, in line 15, the modulus has been reduced below $Q \leq q$, and the most significant bit of the message can be directly computed using Boot, using the fact that the sign function (f_0) is negacyclic. The multiplication by q/Q at line 17 is there only to ensure that Boot is always called with the same ciphertext modulus q. Alternatively, one may use a potentially smaller modulus $Q \leq q$ in the last call, which could be slightly faster. □

The final output of HomSign satisfies $\mathsf{Dec}(\mathbf{c}, d) = q/4 \pm \beta$ when the initial input encrypts a nonnegative number, and $\mathsf{Dec}(\mathbf{c}, d) = -q/4 \pm \beta$ when it encrypts a negative number. Sign computation algorithms with different output encodings are easily obtained by simply changing the function f_0 used in line 18. Likewise, the ciphertext modulus of the final output of HomSign can be set arbitrarily by simply changing the output modulus of the last invocation of Boot at line 18.

Remark 1. *Since the running time of* HomSign *is proportional to* $\log Q \,/\, \log(q/\alpha)$*, it is always best to set* α *to the smallest possible value* $\alpha = 4\beta$*. So, given values for* Q *(from the input specification) and* q, β *from Theorem 1 (typically based on security and efficiency considerations), the running time of* HomSign *is essentially that of* $2\left\lfloor \frac{\log Q}{\log q - \log \beta - 2} \right\rfloor + 1$ *invocations of* Boot *or, equivalently,* $\left\lfloor \frac{\log Q}{\log q - \log \beta - 2} \right\rfloor$ *invocations of* HomFloor *+ 1 invocation of* Boot*.*

3.2 Homomorphic Floor Function for Arbitrary Ciphertexts using Three Invocations of Boot

We also propose an alternative floor function evaluation algorithm that works for arbitrary ciphertexts. This algorithm requires three invocations of Boot but makes no assumption on the size of the input error. Although this approach is typically less efficient than HomFloor when used as a subroutine in HomSign (as shown later in Section 6.1), it has some advantages when applied directly to an arbitrary ciphertext. For instance, when the message and noise are not separable, as in the CKKS scheme, the use of this procedure avoids calling a prior modulus switching operation, which may accidentally change the sign of encrypted values close to zero. When used as a subroutine for HomSign, the alternative floor function procedure allows us to replace $\alpha = 4\beta$ with $\alpha = 2\beta$, hence gaining one extra bit of precision in each floor function iteration at the expense of one extra invocation of Boot.

Lemma 3. *Let* β *be the bootstrapping noise from Theorem 1, and assume* $q \geq 16\beta$*. The procedure* HomFloorAlt *in Algorithm 2, on input a ciphertext* $(\mathbf{c}, d) \in \mathbb{Z}_Q^{n+1}$ *with* $\mathsf{Dec}(\mathbf{c}, d) = m \in \mathbb{Z}_Q$*, outputs a ciphertext* $(\mathbf{c}', d') \in \mathbb{Z}_Q^{n+1}$ *with* $\mathsf{Dec}(\mathbf{c}, d) = \tilde{m}q + e \in \mathbb{Z}_Q$ *for* $\tilde{m} = \lfloor m/q \rfloor$ *and some* $|e| < \beta$*.*

Proof. The ciphertext $(\mathbf{a}, b)$ computed in line 2 decrypts to $m' = \mathsf{Dec}(\mathbf{a}, b) = m \bmod q$, the $\log_2 q$ least significant digits of m. Let x be the two most significant bits of m'. Function f_2 only works on these two bits, mapping $00 \mapsto 11$, $10 \mapsto 01$, and $01, 11 \mapsto 00$. When $f_2(m')$ is subtracted from $(\mathbf{c}, d)$

Algorithm 2 Alternative Algorithm for Homomorphic Sign Computation

1: **procedure** $\mathsf{HomFloorAlt}(Q, (\mathbf{c}, d))$
2: $(\mathbf{a}, b) \leftarrow (\mathbf{c}, d) \bmod q$
3: $(\mathbf{c}, d) \leftarrow (\mathbf{c}, d) - \mathsf{Boot}[f_2](\mathbf{a}, b) \pmod{Q}$
4: $d \leftarrow d + \beta - \frac{q}{4}$
5: $(\mathbf{a}, b) \leftarrow (\mathbf{c}, d) \bmod q$
6: $(\mathbf{c}, d) \leftarrow (\mathbf{c}, d) - \mathsf{Boot}[f_0](\mathbf{a}, b) \pmod{Q}$
7: $d \leftarrow d + \beta - \frac{q}{4}$
8: $(\mathbf{a}, b) \leftarrow (\mathbf{c}, d) \bmod q$
9: $(\mathbf{c}, d) \leftarrow (\mathbf{c}, d) - \mathsf{Boot}[f_1](\mathbf{a}, b) \pmod{Q}$
10: **return** $(\mathbf{c}, d)$
11: **end procedure**

in line 3, the corresponding bits of m are mapped either to 11 (when $x = 11$) or to 01 (otherwise). In particular, the second bit is always one. Subtracting $q/4$ from d on line 4 makes this bit always zero. Adding β in line 4 also ensures that the bootstrapping error added by Boot is positive, in the range $(0, 2\beta)$. At this point (line 5) we have a ciphertext such that $\mathsf{Dec}(\mathbf{c}, d) = \tilde{m} \cdot q + b \cdot (q/2) + x + e$ for some (unknown) bit $\tilde{b} \in \{0, 1\}$, positive integer $x \in [0, q/4)$ and positive bootstrapping error $e \in (0, 2\beta)$. Similarly, we have $\mathsf{Dec}(\mathbf{a}, b) = \tilde{b}(q/2) + x + e$. Assuming $q \geq 8\beta$, adding e to $\tilde{b}(q/2)$ does not change the bit $\tilde{b}$. So, $f_0(\tilde{b}(q/2) + x + e) = -q/4$ when $\tilde{b} = 0$ and $+q/4$ when $b = 1$. Similarly to Lemma 2, subtracting $\mathsf{Boot}[f_0](\mathbf{a}, b)$ from $(\mathbf{c}, d)$ in line 6 and adjusting the value of d in line 7 has the effect of clearing the bit $\tilde{b}$, while adding a positive bootstrapping error $e \in (0, 2\beta)$.

