Amortized Functional Bootstrapping in less than 7ms, with $\tilde{O}(1)$ Polynomial Multiplications

Zeyu Liu
zeyu.liu@yale.edu
Yale University

Yunhao Wang[*]
yunhao.wang@yale.edu
Yale University

October 7, 2023

Abstract

Amortized bootstrapping offers a way to refresh multiple ciphertexts of a fully homomorphic encryption scheme in parallel more efficiently than refreshing a single ciphertext at a time. Micciancio and Sorrell (ICALP 2018) first proposed the technique to bootstrap n LWE ciphertexts simultaneously, reducing the total cost from $\tilde{O}(n^2)$ to $\tilde{O}(3^\epsilon n^{1+\frac{1}{\epsilon}})$ for arbitrary $\epsilon > 0$. Several recent works have further improved the asymptotic cost. Despite these amazing progresses in theoretical efficiency, none of them demonstrates the practicality of batched LWE ciphertext bootstrapping. Moreover, most of these works only support limited functional bootstrapping, i.e. only supporting the evaluation of some specific type of function when performing bootstrapping.

In this work, we propose a construction that is not only asymptotically efficient (requiring only $\tilde{O}(n)$ polynomial multiplications for bootstrapping of n LWE ciphertexts) but also concretely efficient. We implement our scheme as a C++ library and show that it takes $<$ 5ms per LWE ciphertext to bootstrap for a binary gate, which is an order of magnitude faster than the state-of-the-art C++ implementation on LWE ciphertext bootstrapping in OpenFHE. Furthermore, our construction supports batched arbitrary functional bootstrapping. For a 9-bit messages space, our scheme takes ∼6.7ms per LWE ciphertext to evaluate an arbitrary function with bootstrapping, which is about two to three magnitudes faster than all the existing schemes that achieve a similar functionality and message space.

[*] Part of the work was done when the author was at Columbia University.

Contents

1 Introduction

Fully homomorphic encryption (FHE) schemes support the evaluation of an arbitrary circuit over the encrypted data without decrypting it. Thus, it is a very powerful tool that can be widely applied in many scenarios, like secure machine learning[30], private information retrieval[42], private set intersection [17], and so on.

There are two major lines of FHE schemes. The first is BGV and its variants [9, 7, 19, 14]. These works encrypt a vector of N messages using an RLWE ciphertext. This line of work allows the "Single Instruction Multiple Data" (SIMD) operations [*]. Thus, they provide an efficient way to evaluate a circuit with a large number of inputs. However, there exist two major limitations of this line of work: (1) they natively only support circuits with some pre-known depth; since each level of circuit evaluation adds extra noise to the RLWE ciphertexts, for deeper circuits, the noise may accumulate to a point that no more operations can be evaluated. Therefore, bootstrapping is needed to reduce the noise inside an RLWE ciphertext. For BGV/BFV, the amortized bootstrapping time for a 16-bit message can be over 300 milliseconds [25, 12]. (2) they only efficiently support addition and multiplication, while non-polynomial evaluation (e.g., comparisons) can be very costly.

The second line of work starts with FHEW [18] and is later improved by TFHE [15] and Lee et al. [36]. These works support bootstrapping in tens of milliseconds. Instead of batching N messages in a single RLWE ciphertext, they encrypt a single bit inside an LWE ciphertext. Bootstrapping is done together with a NAND gate evaluation between two encrypted bits. The output ciphertext maintains the same level of noise as the input ciphertext. Since the NAND gate is a universal gate, these works can be used to evaluate arbitrary circuits. Besides, they provide a more general functionality, called functional bootstrapping: given an LWE ciphertext encrypting a message m, one can obtain an output LWE ciphertext encrypting $f(m)$, while the input ciphertext and output ciphertext have the same level of noise. However, they also have two major limitations: (1) they focus on bootstrapping one message at a time and are not able to take advantage of the SIMD feature as in the other line of work; (2) their functional bootstrapping only supports negacyclic functions (i.e., functions $f : \mathbb{Z}_q \to \mathbb{Z}$ such that $f(x) = -f(x + q/2)$).

[44] first attempts to combine these two lines of work and introduces the batched bootstrapping to support fast bootstrapping over n LWE ciphertexts at the same time. Recently, several works [24, 45, 37, 38] have further greatly improved the asymptotic behavior ([38] gives an algorithm with an amortized cost of $\tilde{O}(1)$ polynomial multiplications per LWE ciphertext bootstrapping). However, whether these works are practical remains to be an open problem. The only implementation in [24] requires > 1 second to bootstrap a single LWE ciphertext encrypting a 7-bit message (amortized over n ciphertexts).

Therefore, the central question we address is:

Can we construct a batched bootstrapping algorithm for LWE ciphertexts, that is (1) practical, taking only milliseconds per LWE ciphertext bootstrapping; (2) asymptotically efficient, requiring only $\tilde{O}(1)$ polynomial multiplications per LWE ciphertext bootstrapping, and (3) flexible, supporting an arbitrary function evaluation during the bootstrapping procedure?

In this work, we firmly answer *yes* to this question. We design algorithms that are both asymptotically efficient ($\tilde{O}(1)$ polynomial multiplications per LWE ciphertext bootstrapping) and concretely fast (orders of magnitude faster than any existing works), while supporting functional bootstrapping for arbitrary functions.

1.1 Our Contribution

Batched LWE ciphertext bootstrapping for NAND. We propose a novel method to perform batched NAND gate evaluation between two LWE ciphertexts without increasing the error. Since the NAND gate is universal and can be used to construct any circuits, this achieves the property of fully homomorphic operation. The asymptotic amortized cost is $\tilde{O}(1)$ homomorphic multiplications per NAND gate, which is almost optimal.

[*] In other words, they allow evaluating multiplication and additions over two vectors (component-wisely), each with N messages, by evaluating the same operations over the two RLWE ciphertexts encrypting those two vectors.

Batched LWE ciphertext bootstrapping for general binary gates. We extend our scheme to support other types of binary gates without any overhead. Moreover, we support evaluating different gates in parallel (instead of requiring all gates to be the same type) when performing the batched bootstrapping.

Batched arbitrary functional bootstrapping. We further extend our scheme to support arbitrary function evaluation. In more detail, we allow one to evaluate an arbitrary function $f : \mathbb{Z}_p \to \mathbb{Z}_p$ any message $m \in \mathbb{Z}_p$ encrypted in an LWE ciphertext, without increasing the error of the output LWE ciphertexts. The asymptotic amortized cost remains $\tilde{O}(1)$ homomorphic multiplications per LWE ciphertext.