This shows that, at line 8, we have $\mathsf{Dec}(\mathbf{c}, d) = \tilde{m}q + x + e + e'$ where $\mathsf{Dec}(\mathbf{a}, b) = x + e + e' \in (0, q/4 + 4\beta)$. Assuming $q \geq 16\beta$, we have $x + e + e' < q/4 + 4\beta \leq q/2$. So, $f_1(x + e + e') = x + e + e'$, and subtracting $\mathsf{Boot}[f_1](\mathbf{a}, b)$ from $(\mathbf{c}, d)$ in line 9, gives a ciphertext such that $\mathsf{Dec}(\mathbf{c}, d) = \tilde{m}q \pm \beta$. □

The $\mathsf{HomFloorAlt}$ algorithm can be used to homomorphically compute the sign of a ciphertext using essentially the same process as $\mathsf{HomSign}$. We only need to choose an approximate value of α, and replace the call to $\mathsf{HomFloor}(Q, (\mathbf{c}, d))$ with the call to $\mathsf{HomFloorAlt}(Q, (\mathbf{c}, d + \alpha/2))$ to ensure that the noise is positive, so it does not alter the most significant bit of the message.

By Lemma 3, the ciphertext computed by $\mathsf{HomFloorAlt}$ has noise at most β. So, by Lemma 1, switching the modulus to $(\alpha/q)Q$ increases the error to $(\alpha/q)\beta + \beta''$, where β'' is the modulus switching noise. For correctness, we need this error to be bounded by $\alpha/2$. This condition holds when

$$\frac{\beta}{q} + \frac{\beta''}{\alpha} \leq \frac{1}{2}.$$

Setting $q = 16\beta$, this is equivalent to $\alpha \geq (16/7)\beta''$.

In summary, the $\mathsf{HomSign}$ algorithm based on the $\mathsf{HomFloorAlt}$ procedure proposed in this section makes a total of

$$1 + 3\left\lfloor \frac{\log Q}{\log q + \log_2 7 - 4 - \log \beta''} \right\rfloor \approx 3\frac{\log Q}{\log q - \log \beta''}$$

calls to Boot.

4 From Floor Function to Arbitrary Function Evaluation

As discussed, the FHEW functional bootstrapping requires the evaluated functions to be negacyclic. However, this greatly restricts the power of functional bootstrapping. In this section, we show how to extend our main idea of $\mathsf{HomFloor}$ to functional bootstrapping of arbitrary functions.

Let us first formally define the problem. Given a ciphertext $(\mathbf{c}, d)$ with modulus q encrypting a digit $m \in \mathbb{Z}_{q/\alpha}$, and an arbitrary function $f : \mathbb{Z}_{q/\alpha} \to \mathbb{Z}_{Q/\alpha}$, we want to obtain a ciphertext $(\mathbf{c}', d') \in \mathbb{Z}_Q^{n+1}$ such that $\lceil \mathsf{Dec}(\mathbf{c}', d')/\alpha \rfloor = f(m)$.

At a high level, we proceed as follows: first, we use modulus switching to raise the ciphertext modulus from q to $2q$. This process (randomly) maps an encrypted value $m \in \mathbb{Z}_{q/\alpha}$ to either $m \in \mathbb{Z}_{2q/\alpha}$ or $m + q/\alpha \in \mathbb{Z}_{2q/\alpha}$. The main purpose of this step is to double the size of the message space by introducing an extra (most significant) bit.

Next, similar to $\mathsf{HomFloor}$, we first use an extraction function $f_0'(x)$ (similar to f_0 in Fig. 1) to remove the MSB of the (modulus-raised) encrypted plaintext $m \in \mathbb{Z}_{2q/\alpha}$, i.e., for plaintext $m \in \mathbb{Z}_{2q/\alpha}$ we homomorphically evaluate f_0' to obtain an encrypted value $m' = m \pmod{q/\alpha} \in \mathbb{Z}_{2q/\alpha}$. This is the same as the original message m, but as an element of a larger message space.

Then, we create a new function $f_1' : \mathbb{Z}_{2q} \to \mathbb{Z}_Q$ by setting

- $f_1'(x) = \alpha \cdot f(\lceil x/\alpha \rfloor)$ to the function we want to compute for $x < q$, and
- $f_1'(x) = -\alpha \cdot f(\lceil (2q - x)/\alpha \rfloor)$ for $x \geq q$ to satisfy the negacyclic requirement.

We evaluate this function via functional bootstrapping to obtain a ciphertext $(\mathbf{c}', d')$ such that $\lceil \mathsf{Dec}(\mathbf{c}', d')/\alpha \rfloor = f(m')$.

The resulting procedure for arbitrary function evaluation is listed in Algorithm 3.

Algorithm 3 Algorithm for Arbitrary Function Evaluation

Auxiliary math functions $f_0 \colon \mathbb{Z}_{2q} \to \mathbb{Z}_{2q}$

$$f_0'(x) = \left(q \left\lfloor \frac{x}{q} \right\rfloor - \frac{q}{2} \right) \bmod 2q$$

procedure EvalFunc($f : \mathbb{Z}_{q/\alpha} \to \mathbb{Z}_{Q/\alpha}, q, Q, \alpha, (\mathbf{c}, d)$)
  Let

$$f_1'(x) = \begin{cases} \alpha f(\lceil x/\alpha \rfloor) & \text{if } x < q \\ -\alpha f(\lceil (2q - x)/\alpha \rfloor) & \text{otherwise} \end{cases} \bmod Q$$

  $d \leftarrow d + \beta$
  $(\mathbf{c}, d) \leftarrow (\mathbf{c}, d) \pmod{2q}$
  $(\mathbf{c}, d) \leftarrow (\mathbf{c}, d) - \mathsf{Boot}[f_0'](\mathbf{c}, d) \pmod{2q}$
  $d \leftarrow d + \beta - \frac{q}{2}$
  $(\mathbf{c}, d) \leftarrow \mathsf{Boot}[f_1'](\mathbf{c}, d) \pmod{Q}$
  return $(\mathbf{c}, d)$
end procedure

Note that if the function $f(x)$ is periodic (i.e., $f(x) = f(x + q/2 \pmod q)$ for all $x \in \mathbb{Z}_q$), the extension to $\mathbb{Z}_{2q}$ is not needed and we can replace all instances of q with $q/2$ in Algorithm 3. This

gains one extra bit of precision for periodic functions, as compared to arbitrary functions.

For Algorithm 3, we can formulate the following theorem.

Theorem 3. *For any Q, q, m and $\beta \leq \alpha/4$, the procedure* EvalFunc *in Algorithm 3, on input a ciphertext $(\mathbf{c}, d) \in \mathbb{Z}_q^{n+1}$ such that $|\mathsf{Dec}(\mathbf{c}, d) - \alpha \cdot m| < \beta$ and an arbitrary function $f : \mathbb{Z}_{q/\alpha} \to \mathbb{Z}_{Q/\alpha}$, outputs a ciphertext $(\mathbf{c}', d') \in \mathbb{Z}_Q^{n+1}$ such that $|\mathsf{Dec}(\mathbf{c}', d') - \alpha \cdot f(m)| < \beta$.*

Proof. We prove the theorem by tracing the value encrypted by the input ciphertexts $(\mathbf{c}, d)$. By assumption, $\mathsf{Dec}(\mathbf{c}, d) = \alpha m + e$ for some $|e| < \beta$. Adding β on line 3 makes the error positive $e \in (0, 2\beta)$. Line 4 raises the ciphertext's modulus to $2q$ and thus we (randomly) obtain one of the following: $\mu = \mathsf{Dec}(\mathbf{c}, d) = \alpha m + e \in \mathbb{Z}_{2q}$ or $\mu = \mathsf{Dec}(\mathbf{c}, d) = \alpha m + e + q \in \mathbb{Z}_{2q}$. Then, line 5 executes $\mathsf{Boot}[f_0']$, and line 6 shifts the result by subtracting $q/2$. Based on a similar argument as in the proof of Lemma 2, these two lines together clear the MSB of μ (i.e., now $\mathsf{Dec}(\mathbf{c}, d) = \alpha m + e \in \mathbb{Z}_{2q}$) while increasing the error by at most β, and hence the updated encrypted value is $\mu = \mathsf{Dec}(\mathbf{c}, d) \in [0, q)$. Finally, line 7 executes $\mathsf{Boot}[f_1']$ and we obtain $\mathsf{Dec}(\mathbf{c}, d) = \alpha f(m) + e \in \mathbb{Z}_Q$ with $|e| < \beta$, and therefore, the resulted $(\mathbf{c}, d)$ encrypts a plaintext $m' = \lceil \mathsf{Dec}(\mathbf{c}, d)/\alpha \rfloor = f(m)$ where m is the input plaintext as we required. □

An alternative arbitrary function evaluation can be trivially derived based on HomFloorAlt using the same steps as described here. As the efficiency of this alternative algorithm is worse, we do not discuss it in the paper.

Note that our general bootstrapping algorithm works efficiently in practice only for small plaintext moduli p because the FHEW bootstrapping becomes prohibitively expensive as the plaintext modulus is increased (more than doubles for each extra bit of precision). However, we can extend it to larger plaintext moduli using the procedure for homomorphic digit decomposition described in the next section.

5 Homomorphic Digit Decomposition

The high-level idea of homomorphic digit decomposition is to decompose an LWE ciphertext with a large plaintext (ciphertext) modulus into a vector of LWE ciphertexts with small plaintext (ciphertext) moduli, corresponding to the digit sizes. In this section we extend our sign evaluation algorithm in Section 3 to achieve homomorphic digit decomposition.

As pointed out in Section 4, one useful application of such digit decomposition is the evaluation of functions over large-precision ciphertexts using lookup tables, i.e., the evaluation of arbitrary functions for large plaintext moduli. Two methods for evaluating a Look-Up Table (LUT) using (a vector of) LWE ciphertexts for each digit are presented in [22]. The first (more general) approach uses tree evaluation while the second (more special-purpose) approach is based on chaining. These methods allow breaking down a large LUT into small LUTs, each of which corresponds to a decomposed digit of the original ciphertext encrypting a large number. These small LUTs can be completely different from each other. In summary, the evaluation of an arbitrary function over a large plaintext space gets expressed as LUT evaluations over encrypted digits.

The LWE ciphertexts for each digit can be "extracted" from a large-precision LWE ciphertext using the homomorphic digit decomposition algorithm presented in this section, and then the general bootstrapping procedure from Section 4 can be used to evaluate for each digit arbitrary functions/lookup tables over small plaintext moduli. In other words, the digit decomposition

procedure presented in this section and small-LUT evaluation procedure presented in Section 4 are two core subroutines in arbitrary function evaluation for larger plaintext spaces.

5.1 Digit Decomposition into Fixed-Size Digits

We first assume for simplicity that all output ciphertexts have the same modulus q and $\log(Q/\alpha)$ divides $\log(q/\alpha)$. Let us formally define the problem. Given an input LWE $(\mathbf{c}, d) \in \mathbb{Z}_Q^{n+1}$ encrypting a message $m \in \mathbb{Z}_{Q/\alpha}$, our goal is to obtain a vector of ciphertexts $((\mathbf{c}_i, d_i) \in \mathbb{Z}_q^{n+1})_{i\in[k]}$, where $k = \frac{\log(Q/\alpha)}{\log(q/\alpha)}$, such that each ciphertext $(\mathbf{c}_i, d_i)$ encrypts a digit $m_i \in \mathbb{Z}_{q/\alpha}$ and $m = \sum_{i=1}^{k} m_i \cdot (q/\alpha)^{i-1}$.

Let $\alpha = 4\beta$ and the input ciphertext $(\mathbf{c}, d) \in \mathbb{Z}_Q^{n+1}$ have noise $< \beta$. Then we can perform digit decomposition using Algorithm 4. The high-level idea is to extract each least significant digit, remove it using $\mathsf{HomFloor}$, and then use the modulus switching procedure to reduce the modulus from Q to $\alpha Q/q$, hence moving to the next least significant digit.

Theorem 4. *Let $\beta > 2$ be an upper bound on both the bootstrapping noise (from Theorem 1) and the size of the secret key*[2] *$\|\mathbf{s}\|_1 \le \beta$. Let $\alpha \ge 4\beta$ be a power of 2. The procedure* $\mathsf{DigitDecomp}$ *in Algorithm 4, on input an LWE ciphertext $(\mathbf{c}, d) \in \mathbb{Z}_Q^{n+1}$ encrypting a message $m \in \mathbb{Z}_{Q/\alpha}$ with error bounded by $|\mathsf{Dec}(\mathbf{c}, d) - \alpha \cdot m| < \beta$, outputs ciphertexts $((\mathbf{c}_i, d_i))_{i\in[k]}$ such that $m = \sum_{i=1}^{k} m_i \cdot (q/\alpha)^i$, where $m_i = \lceil \mathsf{Dec}(\mathbf{c}_i, d_i)/\alpha \rfloor$, $k = \frac{\log(Q/\alpha)}{\log(q/\alpha)}$, and $|\mathsf{Dec}(\mathbf{c}_i, d_i) - \alpha \cdot m_i| < \beta$.*

Proof. By the correctness of $\mathsf{HomFloor}$ shown in Lemma 2, we directly see that $m = \sum_{i=0}^{k} m_i \cdot (q/\alpha)^i$, where $m_i = \lceil \mathsf{Dec}(\mathbf{c_i}, d_i)/\alpha \rfloor$. The first ciphertext $(\mathbf{c}_1, d_1)$ in the vector has the same noise as the input ciphertext, i.e., at most β. Then, for $(\mathbf{c}_i, d_i)$, where $i \in [2, k]$, we have the same noise as for input ciphertexts of $\mathsf{HomFloor}$, again at most β, which follows from the proof of Theorem 2. □

Alternatively, we can formulate a digit decomposition algorithm based on $\mathsf{HomFloorAlt}$ by trivially replacing $\mathsf{HomFloor}$ with $\mathsf{HomFloorAlt}$ and changing α from 4β to 2β.