Implementation and evaluation. We implement our schemes as an open-sourced C++ library and measure the concrete performance for a variety of parameters to compare with prior works. Salient observations include:

- For arbitrary binary gates, the cost is less than 5ms per gate, which is *more than an order of magnitude faster* than the state-of-the-art C++ implementation of TFHE bootstrapping (OpenFHE [4]), and more than 3x faster than the state-of-the-art rust implementation (TFHE-rs [50]).
- For arbitrary function evaluation of $m \in \mathbb{Z}_p$, the cost is $\sim$6.7ms when p is of 9 bits, and $\sim$39ms when p has 12 bits. Both results are about *two to three orders of magnitude faster* than all the existing arbitrary function evaluation methods providing the same plaintext space.

FHEW/TFHE - BFV/BGV scheme switching. Our construction can be adapted to perform scheme switching between FHEW/TFHE-like cryptosystems and BFV/BGV, which is of its own interest. We also benchmark this scheme switching method: $\sim$296 seconds to switch 32768 LWE ciphertexts into 1 BFV ciphertext, and $\sim$17 seconds to switch a single BFV ciphertext into 32768 LWE ciphertexts.

1.2 Technical Overview

Main idea. Our starting point comes from the observation that the BFV/BGV HE schemes operate over a finite field $\mathbb{Z}_t$ for some prime t. Thus, for an LWE ciphertext $(\vec{a}, b) \in \mathbb{Z}_t^{n+1}$, encrypted under secret key $\mathsf{sk} \in \mathbb{Z}_t^n$ (which satisfies $b \equiv \langle \vec{a}, \mathsf{sk} \rangle + \alpha \cdot m + e \bmod t$, for some message m, encoding parameter α and error e), we homomorphically evaluate $b - \langle \vec{a}, \mathsf{sk} \rangle$ resulting in $\alpha \cdot m + e \pmod t \in \mathbb{Z}_t$ instead of $\alpha \cdot m + e + k \cdot t \in \mathbb{Z}$. To recover m, we simply evaluate a polynomial function f over $\mathbb{Z}_t$, where $f : \mathbb{Z}_t \to \mathbb{Z}_t$ is a degree $t - 1$ polynomial function.

Therefore, evaluating $b - \langle \vec{a}, \mathsf{sk} \rangle$ and function f using BFV/BGV homomorphically decrypts an LWE ciphertext into a BFV/BGV ciphertext encrypting m. Due to the SIMD feature of BFV/BGV, the decryption of N LWE ciphertexts can be evaluated at the same time. Then, we just need to switch a BFV/BGV ciphertext with an underlying ring dimension N into N LWE ciphertexts. To achieve this, we adapt the $\mathsf{SlotToCoeff}$ algorithm introduced in [13] to the BFV setting followed by the $\mathsf{SampleExtract}$ algorithm introduced in [15].

Optimizations. We provide various techniques to further optimize the performance.

- We maximize the number of zero coefficients of the function f to make the evaluation more efficient.
- We minimize the number of rotations and ciphertext multiplications by using the baby-step-giant-step linear transformation [26, 30], Paterson-Stockmayer algorithm [47], and generating additional bootstrapping keys.
- To support evaluating different binary gates in parallel when doing bootstrapping, we introduce ways to pre-process and post-process the LWE ciphertexts, so that we can take advantage of the SIMD feature of the underlying BFV scheme even when the gates are distinct.

1.3 Organization

The rest of the paper is organized as follows. Section 2 discusses related works and compares our construction with the existing (batched) LWE ciphertexts bootstrapping methods in terms of asymptotic efficiency and

	Ours	Prior works			Concurrent Works	
Scheme(s)	§4-§6	[18, 15, 36]	[44]	[24, 45]	[37]	[38]
Amortized Cost per Binary Gate Bootstrapping	$\tilde{O}(1)$	$O(n)$	$\tilde{O}(3^{1/\epsilon} \cdot n^{\epsilon})$	$O(\epsilon \cdot n^{1/\epsilon})$	$\tilde{O}(n^{0.75})$	$\tilde{O}(1)$
Arbitrary Function Evaluation	Yes	No		Yes (by [24])	No	
Implementation	Yes	No		Yes (by [24])	No	

Table 1: Comparisons with prior works. Amortized cost per binary gate bootstrapping counts the number of FHE multiplications per LWE ciphertext bootstrapping (amortized over $N > n$ ciphertexts, where n is the secret key dimension of the LWE ciphertexts). Arbitrary function evaluation is whether the scheme supports arbitrary lookup table evaluation over the plaintext space.

functionality. Section 3 introduces some necessary background on LWE ciphertexts bootstrapping and BFV homomorphic encryption scheme. Section 4 describes our main construction to do batched LWE ciphertexts bootstrapping for NAND gates. Section 5 extends the construction to allow arbitrary binary gates evaluation in parallel. Section 6 further adapts the construction to support batched arbitrary functional bootstrapping. Section 7 discusses our implementation, experimental results, and comparisons with other schemes providing similar or weaker functionalities. Section 8 introduces some extensions of our construction. Section 9 concludes the paper.

2 Related Works

2.1 (Batched) FHEW/TFHE-like Bootstrapping

We give a comparison between our work and the prior works on FHEW/TFHE-like cryptosystems in Table 1, regarding the asymptotic efficiency and functionalities. For concrete performance comparisons, see Section 7.

Non-batched schemes. FHEW [18] cryptosystem was introduced to focus on evaluating NAND gate between two LWE ciphertexts without increasing the error of the output ciphertext. As NAND gate is universal, such a cryptosystem can be used to homomorphically evaluate any arbitrary binary circuit. It is later improved by TFHE [15] and Lee et al. [36]. However, all of these works primarily focus on evaluating a single NAND gate at a time and are both asymptotically and concretely relatively costly.

Prior works on batched bootstrapping. Batched bootstrapping instead evaluate multiple NAND gates in parallel. [44] was the first to introduce this concept, and was later improved and implemented by [24]. Another work [45] also achieves the same asymptotic efficiency as in [24].

Concurrent and independent works on batched bootstrapping. Two recent works [37] and [38] made great progress along this path. [38] improves the asymptotic cost to $\tilde{O}(1)$ homomorphic multiplications per gate bootstrapping, which is asymptotically the same as our construction and is almost optimal. However, to our knowledge, these two works have not yet been implemented, so whether they are concretely efficient remains to be an open problem.