Algorithm 4 Algorithm for Homomorphic Digit Decomposition based on $\mathsf{HomFloor}$

procedure DIGITDECOMP$(Q, q, (\mathbf{c}, d))$
  $k \leftarrow 1$
  while $Q > q$ **do**
    $(\mathbf{c}_k, d_k) \leftarrow (\mathbf{c}, d) \pmod q$
    $(\mathbf{c}, d) \leftarrow \mathsf{HomFloor}(Q, q, (\mathbf{c}, d))$
    $(\mathbf{c}, d) \leftarrow \left\lceil \frac{\alpha}{q} \cdot (\mathbf{c}, d) \right\rfloor$
    $Q \leftarrow \alpha Q/q$
    $k \leftarrow k + 1$
  end while
  $(\mathbf{c}_k, d_k) \leftarrow (\mathbf{c}, d)$
  return $\{(\mathbf{c}_i, d_i)\}_{i\in[k]}$
end procedure

[2]The weaker bound $\beta \ge O(\|\mathbf{s}\|_2) = O(\sqrt{n})$ suffices when using randomized modulus switching, or heuristically when assuming the input ciphertext is random.

5.2 Digit Decomposition into Varying-Size Digits

In some scenarios, it is desired to decompose a large-message LWE ciphertext into a vector of LWE ciphertexts with different digit sizes, where each digit size is a power of two. Our algorithm can also be extended to this more general case.

Let us first formally define the problem. Given an input LWE ciphertext $(\mathbf{c}, d) \in \mathbb{Z}_Q^{n+1}$, encrypting a message $m \in \mathbb{Z}_{Q/\alpha}$, our goal is to output a vector of ciphertexts $((\mathbf{c}_i, d_i) \in \mathbb{Z}_{q_i}^{n+1})_{i\in[k]}$, where k denotes the vector size and $(\prod_{i=1}^{k} \frac{q_i}{\alpha}) = \frac{Q}{\alpha}$, such that each ciphertext encrypts a digit $m_i \in \mathbb{Z}_{q_i/\alpha}$ and $m = m_1 + \sum_{i=2}^{k} m_i \cdot (\prod_{j=1}^{i-1} \frac{q_j}{\alpha})$.

This can be achieved by making small modifications in Algorithm 4. Instead of evaluating $\mathsf{DigitDecomp}$ with modulus q in every iteration, we use q_i in the i^{th} iteration, and replace $\left\lceil \frac{\alpha}{q} \cdot (\mathbf{c}, d) \right\rfloor$ with $\left\lceil \frac{\alpha}{q_i} \cdot (\mathbf{c}, d) \right\rfloor$.

Note that the computational complexity of varying-size digit decomposition depends on the value of each q_i as different values of N and potentially other parameters may be needed for a given value of q_i.

6 Parameter Selection and Optimizations

The proposed algorithms work with the following parameters:

- q, small (power-of-two) (LWE) modulus;
- n, lattice parameter for the LWE scheme;
- Q', RLWE/RGSW modulus (used for NTTs);
- Q, input (power-of-two) modulus;
- Q_{ks}, LWE/RLWE modulus used for key switching;
- N, ring dimension for RLWE/RGSW;
- B_g, gadget base for digit decomposition in each accumulator update, which breaks integers $\bmod Q$ into d_g digits;
- B_{ks}, gadget base for key switching, which breaks integers $\bmod Q$ into d_{ks} digits;

6.1 Selecting the Floor Function Evaluation Method

There are two options for evaluating the floor function: $\mathsf{HomFloor}$ and $\mathsf{HomFloorAlt}$. Given a ciphertext modulus q, noise bound β, and small plaintext modulus p, $\mathsf{HomFloor}$ can support $p \leq q/\alpha$ where $\alpha \geq 4\beta$ with two bootstrapping operations while $\mathsf{HomFloorAlt}$ can support the plaintext modulus of $2p$ with three bootstrapping operations. Hence $\mathsf{HomFloorAlt}$ is about 1.5x slower but can process 1 extra bit. If we denote as P the desired (large) plaintext space for sign evaluation (i.e., $P = Q/\alpha$, where Q is the (large) modulus of the input ciphertext), then evaluating $\mathsf{HomSign}$ using $\mathsf{HomFloor}$ requires $1+2\left\lfloor \frac{\log P}{\log p} \right\rfloor$ bootstrapping operations and evaluating $\mathsf{HomSign}$ using $\mathsf{HomFloorAlt}$ requires $1 + 3\left\lfloor \frac{\log P}{\log p+1} \right\rfloor$ bootstrapping operations.

It is easy to see that for $p = 2$, using HomFloorAlt is faster by a factor of about 4/3. For $p = 2^2 = 4$, the number of bootstrapping operations is roughly the same, and for higher values of p using HomFloor is faster. In practice, the value of p is at least $2^3 = 8$ (or actually $2^4 = 16$ for the optimized setting described in Section 6.3), and, therefore, HomFloor is always the preferred floor function evaluation algorithm in practice.

6.2 Module-LWE vs RLWE

As an alternative to RLWE in the bootstrapping procedure described in Theorem 1, we consider a module-LWE accumulator instead of the RLWE one. In this case, we can replace one ring element of dimension N with w ring elements, each with dimension N/w for some $w \in \mathbb{Z}^+$. Therefore, we use w NTT operations for the ring dimension N/w to replace one NTT operation for the ring dimension N. This can give a speed-up of roughly $\log N/(\log N - \log w)$. However, since $q = 2N$, we would lose one bit as w is doubled, i.e., $\log w$ bits in total. If we have $1+2\lfloor\frac{\log P}{\log p}\rfloor$ bootstrapping operations for RLWE, then we will have $\frac{\log N}{\log N - \log w}\left(1 + 2\lfloor\frac{\log P}{\log p - \log w}\rfloor\right)$ as a complexity for Module-LWE in terms of equivalent bootstrapping operations.

For the practical values of N (at least 1024) and p (8 or 16), it can be easily shown that RLWE is always faster than Module-LWE for any $w > 1$. Therefore, RLWE is always preferred in practice.

6.3 Optimizations

Throughout the paper, so far, we have used the worst-case error bound of 4β. This was done primarily for simplicity so we could work with a power-of-two β. In the actual implementation, we can use an average-case error estimate. We consider this as an implementation-level optimization.

If each ciphertext has an error bound β, adding two ciphertexts with errors sampled independently from each other will result in an error bound of $2\sqrt{2}\beta$, which can be easily shown using subgaussian analysis/Central Limit Theorem arguments, and was confirmed experimentally.

Such optimization can end up in an even tighter noise bound in practice (essentially going from $2\sqrt{2}\beta$ to 2β). Our experimental results (based on 1,000 runs) suggest that a single ciphertext after bootstrapping has a standard deviation $\sigma \approx 11.5$. If we set the probability of error to less than 2^{-32}, then the estimated β is 73, which rounds up to 128. When two independent ciphertexts are added together, we get a noise with standard deviation $\sigma \approx 16.3$, and for the same probability the estimated bound β is 103, which also rounds up to 128.

Therefore, in practice, we can remove the second addition of β in HomFloor (at line 5 of Algorithm 1). The same optimization can be applied to HomFloorAlt, DigitDecomp, and EvalFunc.

6.4 Setting the Parameters

For HomSign and DigitDecomp, the main input parameter is Q. Typically $\log Q$ should be set to $\log P + \log(\tilde{\beta} + \beta) + 1$, where $\log P$ is precision in bits of the input plaintext, $\tilde{\beta}$ is the error in the input ciphertext, and β is the FHEW bootstrapping error bound defined in Theorem 1. It is recommended to perform modulus switching to obtain the smallest acceptable value of Q before running the procedures.

After Q is fixed, one needs to find a prime number $Q' > Q$ to support the NTT operations during bootstrapping. Based on the desired security level, we can fix the ring dimension N using the HE standard [4] or LWE estimator [5]. For example, for a ring of dimension $N = 2048$, for 128-bit

security against classical computer attacks, we can set $\log Q'$ to at most 54 bits; for 256-bit security, we can support at most 29 bits. With N fixed, we choose $q = 2N$ for maximum performance.

Together with Q', we need to choose B_g, which is the gadget base to decompose Q'. For best performance, we generally set B_g to the smallest power-of-two $> \sqrt{Q'}$, i.e., $d_g = 2$. B_g is the main parameter that determines the noise growth. Roughly speaking, we need $\frac{Q \cdot B_g}{Q'} \ll 1$. For best runtime performance, $B_g = \lceil \sqrt{Q'} \rceil$, we need $\frac{Q}{\sqrt{Q'}} \ll 1$. If we have $B_g = \lceil Q'^{1/3} \rceil$ ($d_g = 3$), we get a slowdown of 3/4, but then we can support larger Q as the requirement is then $\frac{Q}{Q'^{2/3}} \ll 1$. According to our experiments, roughly $\frac{Q \cdot B_g}{Q'} \approx 2^{-11}$ should be sufficient to achieve the noise standard deviation of ≈ 11.5 after one bootstrapping (which is enough to maintain a failure probability $< 2^{-32}$ with error bound 128, because adding two bootstrapped ciphertexts would result in a noise standard deviation ≈ 16.3).

The last remaining parameter is p, which is the small plaintext modulus for each digit in HomSign and Decomp, i.e., the internal plaintext modulus in HomFloor. We have $p = q/(4\beta)$ as the worst-case bound in our algorithms. However, the optimizations in Section 6.3 allow us to use $p = q/(2\beta)$ in the implementation.

6.5 Noise Estimates

Bootstrapping results in a ciphertext with an error from a Gaussian distribution of standard deviation $\sigma = \sqrt{\frac{q^2}{Q_{ks}^2}(\frac{Q_{ks}^2}{Q'^2}\sigma_{ACC}^2 + \sigma_{MS_1}^2 + \sigma_{KS}^2) + \sigma_{MS_2}^2}$, where $\sigma_{MS_1}^2 = \frac{|s_N|^2+1}{3}$, $\sigma_{MS_2}^2 = \frac{|s_n|^2+1}{3}$, $\sigma_{ACC}^2 = 4d_g n N \frac{B_g^2}{6}\sigma_{BK}^2$, and $\sigma_{KS}^2 = \sigma_{BK} N d_{ks}$ for a uniform ternary secret keys s_N with dimension N and s_n with dimension n, as estimated in [29]. Note that here we use a heuristic (average-case) estimate for σ_{MS}^2.

To guarantee that we can have a failure probability $< 2^{-32}$ as proposed in [16, 19, 29], we set $\beta \approx 6.37\sigma$, and we then round β to the smallest power-of-two greater than 6.37σ. However, sometimes $\sqrt{2} \cdot 6.37\sigma$ is also smaller than the rounded β. Therefore, we can use the same β even if we have a $\sqrt{2}$ loss in Algorithms 1 and 4.

6.6 Computational Complexity

For our experiments, we used the TFHE/GINX bootstrapping method with ternary secret keys [29]. Each bootstrapping takes roughly $2n(d_g + 1)$ NTT operations (we employed the ternary CMUX optimization recently proposed by Bonte et. al [6]) and each NTT operation is $O(N \log N)$.

7 Implementation and Performance Evaluation

7.1 Parameters Used for Our Implementation

In our implementation, we limited Q to at most 2^{29}, which supports up to 21 bits of precision. This precision is sufficient for most applications. One common use of FHEW-based comparisons is in applications that use the CKKS scheme for all polynomial computations, and then switch to FHEW for comparison-based computations [25]. The precision typically achieved in these applications is not higher than 20 bits (as it is limited primarily by the precision of CKKS bootstrapping [7]).

Once Q is fixed, we need to find Q' such that $Q/Q'^{\frac{d_g-1}{d_g}} \ll 1$, as explained in Section 6.4. We set $\log Q'$ to 54, which is the largest modulus size that supports 128-bit security for $N = 2048$ [4].

Next, we need to choose B_g. For $Q' < 2^{54}$, there are three main practical options: $B_g = 2^{27}$ (two digits in RGSW gadget decomposition, i.e., $d_g = 2$), $B_g = 2^{18}$ ($d_g = 3$), and $B_g = 2^{14}$ ($d_g = 4$). For $Q \leq 2^{16}$, we can use $B_g = 2^{27}$ (fastest bootstrapping). For $2^{16} < Q \leq 2^{25}$, we use $B_g = 2^{18}$. For $2^{25} < Q \leq 2^{29}$, we use $B_g = 2^{14}$.

Note that we can dynamically change from $B_g = 2^{14}$ to $B_g = 2^{18}$ and then to $B_g = 2^{27}$ as the value of Q gets progressively reduced via HomFloor iterations in HomSign and DigitDecomp, resulting in a speed-up of later bootstrapping operations. When using this dynamic mode, a bootstrapping key for each value of B_g should be generated and loaded in computer memory. Hence, there is a tradeoff between runtime and storage. One can either use the smallest B_g for all bootstrapping operations and the smallest storage for the bootstrapping key or use multiple values of B_g, improving the runtime of later bootstrapping operations at the expense of increased storage requirements.

We use $n = 1305$, $\sigma_{BK} = 3.19$, $Q_{ks} = 2^{35}$, and $B_{ks} = 32$, where σ_{BK} is the standard deviation of the noise to encrypt the bootstrapping keys. All other parameters are set to the same values as in [29].