2.2 Arbitrary Function Evaluation

One important feature of FHEW/TFHE-like cryptosystems is that they allow functional bootstrapping: one can evaluate a function over an LWE ciphertext without increasing the error. However, to our knowledge, most of the schemes in Section 2.1 (see Table 1) can only be used to evaluate a negacyclic function (i.e., a function $f : \mathbb{Z}_q \to \mathbb{Z}$ satisfying $f(x) = -f(x + q/2)$). Moreover, the input LWE ciphertexts of these works have small precision (normally ≤ 5 bits to remain practical). Thus, the capability is very limited. On the other hand, another line of work constructs arbitrary function evaluation for this kind of cryptosystem, some of which also try to allow larger precision (e.g., 10-20 bits).

Prior works on arbitrary function evaluation. [16, 33, 40, 51] develop distinct ways to evaluate arbitrary functions (each can be represented as look-up tables (LUT) over the plaintext space). [16, 33] rely on homomorphic multiplications ([16] relying on BFV-like multiplications and [33] relying on GSW-based multiplications). [40, 51] are instead based on a more lightweight method (only addition between LWE ciphertexts is needed).

Note that the basic methods of all these works only work for small precision. [16, 40] provide a way to extend their scheme to allow larger precisions by first dividing a large-precision input ciphertext (e.g., 21-bit precision) into several small-precision ciphertexts (e.g., 7 ciphertexts each with 3-bit precision). Then, these small ciphertexts are fed into the algorithms introduced in [23] to evaluate large-precision functions (large-precision LUTs). A recent concurrent and independent work [41] further improves upon [40, 33].

[23] itself provides a way to evaluate a large LUT over a vector of ciphertexts with small plaintext space. [39] (concurrent and independent) further optimize this method. Note that this method assumes the input to be a vector of ciphertexts with small precision, instead of with large precision. Hence, in most applications, the ciphertexts are required to be decomposed first by applying algorithms in [16, 40], which introduces an extra overhead.

2.3 Other Related Works

Bootstrapping, first introduced by [21], is greatly explored in many works. Our construction is similar to the bootstrapping procedure introduced in [19] in that we also take advantage of the free modulo t operation when homomorphically evaluating a circuit using BFV, where t is the plaintext space. However, our work is different from [19] in several ways. First, our goal is to support batched functional bootstrapping for LWE ciphertexts, while [19] simply aims to reduce the error for a BFV ciphertext. Thus, our construction not only directly achieves the functionality of FHE, but also provides much more flexibility when evluating the bootstrapping circuit. Second, [19] discusses t being a power-of-two, which is not very compatible with power-of-two cyclotomics, and thus limits its practicality. Our construction, instead, uses a large prime field t, to guarantee practicality. Third, we introduce additional optimization techniques to make our procedure concretely efficient.

3 Preliminary

Let N be a power of two. Let $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ denote the $2N$-th cyclotomic ring, and $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R}$ for some $Q \in \mathbb{Z}$. Let $[n]$ denote the set $\{1, \ldots, n\}$. Let $\vec{a}$ denote a vector and $\vec{a}[i]$ denote the i-th element of $\vec{a}$. Similarly, if A is a matrix, let $A[i][j]$ denote the element on the i-th row and j-th column of matrix A. Let $\|\vec{x}\|_\ell$ denote the ℓ-norm for vector $\vec{x}$ (calculated as $(\sum_{i\in|\vec{x}|} \vec{x}[i]^\ell)^{1/\ell}$). If $x \in \mathcal{R}$, let $\|x\|_\ell$ denote the ℓ-norm of the coefficient vector of x, and let $x[i]$ denote the i-th coefficient of x.

When a function needs to take a key but is called without the key (e.g., $\mathsf{Dec}(\mathsf{ct})$ where ct is some LWE ciphertext and Dec is the decryption procedure of LWE scheme), it is assumed that the key is taken implicitly and correctly unless otherwise specified.

3.1 Hard Problems

Definition 3.1 (Decisional learning with error problem)**.** Let $n, q, \mathcal{D}, \chi$ be parameters dependent on λ. The learning with error (LWE) problem states the following: for $a \leftarrow_\$ \mathbb{Z}_q^n$ sampled uniformly at random, it holds that $(a, \langle a, s\rangle + e) \approx_c (a, b)$, where $s \leftarrow \mathcal{D}, e \leftarrow \chi$ and $b \leftarrow_\$ \mathbb{Z}_q$.

Let $\mathsf{LWE}_{n,q,\mathcal{D},\chi}$ denote the LWE assumption parameterized by $n, q, \mathcal{D}, \chi$.

Definition 3.2 (Decisional ring learning with error problem)**.** Let $N, Q, \mathcal{D}, \chi$ be parameters dependent on λ and N being a power of two. Let $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$. The ring learning with error (RLWE) problem states the following: for $a \leftarrow_\$ \mathcal{R}_Q$ sampled uniformly at random, it holds that $(a, a \cdot s + e) \approx_c (a, b)$, where $s \leftarrow \mathcal{D}, e \leftarrow \chi$ and $b \leftarrow_\$ \mathcal{R}_Q$.

Let $\mathsf{RLWE}_{N,Q,\mathcal{D},\chi}$ denote the RLWE assumption parameterized by $N, Q, \mathcal{D}, \chi$.

3.2 FHEW/TFHE Cryptosystem

FHEW was first introduced in [18], and later improved by TFHE [15]. Recent work [36] also follows this line of work.

All these bootstrapping procedures are based on (CPA secure) LWE encryption parameterized by a secret dimension n, ciphertext modulus q, plaintext modulus p, secret key distirbution $\mathcal{D}$, and error distribution χ (such that $\mathsf{LWE}_{n,q,\mathcal{D},\chi}$ holds) defined as follows: under (secret) key $\mathsf{sk} \leftarrow \mathcal{D}$, the LWE encryption of a message $m \in \mathbb{Z}_p$ is a vector $\mathsf{ct} = (\vec{a}, b) \in \mathbb{Z}_q^{n+1}$ such that

$$b = \langle \vec{a}, \mathsf{sk} \rangle + \alpha \cdot m + e \pmod q$$

where $\alpha = \lfloor q/p \rfloor$, and $e \leftarrow \chi$ is a small error term satisfying $|e| < \lfloor q/(2p) \rfloor$. The message m is recovered by computing the LWE decryption function

$$\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) = \left\lceil \frac{b - \langle \vec{a}, \mathsf{sk} \rangle \pmod q}{\alpha} \right\rfloor$$

Let $\mathsf{err}(\mathsf{ct})$ denote $e \leftarrow \mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) - \alpha \cdot m$, where sk is the corresponding secret key of ct, $m = \mathsf{Dec}(\mathsf{ct})$, and $\alpha = \lfloor q/p \rfloor$.