For the parameters above, the estimated standard deviation σ of a bootstrapped ciphertext is about 11.5 (based on 1,000 bootstrapping runs). For a sum of two bootstrapped ciphertexts, the standard deviation σ_{sum} is about 16.3. We can use this value of σ_{sum} to select the value of plaintext modulus p. The failure probability is given by $1 - \mathsf{erf}(\frac{q/p}{2\sqrt{2}\sigma_{sum}})$. To guarantee the probability of success for HomSign to be at least $1 - 2^{32}$, similar to [16, 19, 29], we set $p = 16 = 2^4$. For this value of p, the error upper bound β is 128. This implies we can achieve 4 bits of precision in the HomFloor function, i.e., we can work with digits of up to 4 bits per iteration when dealing with large-precision LWE ciphertexts.

Remark 2. *Although we restricted Q to 2^{29} and $\log Q'$ to 54 bits, higher values of both Q and Q' can be supported. For Q' larger than 64 bits, the machine word size for many modern computing environments, a Residue Number System (RNS) variant of* RLWE *and the corresponding RNS digit decomposition can be instantiated using the lattice gadget techniques presented in [21].*

7.2 Software Implementation

We implemented HomSign, DigitDecomp, and EvalFunc in PALISADE v1.11.6 [1]. The evaluation environment was a commodity desktop computer system with an Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz and 64 GB of RAM, running Ubuntu 18.04 LTS. The C++ compiler was g++ 10.1.0. We compiled PALISADE with the following CMake flag: WITH_NATIVEOPT=ON (machine-specific optimizations were applied by the compiler).

7.3 Experimental Results

For Q bounded to 2^{29} and the parameter values discussed in Section 7.1, the runtime of HomSign and DigitDecomp can be described in terms of bootstrapping times for $d_g = 2$, $d_g = 3$, and $d_g = 4$. For $Q \leq 2^{16}$ we use $d_g = 2$, for $2^{16} < Q \leq 2^{25}$ we use $d_g = 3$, and for $2^{25} < Q \leq 2^{29}$ we use $d_g = 4$.

The single-threaded runtimes for $d_g = 2$, $d_g = 3$, and $d_g = 4$ in our evaluation environment were 442, 600, and 785 ms, respectively. The runtimes for HomSign, DigitDecomp, and EvalFunc are listed

in Table 2. When $\log P = 4$, only one bootstrapping invocation is needed. Then for each next 4 bits (each digit), two more bootstrapping invocations are needed, as explained in Section 6.1. Note that although for $Q = 2^{25}$ and $Q = 2^{26}$, the number of bootstrapping operations is the same (four calls to HomFloor, each with two bootstrapping invocations, plus one extra bootstrapping), the runtimes are different because for $Q = 2^{25}$ we have three bootstrapping operations at $d_g = 2$ and six bootstrapping operations at $d_g = 2$, while for $Q = 2^{26}$ we have three bootstrapping operations at $d_g = 2$, four bootstrapping operations at $d_g = 3$, and two more bootstrapping operations at $d_g = 4$. Moreover, note that for $Q = 2^{28}$ and $Q = 2^{29}$, there is a relatively large runtime gap. This is because we need one more call to HomFloor for $Q = 2^{29}$ and therefore two additional bootstrapping invocations. In general, the runtime is roughly linear in $\log Q$. For arbitrary function evaluation, we can process one bit less compared to the HomFloor function in HomSign.

For EvalFunc, we used the function $y = x^3$, but any other function over modulus P could be used instead, and we verified this experimentally. As explained, EvalFunc with $P = 2^3$ can be used as a subroutine to support arbitrary function (LUT) evaluation for larger plaintext moduli; this LUT evaluation is achieved using a combination of either tree or chain method introduced in [22] together with the digit decomposition method proposed in Section 5. One can also increase P for a single "digit" by increasing the ring dimension (each extra bit of P requires doubling the ring dimension, i.e., roughly doubling the runtime). We chose specifically $\log P = 3$ for EvalFunc to illustrate the runtime for arbitrary functions as this setting corresponds to the ring dimension $N = 2048$, which was used for all proposed capabilities in our implementation for simplicity.

Table 2: Single-threaded timing results of HomSign, DigitDecomp, and EvalFunc for $(\log P)$-bit encrypted numbers at $N = 2048, q = 2N = 4096$. Recall that in HomSign/DigitDecomp, as we proceed, $\log P$ becomes smaller and B_g is dynamically increased to improve the runtime performance, as suggested in Section 7.1.

Function	Q	$\log P$ [bits]	runtime [ms]	Initial B_g
HomSign/DigitDecomp	2^{12}	4	442	2^{27}
HomSign/DigitDecomp	2^{16}	8	1,322	2^{27}
HomSign/DigitDecomp	2^{20}	12	2,515	2^{18}
HomSign/DigitDecomp	2^{24}	16	3,709	2^{18}
HomSign/DigitDecomp	2^{25}	17	4,589	2^{18}
HomSign/DigitDecomp	2^{26}	18	5,216	2^{14}
HomSign/DigitDecomp	2^{28}	20	5,222	2^{14}
HomSign/DigitDecomp	2^{29}	21	6,096	2^{14}
EvalFunc	2^{12}	3	884	2^{27}

It is possible to use a smaller ring dimension $N = 1024$ and $\log Q' \leq 27$ for $Q = 2^{12}$ (but not for higher Q) at the cost of reducing $\log P$ by one bit, i.e., use the same bootstrapping parameters as for Boolean circuit evaluation in [29], but we have chosen to run all experiments at $N = 2048$ for simplicity/uniformity and best precision. Similarly, we can reduce n and Q_{ks} if $Q < 2^{29}$ is desired, hence reducing the runtime by a factor proportional to n. But we did not include this optimization to provide a general functionality up to 21 bits of precision and illustrate the linear dependence of runtime on $\log Q$ and $\log P$.

For comparison, the TFHE/GINX bootstrapping runtime for $N = 1024$ using the same parameters as in [29] with the CMake flag NATIVE_SIZE=32 for the clang++ 9.0.0 compiler was 74 ms (we observed that clang++ 9.0.0 is faster than g++ 10.1.0 when 32-bit integers are used for modular

arithmetic in PALISADE). This implies that the bootstrapping operations in our implementation are 6.0x (for $d_g = 2$), 8.1x (for $d_g = 3$), and 10.6x (for $d_g = 4$) slower than the bootstrapping time for a single Boolean gate evaluation [29] when using our computing environment. This slowdown is primarily caused by increased values of n from 502 to 1305 and N from 1024 to 2048 (both parameters proportionally increase the runtime). If a smaller precision (below 21 bits) is desired, this slowdown can be reduced by using smaller values of n (also, a smaller value of N can be used if the precision of 4 bits is sufficient for a given application).