FHEW/TFHE functional bootstrapping. FHEW/TFHE bootstrapping procedure, denoted by $\mathsf{Boot}(\mathsf{btk}, f, \mathsf{ct})$ satisfying the following property:

Given a correct bootstrapping key btk generated from sk, any negacylic function $f : \mathbb{Z}_q \to \mathbb{Z}$ (i.e., $f(x + q/2) = -f(x)$), and any ciphertext ct with $\mathsf{err}(\mathsf{ct}) < \beta$ encrypted under sk, let $\mathsf{ct}' \leftarrow \mathsf{Boot}(\mathsf{btk}, f, \mathsf{ct})$, it satisfies that $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}') \cdot \alpha = f(\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) \cdot \alpha) \pmod q$ and $\mathsf{err}(\mathsf{ct}') < \beta'$ where $\beta' \leq \beta$.

When we use $\mathsf{Boot}(f, \mathsf{ct})$, we implicitly mean that Boot is called with the correct bootstrapping key btk.

Note that this is the generalized functional bootstrapping achieved by [18, 15]. In this paper, we achieve an even stronger functionality than this generalized bootstrapping by removing the requirement that f is negacyclic.

FHEW/TFHE bootstrapping for the NAND gate. To evaluate a NAND gate (which is a universal gate) between two LWE ciphertexts without increasing the error, the procedure is as follows (which is a special application of the generalized functional bootstrapping described above):

- Let $p = 4$. Let $\mathsf{ct}_1, \mathsf{ct}_2$ denote the two input LWE ciphertexts, each of which is encrypting either 0 or 1, with error $\mathsf{err}(\mathsf{ct}_b) < q/16$ for $b \in \{1, 2\}$.

- Compute $\mathsf{ct} \leftarrow \mathsf{ct}_1 + \mathsf{ct}_2$, ct encrypting $0, 1$ or 2, with error $< q/16 + q/16 = q/8$.

- Let $f : \mathbb{Z}_q \to \mathbb{Z}$ be defined as follows:

$$f(x) = \begin{cases} q/8 & \text{if } -q/8 \leq x < 3q/8 \\ -q/8 & \text{otherwise} \end{cases}$$

  which is a negacyclic function. Compute $(\vec{a}', b') = \mathsf{ct}' \leftarrow \mathsf{Boot}(f, \mathsf{ct})$, and then $b' \leftarrow b' + q/8$, which gives $\mathsf{Dec}(\mathsf{ct}') = 0$ if $\mathsf{Dec}(\mathsf{ct}_1 + \mathsf{ct}_2) = 2$ and $\mathsf{Dec}(\mathsf{ct}') = 1$ otherwise, with $\mathsf{err}(\mathsf{ct}') < q/16$.

FHEW/TFHE bootstrapping uses two additional procedures: modulus switching and key switching, which are also used in our construction, we list those two techniques as follows.

Modulus switching. Modulus switching procedure is defined as $\mathsf{ModSwitch}(\mathsf{ct}, q') = (q/q')\mathsf{ct} = \lceil (q/q')(\vec{a}, b) \rfloor$ where $\mathsf{ct} = (\vec{a}, b)$ is an LWE ciphertext with ciphertext modulus q. We formalize it using the following lemma, adapted from [18]:

Lemma 3.1 (Modulus switching)**.** Let $\mathsf{ct} = (\vec{a}, b) \in \mathbb{Z}_q^{n+1}$ be an LWE encryption of a message $m \in \mathbb{Z}_p$ under secret key $\mathsf{sk} \in \mathbb{Z}^n$, with ciphertext modulus q and noise bound $\mathsf{err}(\mathsf{ct}) < \beta$. Then, for any modulus q', the rounded ciphertext $\mathsf{ct}' = (\vec{a}', b') \leftarrow \mathsf{ModSwitch}(\mathsf{ct}, q')$ is an encryption of the same message m under sk with ciphertext modulus q' and noise bound $|\mathsf{Dec}(\vec{a}', b') - \lfloor q'/p \rfloor m| < (q'/q)\beta + \beta''$, where $\beta'' = \frac{1}{2}(\|\mathsf{sk}\|_1 + 1)$.

Note that for FHEW/TFHE and other FHE schemes, sk is usually a short vector (i.e., $\|\mathsf{sk}\|_\infty \leq \delta$ for some small δ, e.g., $\delta = 1$ for ternary and binary secret keys).

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 $O(\|\mathsf{sk}\|_2)$. For uniformly random ternary keys $\mathsf{sk} \in \{0, 1, -1\}^n$, it satisfies $\beta'' \approx \sqrt{n}/3$ as discussed in [18, 40].

Key switching. Key switching allows converting an LWE encryption under a key $\mathsf{sk} \in \mathbb{Z}_q^n$ into an LWE encryption of the same message with the same ciphertext modulus and plaintext modulus (and slightly larger error) under a different key $\mathsf{sk}' \in \mathbb{Z}_q^{n'}$. The key switching procedure is parameterized by a base B_{ks} (e.g., 2). Let $d_{\mathsf{ks}} = \lceil \log_{B_{\mathsf{ks}}}(q) \rceil$. Let $\mathsf{ksk}_{i,j,v} \in \mathbb{Z}_q^{n'+1} \leftarrow (\vec{\alpha}_{i,j,v}, \langle \vec{\alpha}_{i,j,v}, \mathsf{sk}' \rangle + e + v\mathsf{sk}[i]B_{\mathsf{ks}}^i)$ for $i \in [n], v \in [B_{\mathsf{ks}}], j \in [d_{\mathsf{ks}}]$, where $\vec{\alpha}_{\cdot,\cdot,\cdot}$ is a randomly sampled vector from $\mathbb{Z}_q^{n'}$ and e is some small error sampled from χ_σ (some Gaussian distribution with mean 0 and standard deviation σ). Let $K = \{\mathsf{ksk}_{i,j,v}\}$ denote the key switching key. $\mathsf{KeySwitch}(K, \mathsf{ct})$ is then defined as follows: given $(\vec{a}, b) \in \mathbb{Z}_q^{n+1}$ as the input ciphertext, first compute the base-B_{ks} expansion of $\vec{a}[i] = \sum_{j \in [d_{\mathsf{ks}}]} \vec{a}[i]_j B_{\mathsf{ks}}^j$, for all $i \in [n]$; then, output $\mathsf{ct}' = (\vec{0}, b) - \sum_{i \in [n], j \in [d_{\mathsf{ks}}]} \mathsf{ksk}_{i,j,\vec{a}[i]_j}$. We formalize this property using the following lemma, adapted from [18].

Lemma 3.2 (Key switching)**.** Given an LWE ciphertext $\mathsf{ct} \in \mathbb{Z}_q^{n+1}$ encrypting message $m \in \mathbb{Z}_p$ under secret key $\mathsf{sk} \in \mathbb{Z}_q^n$, and a key switching key K generated using sk and $\mathsf{sk}' \in \mathbb{Z}_q^{n'}$, let $\mathsf{ct}' \in \mathbb{Z}_q^{n'+1} \leftarrow \mathsf{KeySwitch}(K, \mathsf{ct})$, it holds that $\mathsf{Dec}(\mathsf{ct}', \mathsf{sk}') = \mathsf{Dec}(\mathsf{ct}, \mathsf{sk})$ and $\mathsf{err}(\mathsf{ct}') \leftarrow \chi_{\sigma + n \cdot d_{\mathsf{ks}} \cdot \sigma'}$ where σ is the error standard deviation for ct and σ' is the error standard deviation for each element in the key switching key.

The security of key switching is also intuitive. Essentially, key switching is simply summing up key-switching keys, which are all LWE ciphertexts, and since all the information needed is public, the resulting ciphertext is semantically secure.

3.3 B/FV Leveled Homomorphic Encryption

The BFV leveled homomorphic encryption scheme is first introduced in [8] using standard LWE assumption, and later adapted to ring LWE assumption by [20].

Given a polynomial $\in \mathcal{R}_t = \mathbb{Z}_t[X]/(X^N + 1)$, the BFV scheme encrypts it into a ciphertext consisting of two polynomials, where each polynomial is from a larger cyclotomic ring $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N + 1)$ for some $Q > t$. We refer t as the plaintext modulus, Q as the ciphertext modulus, and N as the ring dimension. t satisfies that $t \equiv 1 \mod 2N$, where N is a power of two.

Plaintext encoding. To encrypt a plaintext $\vec{m} = (m_1, \ldots, m_N) \in \mathbb{Z}_t^N$, BFV creates polynomial $m(X) = \sum_{i \in [N]} m_i X^{i-1}$, and then encodes it by constructing another polynomial $y(X) = \sum_{i \in [N]} y_i X^{i-1}$ where $m_i = y(\eta_j)$, $\eta_j := \eta^{3^j} \mod t$, and η is the $2N$-th primitive root of unity of t. Such encoding can be done using an Inverse Number Theoretic Transformation (INTT), which is a linear transformation (and can be represented as matrix multiplication).

Encryption and decryption. The BFV ciphertext encrypting $\vec{m}$ under $\mathsf{sk} \leftarrow \mathcal{D}$ has the format $\mathsf{ct} = (a, b) \in \mathcal{R}_Q^2$, satisfying $b - a \cdot \mathsf{sk} = \lfloor Q/t \rfloor \cdot y + e$ where $\lfloor Q/t \rfloor \cdot y \in \mathcal{R}_Q$ and y is the polynomial encoded in the way above, and e is some small error term sampled from some Gaussian distribution over $\mathcal{R}_Q$. Note that this encryption using $\lfloor Q/t \rfloor \cdot y$ for some message y is exactly the same as the LWE encryption we have above (there we have $\alpha = \lfloor q/p \rfloor$).

Symmetric key encryption can be done by simply sampling a random a and constructing b accordingly using sk. Public key encryption can also be achieved easily but it is not relevant to our paper so we refer the readers to [8, 20, 31] for details.

Decryption is thus to calculate $y' \leftarrow \lceil (t/Q) \cdot (b - a \cdot \mathsf{sk}) \rfloor \in \mathcal{R}_t$ (note that $(b - a \cdot \mathsf{sk})$ is done over $\mathcal{R}_Q$), and then decodes it by applying a procedure to revert the encoding process (which is also a linear transformation). We assume $\mathsf{BFV.Dec}$ outputs plaintext $\in \mathbb{Z}_t^N$, which is the decoded form, for simplicity. Similarly, we assume $\mathsf{BFV.Enc}$ contains the encoding process.

BFV operations. BFV essentially supports addition, multiplication, rotation, and polynomial function evaluation, satisfying the following property:

- (Addition) $\mathsf{BFV.Dec}(\mathsf{ct}_1 + \mathsf{ct}_2) = \mathsf{BFV.Dec}(\mathsf{ct}_1) + \mathsf{BFV.Dec}(\mathsf{ct}_2)$
- (Multiplication) $\mathsf{BFV.Dec}(\mathsf{ct}_1 \times \mathsf{ct}_2) = \mathsf{BFV.Dec}(\mathsf{ct}_1) \times \mathsf{BFV.Dec}(\mathsf{ct}_2)$
- (Rotation) $\mathsf{BFV.Dec}(\mathsf{rot}(\mathsf{ct}, j))[i] = \mathsf{BFV.Dec}(\mathsf{ct})[i + j \pmod N]$ for all $i, j \in [N]$
- (Polynomial evaluation) $\mathsf{BFV.Dec}(\mathsf{BFV.Eval}(\mathsf{ct}, f)) = f(\mathsf{BFV.Dec}(\mathsf{ct}))$, where $f : \mathbb{Z}_t \to \mathbb{Z}_t$ is a polynomial function. Note that this is implied by addition and multiplication.

BFV ciphertexts addition is done by adding the two pairs of polynomials accordingly (i.e., $\mathsf{ct}_1 + \mathsf{ct}_2 = (a_1 + a_2, b_1 + b_2)$. Multiplication requires a tensor product between two ciphertexts followed by a relinearization processing (i.e., a way to bring the product result of three ring elements back to two elements), altogether taking $\mathsf{polylog}(Q)$ polynomial multiplications (or equivalently $O(N \mathsf{polylog}(Q) \log(N))$ integer multiplications). Rotation is done via Galois automorphism which also takes $\mathsf{polylog}(Q)$ polynomial multiplications. Multiplication requires a BFV evaluation key for relinearization and rotation requires a BFV rotation key. We assume all keys are correctly and implicitly taken. All operations are operated over the entire plaintext vector $m \in \mathbb{Z}_t^N$. Thus, all messages need to be evaluated using the same polynomial f by default. This is also known as the Single Instruction Multiple Data (SIMD) property of BFV.

Since we use all of these operations as blackboxes, we omit the details and refer the readers to [8, 20, 49, 31].