8 Comparison with Other Recent Works

8.1 Comparison with algorithms based on FHEW/TFHE bootstrapping

There is a recent work proposing algorithms for homomorphic digit decomposition and arbitrary function evaluation [18]. The high-level idea of their approach is to use the fact that $-1 \cdot (-m) = m$ and extract the most significant bit as part of their procedures. They run two bootstrapping operations (one to extract the MSB and another to evaluate the desired function) and then multiply the results using a homomorphic multiplication, similar to the multiplication in Brakerski's and Brakerski/Fan-Vercauteren (BFV) schemes [9, 20]. The work [18] does not provide any implementation; hence we focus here on the theoretical comparison of approaches.

The most significant difference is the extra noise added in [18] due to the BFV-like homomorphic multiplication. This adds a multiplicative factor $O(N \cdot p)$ to the prior noise, and hence increases Q' by the same factor. In our method, no additional noise beyond the sum of the noises due to bootstrapping operations is needed. The other difference is that each iteration of their HomFloor-like operation in digit decomposition supports one bit less precision than our method. This bit is lost for the same reason that one extra bit is needed in our arbitrary function evaluation, where we have to extend from $\mathbb{Z}_q$ to $\mathbb{Z}_{2q}$.

We can estimate the concrete noise increase in [18] by using the heuristic BFV multiplication noise estimate, $4Np$, from [23, 26]. For the parameters used in our implementation (also accounting for a smaller p, by one bit), the extra factor is $4 \cdot 2^{11} \cdot 2^4 = 2^{17}$. This implies that $\log Q'$ has to be increased by 17 bits. According to [4] and our noise estimates, this will require increasing the ring dimension N from 2048 to 4096 to achieve the same security level and roughly the same precision (i.e., same $\log P$). The reduced precision per iteration of their HomFloor-like function may further increase the computational complexity. In summary, our estimates suggest that the method proposed in [18] will be at least two times slower for digit decomposition for the parameters used in our implementation. We expect a similar improvement for arbitrary function evaluation (except that our algorithm supports the same largest plaintext modulus as their algorithm, i.e., there is no 1-bit advantage as in the case of HomFloor).

Another potential drawback of the approach in [18] is the need for a BFV-like relinearization key and related extra implementation complexity. In this sense, our approach is simpler as it requires only regular FHEW/TFHE keys.

There is another recent work proposing an algorithm for arbitrary function evaluation [27]. The high-level idea is similar to [18], i.e., use the fact that $-1 \cdot (-m) = m$. The difference is that [27] performs multiplication using a GSW ciphertext (which encrypts the sign bit). They also propose a method to use an LWE$'$ ciphertext (a vector of LWE ciphertexts, see details in [29]) for multiplication instead of using a GSW ciphertext, as only plaintext multiplications are needed in

their algorithm, instead of ciphertext multiplications. This makes the extraction of the sign bit two times faster than the GSW-based method. Their algorithm requires $d'_g + 1 \geq 3$ bootstrappings to perform an arbitrary function evaluation whereas our method requires only 2 bootstrappings and is independent of d'_g. Here, d'_g refers to the number of digits for gadget decomposition specific to their LWE$'$ multiplication. Their algorithm also increases the noise by a multiplicative factor of $O(\sqrt{N d'_g} Q'^{1/d'_g})$, which is the cost of GSW-like multiplication, as compared to our approach.

Both methods [18, 27] can be extended to support large-precision sign evaluation (though this was not done in these works), but will have the same drawbacks as for arbitrary function evaluation: asymptotically higher noise growth (both methods) and increased number of bootstrapping operations (applies to [27] only). Another advantage of our method is that no multiplication support is needed for the homomorphic encryption scheme that invokes the FHEW bootstrapping, i.e., an additively homomorphic LWE scheme can be used. In methods [18, 27], a homomorphic encryption scheme supporting both additions and multiplications is needed.

8.2 Comparison with CKKS sign evaluation

We compare our sign-evaluation method with the state-of-the-art CKKS sign-evaluation method [28] (i.e., CKKS-based comparison between two numbers) and summarize the advantages of our approach below.

- As shown in Table VII of [28], with 20 bits of accuracy, their approach takes $\sim$ 30 seconds (for 64K slots), while ours takes about 6 seconds (for 1 slot). Therefore, if the number of comparisons needed is small (e.g., 5 comparisons), our method is faster.
- Our method is easily parallelizable while the CKKS method supports limited parallelization (only over RNS residues). Therefore, on a server-grade multi-core machine, our performance can be better even for a larger number of comparisons.
- When combined with CKKS for other applications (as shown in Section 9), our method does not require the ring dimension to be very large (2^{15} is already enough), while the CKKS method requires the ring dimension to be 2^{17} or higher, which may not be desired for the original application and therefore can greatly impact the performance, e.g., memory requirements and runtime.
- Higher precision for the CKKS method (e.g., 50 bits) can be harder to support as $\log Q$ can easily exceed 3000 (and the ring dimension will increase accordingly). The scaling factor will also need to be adjusted accordingly, increasing the underlying machine word size from 64-bit to 128-bit, which further reduces the performance (as high as 8x slower, judging by the PALISADE CKKS implementation). On the other hand, our method simply needs to use the RNS variant of RLWE as mentioned in Remark 2 of our paper (there is only an increase in the ring dimension). Hence, the decrease in runtime for higher precision is much smaller for our method.
- Our method is much simpler to implement/use (no special composite polynomials are needed).
- When multiple invocations of CKKS sign evaluation are needed, CKKS bootstrapping should be called in between, which significantly increases the runtime of the CKKS-based approach. Our method does not have any additional requirements for multiple invocations of the sign function as it inherently includes FHEW/TFHE bootstrapping.

9 Application

In this section, we consider an application of our large-precision comparison method where CKKS and FHEW/TFHE are used together. We combine CKKS and FHEW/TFHE using the scheme switching methods described in [25] based on the ideas proposed in [8].

The large-precision comparison is used to evaluate the Heaviside activation function arising in some machine learning applications [2, 11, 31], which is defined as

$$H(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

In the case of artificial neural network networks, e.g., in a deep learning model for functions with jump discontinuities, the input x is often computed as an inner product of (encrypted) inputs and (encrypted) weights, which can be performed using CKKS (along with other linear/polynomial computations needed for the model). In our example, we evaluate an inner product with CKKS and then evaluate the Heaviside function by negating the CKKS ciphertext containing 256 valid slots and switching it to 256 FHEW/TFHE ciphertexts. We perform our large-precision sign evaluation on these 256 ciphertexts using Algorithm 1. Lastly, we switch the comparison results back to a CKKS ciphertext.