Short keys. In practice, $\mathsf{sk} \in \mathcal{R}$ is almost always a short vector (i.e., $\|\mathsf{sk}\|_\infty \leq \delta$ for some small δ, e.g., $\delta = 1$ for ternary and binary secret keys). $\mathsf{sk} \in \mathcal{R}$ can be easily represented in $\mathcal{R}_Q$ (e.g., if $\mathsf{sk}[i] = -1$, it is represented as $Q - 1$). Therefore, we directly view $\mathsf{sk} \in \mathcal{R}_Q$ for simplicity. When sk is transformed to $\mathcal{R}_{Q'}$, the transformation is done in the same way. In this paper, we assume the secret key of BFV is ternary (i.e., $\mathsf{sk}[i] \in \{0, 1, -1\}$ for all $i \in [N]$ for $\mathsf{sk} \in \mathcal{R}$), compliant with FHE standard [1], unless otherwise specified.

BFV modulus switching. Similar to the modulus switching procedure described in Section 3.2 for FHEW/TFHE, the modulus switching procedure for BFV: $\mathsf{BFV.ModSwitch}(\mathsf{ct}_{\mathsf{BFV}}, Q')$ is as follow: let $\mathsf{ct}_{\mathsf{BFV}} = (a, b) \in \mathcal{R}_Q$ be a BFV ciphertext with ring dimension N, ciphertext modulus Q and error e. Then, $\mathsf{BFV.ModSwitch}(\mathsf{ct}_{\mathsf{BFV}}, Q') := (Q'/Q)\mathsf{ct}_{\mathsf{BFV}}$ simply takes every coefficient of a, b, divides them by Q, multiplies them by Q', and rounds to the nearest integer.

BFV key switching. BFV key switching is much more complicated than the key switching for LWE ciphertexts introduced above, but essentially for the same functionality. We skip the details here and refer the readers to [31].

4 Binary NAND Gate Bootstrapping

In this section, we show in detail how to construct a batched bootstrapping process for NAND gate, i.e., the batch version of the original FHEW/TFHE bootstrapping procedure as introduced in Section 3.2.

Given $2N$ LWE ciphertexts $\vec{\mathsf{ct}_1} = (\mathsf{ct}_{1,1}, \ldots, \mathsf{ct}_{1,N}), \vec{\mathsf{ct}_2} = (\mathsf{ct}_{2,1}, \ldots, \mathsf{ct}_{2,N})$, encrypting either 0 or 1 with ciphertext modulus q, plaintext modulus $p = 3$ [†], under the same key sk, with error $< \beta = \lfloor q/12 \rfloor$, we want to construct some procedure $\mathsf{Boot}(\vec{\mathsf{ct}_1}, \vec{\mathsf{ct}_2}, \mathsf{btk})$ where btk is some bootstrapping key to be discussed later, and output $(\mathsf{ct}'_1, \ldots, \mathsf{ct}'_N)$, all encrypted under sk with error $< \beta$, such that $\mathsf{Dec}(\mathsf{ct}'_i) = \neg(\mathsf{Dec}(\mathsf{ct}_{1,i}) \wedge \mathsf{Dec}(\mathsf{ct}_{2,i}))$ for $i \in [N]$.

We perform the batched bootstrapping using BFV scheme, parametrized by ring dimension N, ciphertext modulus Q, and plaintext modulus t.

[†] Note that prior works use $p = 4$.

4.1 Bootstrapping Key Generation

We start by discussing what bootstrapping keys we need. Since our construction is fully based on BFV HE scheme, we need the public key of BFV $\mathsf{BFV.pk}$, the evaluation key of BFV $\mathsf{BFV.evk}$ (for relinearization after multiplications), and the rotation key of BFV $\mathsf{BFV.rtk}$ for arbitrary rotations (to perform $\mathsf{BFV.Rotate}(\mathsf{ct}, \mathsf{BFV.rtk}, i)$. For each i, we need to generate a different BFV rotation key. Thus, $\mathsf{BFV.rtk}$ essentially includes multiple keys corresponding to multiple rotation step sizes. We will fix the specific number of rotation keys included in $\mathsf{BFV.rtk}$ later. Besides, we need $\mathsf{bfvct_{sk}} \leftarrow \mathsf{BFV.Enc}(\mathsf{sk})$. Here $\mathsf{sk} \in \mathbb{Z}_q^n$ [‡], is the secret key used to encrypt the inputs $\vec{\mathsf{ct}}_1, \vec{\mathsf{ct}}_2$. Note that the plaintext space of the BFV scheme is $\mathbb{Z}_t^N$, to make it consistent with the modulus of sk, we let $t = q$. Moreover, to make sk a valid input to $\mathsf{BFV.Enc}$, we repeat sk for N/n times and concatenate together [§], i.e. $(\mathsf{sk}||\dots||\mathsf{sk}) \in \mathbb{Z}_t^N$.

All these public keys are generated using a BFV secret key $\mathsf{sk_{BFV}} = \sum_{i\in[N]} s_i X^{i-1} \in \mathcal{R}_Q$, generated independently from the secret key sk (which is used to encrypt the input LWE ciphertexts).

Let $\overrightarrow{\mathsf{sk_{BFV}}} = (s_1, \dots, s_N)$ denotes the vector of the coefficients of $\mathsf{sk_{BFV}}$. With $\overrightarrow{\mathsf{sk_{BFV}}}$ and sk, we generate the key-switching key K that is used to turn an LWE ciphertext encrypted under $\mathsf{sk_{BFV}}$ to an LWE ciphertext encrypted under sk as introduced in Section 3.2.

Based on the CPA security of BFV, all of these keys are not leaking any information about $\mathsf{sk_{BFV}}$ or sk. Thus, our bootstrapping key is $\mathsf{btk} = (\mathsf{BFV.pk}, \mathsf{BFV.evk}, \mathsf{BFV.rtk}, K, \mathsf{bfvct_{sk}})$.

4.2 Pair-wise Summation

Recall that for an LWE ciphertext $(\vec{a}, b)$ encrypting 1 under sk, we have $b - \langle \mathsf{sk}, \vec{a}\rangle \in (\lfloor q/3 \rfloor - \lfloor q/12 \rfloor, \lfloor q/3 \rfloor + \lfloor q/12 \rfloor)$, as $\mathsf{err}(\mathsf{ct}_{j,i}) < \beta = \lfloor q/12 \rfloor$, and encrypting 0 when $b - \langle \mathsf{sk}, \vec{a}\rangle \in (-\lfloor q/12 \rfloor, \lfloor q/12 \rfloor)$. [¶]

Our first step simply adds the two input vectors pair-wisely. For all $i \in [N]$, given $\mathsf{ct}_{1,i} = (\vec{a}_{1,i}, b_{1,i}), \mathsf{ct}_{2,i} = (\vec{a}_{2,i}, b_{2,i})$, compute $\mathsf{ct}_i = (\vec{a}_i, b_i) \leftarrow (\vec{a}_{1,i} + \vec{a}_{2,i}, b_{1,i} + b_{2,i} + \lfloor q/6 \rfloor)$, where the vector addition is done via element-wise addition. (We shift the result by $\lfloor q/6 \rfloor$ to avoid negative numbers for simplicity in the following range analysis.)

Thus, if $\mathsf{Dec}(\mathsf{ct}_{1,i}) = \mathsf{Dec}(\mathsf{ct}_{2,i}) = 1$ we have:

$$\begin{aligned} b_i - \langle \mathsf{sk}, \vec{a}_i \rangle &= (b_{1,i} - \langle \mathsf{sk}, \vec{a}_{1,i}\rangle) + (b_{2,i} - \langle \mathsf{sk}, \vec{a}_{2,i}\rangle) + \lfloor q/6 \rfloor \\ &\in (\lfloor q/3 \rfloor + \lfloor q/3 \rfloor - \lfloor q/12 \rfloor - \lfloor q/12 \rfloor + \lfloor q/6 \rfloor, \lfloor q/3 \rfloor + \lfloor q/3 \rfloor + \lfloor q/12 \rfloor + \lfloor q/12 \rfloor + \lfloor q/6 \rfloor) \\ &\subseteq (2\lfloor q/3 \rfloor, q) \end{aligned}$$

And similarly, if $\mathsf{Dec}(\mathsf{ct}_{1,i}) = \mathsf{Dec}(\mathsf{ct}_{2,i}) = 0$, we have $b_i - \langle \mathsf{sk}, \vec{a}_i \rangle \in (0, \lfloor q/3 \rfloor)$, otherwise $b_i - \langle \mathsf{sk}, \vec{a}_i \rangle \in (\lfloor q/3 \rfloor, 2\lfloor q/3 \rfloor)$.

4.3 Homomorphic Decryption Circuit

Our next step is to homomorphically decrypt all the ct_i's. The regular LWE decryption procedure is as follow:

$$\mathsf{Dec}(\mathsf{sk}, \mathsf{ct} = (\vec{a}, b)) = \left\lceil \frac{b - \langle \vec{a}, \mathsf{sk}\rangle \pmod q}{\alpha} \right\rfloor$$

which has three steps: (1) *inner product*, (2) *subtraction*, and (3) *division and rounding*. As our goal is to compute a NAND gate and output an LWE ciphertext, during the final step, we also need to map the resulting value in $\mathbb{Z}_3$ into $\{0, \lfloor q/3 \rfloor\}$, which is the encoding of $\{0, 1\}$ correspondingly. Thus, the last step is essentially *division, rounding, and NAND mapping.*

[‡] Recall that technically $\mathsf{sk} \in \mathbb{Z}^n$. However, it can be transformed to $\mathbb{Z}_q^n$ easily as long as $||\mathsf{sk}||_\infty \le \lfloor q/2 \rfloor$. Thus, for simplicity, we view $\mathsf{sk} \in \mathbb{Z}_q^n$. Similarly for the BFV secret key below.

[§] For simplicity we assume $N/n \in \mathbb{Z}^+$.

[¶] Note that $-\lfloor q/12 \rfloor$ is simply $q - \lfloor q/12 \rfloor$.

Algorithm 1 Homomorphic Linear Transformation

```
procedure $\mathsf{LT}(\mathsf{BFV.rtk}, A, \mathsf{bfvct})$    ▷ $A \in \mathbb{Z}_t^{N\times n}$
    ▷ $\mathsf{bfvct}$ encrypts a vector $\vec{v} \in \mathbb{Z}_t^n$ by repeating $v$ $N/n$ times and encrypting the concatenation of those $N/n$ repetitions.
    $\mathsf{rt} \leftarrow \sqrt{n}$
        ▷ We assume $\sqrt{n}$ to be an integer for simplicity. For more general $n$'s, see [30] for details.
    for $i \in [\mathsf{rt}]$ do
        $\mathsf{bfvct}_{\mathsf{rot}_i} \leftarrow \mathsf{BFV.Rotate}(\mathsf{bfvct}, \mathsf{BFV.rtk}, i \cdot \mathsf{rt})$
    Initialize BFV ciphertexts $\mathsf{res}_k$, for $k \in [\mathsf{rt}]$, each encrypting 0's
    for $k \in [\mathsf{rt}]$ do
        for $i \in [\mathsf{rt}]$ do
            Construct $\mathsf{tmp} \in \mathbb{Z}_t^N$, such that $\mathsf{tmp}[j] = A[\mathsf{ind_{ct}}][\mathsf{ind_a}]$, where $\mathsf{ind_{ct}} = (j - k) \mod N$, $\mathsf{ind_a} = (j + i \cdot \mathsf{rt}) \mod n$
            $\mathsf{c} \leftarrow \mathsf{tmp} \times \mathsf{bfvct}_{\mathsf{rot}_i}$
            $\mathsf{res}_k \leftarrow \mathsf{res}_k + \mathsf{c}$
    for $i \in [\mathsf{rt} - 1]$ do
        $\mathsf{c} \leftarrow \mathsf{BFV.Rotate}(\mathsf{res}_{\mathsf{rt}-i+1}, \mathsf{BFV.rtk}, 1)$
        $\mathsf{res}_{\mathsf{rt}-i} \leftarrow \mathsf{BFV.Add}(\mathsf{res}_{\mathsf{rt}-i}, \mathsf{c})$
    return $\mathsf{bfvct}' \leftarrow \mathsf{res}_1$
```

One *key property* we use is that BFV homomorphically computes over $\mathbb{Z}_t$ where we set t equals q which is the LWE ciphertext modulus. Therefore, the mod operation over t is automatically performed during all computations and we just need to design the circuit over $\mathbb{Z}_t$ with $t = q$.

Inner product. We start by homomorphically computing the inner product. Let $\mathsf{ct}_i = (\vec{a}_i, b_i)$. To compute $\langle \vec{a}_i, \mathsf{sk}\rangle$ for all $i \in [N]$ is equivalent to compute $A\mathsf{sk}$ where $A = \begin{pmatrix} \vec{a}_1 \\ \vdots \\ \vec{a}_N \end{pmatrix} \in \mathbb{Z}_t^{N\times n}$.

Naively, this matrix multiplication can be computed with N plaintext-by-ciphertext multiplications together with N rotations. However, we improve this by using the baby-step-giant-step technique, which is first introduced in [26] and later improved in [30]. Thus allows us to compute $A\mathsf{sk}$ with N plaintext-by-ciphertext multiplications and just $2\sqrt{N}$ rotations. Note that each rotation requires a specific rotation key generated accordingly. As the BFV key generation is straightforward, we view it as a blackbox and assume all the keys needed are properly included in btk.