In our experiment, the input precision was about 21 bits (by setting $\log Q = 29$ and other parameters as in Section 7) and the observed output precision was larger than 30 bits, which are both much higher than the results from [25] (input precision of 5-6 bits and output precision not higher than 13 bits). Similar to [25], the runtime for our experiment with 256 slots was dominated by large-precision comparisons, and the contribution of CKKS-FHEW and FHEW-CKKS scheme switching was not higher than 10%. Hence, the runtime can be estimated by multiplying the runtimes from Table 2 by the number of slots (and dividing them by the number of threads if multi-threading is available).

More generally, one can use large-precision comparison to perform an encrypted branch evaluation by checking the values against a threshold (i.e., if the input is above some threshold T, evaluate circuit B; otherwise, evaluate circuit C). This may require high precision as the behavior of B and C can be greatly different.

10 Concluding Remarks

Our experimental results for homomorphic sign evaluation suggest that increasing the precision from 4 bits to 21 incurs a slow-down of only about 14x. If FHEW/TFHE bootstrapping would be used directly, a slow-down of more than 100,000x would be observed. This implies that our large-precision homomorphic sign evaluation implementation can be used for applications that work with 20-bit-precision numbers (and can be extended to a larger precision, as discussed in Remark 2 in Section 7.1). For instance, it can be plugged into the decision tree inference implementation [25] to increase the precision of comparison.

It was also shown that our method for arbitrary function evaluation, which we call general functional bootstrapping (often referred to as programmable bootstrapping in literature), has a lower complexity than two other recently proposed methods [18, 27]. Both of these methods require one multiplication operation while our method can be built on top of an additively homomorphic encryption scheme, similar to the original FHEW construction for Boolean gate evaluation [19].

References

[1] PALISADE Lattice Cryptography Library (release 1.11.6). https://palisade-crypto.org/, Jan. 2022.

[2] Spline adaptive filtering algorithm based on heaviside step function. *Signal, Image and Video Processing*, 2022.

[3] A. Akavia, M. Leibovich, Y. S. Resheff, R. Ron, M. Shahar, and M. Vald. Privacy-preserving decision tree training and prediction against malicious server. Cryptology ePrint Archive, Report 2019/1282, 2019. https://ia.cr/2019/1282.

[4] M. Albrecht, M. Chase, H. Chen, and et al. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[5] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[6] C. Bonte, I. Iliashenko, J. Park, H. V. L. Pereira, and N. P. Smart. Final: Faster fhe instantiated with ntru and lwe. Cryptology ePrint Archive, Report 2022/074, 2022. https://ia.cr/2022/074.

[7] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. Cryptology ePrint Archive, Report 2020/1203, 2020. https://eprint.iacr.org/2020/1203.

[8] C. Boura, N. Gama, M. Georgieva, and D. Jetchev. Chimera: Combining ring-lwe-based fully homomorphic encryption schemes. *Journal of Mathematical Cryptology*, 14(1):316–338, 2020.

[9] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Annual Cryptology Conference*, pages 868–886. Springer, 2012.

[10] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[11] E. Burlakov, E. Zhukovskiy, and V. Verkhlyutov. Neural field equations with neuron-dependent heaviside-type activation function and spatial-dependent delay. *Mathematical Methods in the Applied Sciences*, 44(15):11895–11903, 2021.

[12] H. Chen, W. Dai, M. Kim, and Y. Song. Efficient homomorphic conversion between (ring) lwe ciphertexts. In K. Sako and N. O. Tippenhauer, editors, *Applied Cryptography and Network Security*, pages 460–479, Cham, 2021. Springer.

[13] H. Chen, R. Gilad-Bachrach, K. Han, Z. Huang, A. Jalali, K. Laine, and K. Lauter. Logistic regression over encrypted data from fully homomorphic encryption. Cryptology ePrint Archive, Report 2018/462, 2018. https://ia.cr/2018/462.

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

[15] J. H. Cheon, D. Kim, and D. Kim. Efficient homomorphic comparison methods with optimal complexity. In S. Moriai and H. Wang, editors, *Advances in Cryptology – ASIACRYPT 2020*, pages 221–256, Cham, 2020. Springer.

[16] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In J. H. Cheon and T. Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 3–33, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[17] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for tfhe. In T. Takagi and T. Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 377–408, Cham, 2017. Springer.

[18] I. Chillotti, D. Ligier, J.-B. Orfila, and S. Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for tfhe. Cryptology ePrint Archive, Report 2021/729, 2021. `https://eprint.iacr.org/2021/729`.

[19] L. Ducas and D. Micciancio. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 617–640, Berlin, Heidelberg, 2015. Springer.

[20] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012:144, 2012.

[21] N. Genise, D. Micciancio, and Y. Polyakov. Building an efficient lattice gadget toolkit: Subgaussian sampling and more. In Y. Ishai and V. Rijmen, editors, *EUROCRYPT 2019*, pages 655–684, Cham, 2019. Springer.

[22] A. Guimarães, E. Borin, and D. F. Aranha. Revisiting the functional bootstrap in tfhe. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021:229–253, Feb. 2021.

[23] S. Halevi, Y. Polyakov, and V. Shoup. An improved rns variant of the bfv homomorphic encryption scheme. In M. Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, pages 83–105, Cham, 2019. Springer.

[24] I. Iliashenko and V. Zucca. Faster homomorphic comparison operations for bgv and bfv. *Proceedings on Privacy Enhancing Technologies*, 2021(3):246–264, 2021.

[25] W. jie Lu, Z. Huang, C. Hong, Y. Ma, and H. Qu. Pegasus: Bridging polynomial and non-polynomial evaluations in homomorphic encryption. SP 2021, 2020. `https://eprint.iacr.org/2020/1606`.

[26] A. Kim, Y. Polyakov, and V. Zucca. Revisiting homomorphic encryption schemes for finite fields. In *ASIACRYPT 2021*, page 608–639, Berlin, Heidelberg, 2021. Springer.

[27] K. Kluczniak and L. Schild. Fdfb: Full domain functional bootstrapping towards practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2021/1135, 2021. `https://ia.cr/2021/1135`.

[28] E. Lee, J.-W. Lee, Y.-S. Kim, and J.-S. No. Optimization of homomorphic comparison algorithm on rns-ckks scheme. Cryptology ePrint Archive, Report 2021/1215, 2021. https://ia.cr/2021/1215.

[29] D. Micciancio and Y. Polyakov. *Bootstrapping in FHEW-like Cryptosystems*, page 17–28. Association for Computing Machinery, New York, NY, USA, 2021.

[30] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.

[31] J. W. Siegel and J. Xu. Optimal convergence rates for the orthogonal greedy algorithm. *IEEE Transactions on Information Theory*, pages 1–1, 2022.

[32] Z. Yang, X. Xie, H. Shen, S. Chen, and J. Zhou. Tota: Fully homomorphic encryption with smaller parameters and stronger security. Cryptology ePrint Archive, Paper 2021/1347, 2021. https://eprint.iacr.org/2021/1347.