We adapt this technique and formally present it in Algorithm 1, which takes a matrix $A \in \mathbb{Z}_t^{N\times n}$, a BFV ciphertext bfvct encrypting a vector $m \in \mathbb{Z}_t^n$, and proper BFV rotation keys, and outputs a BFV ciphertext bfvct' encrypting $(Am)^\intercal \in \mathbb{Z}_t^N$. For the correctness of this algorithm, see [26, 30] for details.

Subtraction. Subtraction can be done by computing $\vec{b} - (A\mathsf{sk})^\intercal$ where $\vec{b} = (b_1, \ldots, b_N)$. Again, as BFV is computing the circuits over $\mathbb{Z}_t$, "mod t" part comes for free.

Division, rounding, and NAND mapping. This step is the most involved component in the decryption procedure. The reason is that BFV only supports multiplication and addition over a finite field , while division and rounding are not supported. Thus, we design a polynomial function over the finite field $\mathbb{Z}_t$ to compute division and rounding.

Recall that if $\mathsf{Dec}(\mathsf{ct}_{1,i}) = \mathsf{Dec}(\mathsf{ct}_{2,i}) = 1$ we have $b_i - \langle \mathsf{sk}, \vec{a}_i\rangle + \lfloor q/6 \rfloor \in (2\lfloor q/3 \rfloor, q)$, where $(\vec{a}_i, b_i) = \mathsf{ct}_i$ computed above in Section 4.2. This is the case that should be mapped to 0 for a NANG gate mapping. Otherwise, the result should be mapped to $\lfloor q/3 \rfloor$ (i.e., the encoding of 1).

We first express this division, rounding, and mapping process as a function $\mathsf{DRaM} : \mathbb{Z}_t \to \mathbb{Z}_t$ (recall that $t = q$):

$$\mathsf{DRaM}(x) = \begin{cases} 0 & \text{if } x \geq 2\lfloor t/3 \rfloor \\ \lfloor t/3 \rfloor & \text{otherwise} \end{cases} \tag{1}$$

where DRaM stands for **D**ivision, **R**ounding, **a**nd **M**apping.

Then, we translate this function into a polynomial function $\mathsf{DRaMpoly} : \mathbb{Z}_t \to \mathbb{Z}_t$ by using the following formula adapted from [29, Equation 2]:

$$\mathsf{DRaMpoly}(x) = \mathsf{DRaM}(0) - \sum_{i=1}^{t-1} x^i \sum_{a=0}^{t-1} \mathsf{DRaM}(a) a^{t-1-i} \ .$$

For any t, $\mathsf{DRaMpoly}$ is then equivalent to DRaM and we formalize it by the following lemma.

Lemma 4.1. Given any prime p, for any function $f : \mathbb{Z}_p \to \mathbb{Z}_p$, let $f'(x) := f(0) - \sum_{i=1}^{t-1} x^i \sum_{a=0}^{t-1} f(a) a^{t-1-i}$ then it holds that for any $x \in \mathbb{Z}_p$, $f(x) = f'(x)$.

For correctness proof of the lemma, we refer the readers to [29, Section 3].

Thus, to evaluate the division, rounding, and mapping, we simply need to perform $\mathsf{BFV.Eval}(\mathsf{bfvct}, \mathsf{DRaMpoly})$, where $\mathsf{bfvct} \leftarrow \vec{b} - \mathsf{LT}(\mathsf{BFV.rtk}, A, \mathsf{bfvct}_{\mathsf{sk}})$ is the BFV ciphertext resulted by homomorphically computing $\vec{b} - (A\mathsf{sk})^\intercal$, where $\vec{b} = (b_1, \ldots, b_N), A = \begin{pmatrix} \vec{a}_1 \\ \vdots \\ \vec{a}_N \end{pmatrix}$, and $\mathsf{bfvct}_{\mathsf{sk}}$ is the BFV ciphertext encrypting $\mathsf{sk} \in \mathbb{Z}_t^n$ (generated as in Section 4.1).

Naively computing the polynomial $\mathsf{DRaMpoly}$ requires $O(t)$ ciphertext-by-ciphertext multiplications, which are used to generate x^i for all $i \in [t]$. However, instead, we use the Paterson-Stockmeyer algorithm [47], reducing to $O(\sqrt{t})$ ciphertext-by-ciphertext multiplications.

4.4 BFV Ciphertext to LWE Ciphertexts

After all the processes above, we obtain a BFV ciphertext $\mathsf{bfvct}_{\mathsf{res}}$ encrypting the message vector $(m_1, \ldots, m_N)$. Here we have $m_i = 0$ if $\mathsf{Dec}(\mathsf{ct}_{1,i}) = \mathsf{Dec}(\mathsf{ct}_{2,i}) = 1$, and $m_i = \lfloor q/3 \rfloor$ otherwise, where $\mathsf{ct}_{1,i}, \mathsf{ct}_{2,i}$ for $i \in [N]$ are input ciphertexts. Recall that $\mathsf{bfvct}_{\mathsf{res}}$ is in the encoded form introduced in Section 3.3.

Our next step is to expand this single BFV ciphertext encrypting N messages into N LWE ciphertexts, each encrypting a single m_i. To do this, we first transform $\mathsf{bfvct}_{\mathsf{res}}$ to a BFV ciphertext encrypting $m(X) = \sum_i m_i X^{i-1}$. In other words, we decode the encoded messages homomorphically. Then, we extract N LWE ciphertexts each encrypting a single coefficient m_i, i.e., switching a Ring-LWE ciphertext into LWE ciphertexts.

Homomorphic decoding. Recall that in BFV, to encrypt a vector of messages $(m_1, \ldots, m_N) \in \mathbb{Z}_t^N$, we first use canonical embedding to encode them into a polynomial. In more detail, let $m(X) := \sum_{i \in [N]} m_i X^{i-1}$, we construct a polynomial $y(X) = \sum_{i \in [N]} y_i X^{i-1}$, where $y_i = m(\zeta_i)$, where ζ being the $2N$-th primitive root of unity of t, and $\zeta_i := \zeta^{3^i}$. Thus, a ciphertext $\mathsf{bfvct}_{\mathsf{res}}$ encrypting $(m_1, \ldots, m_N)$ encrypts the polynomial $y(X)$.

To revert this process, we can homomorphically compute $\mathsf{bfvct}'_{\mathsf{res}} \leftarrow \mathsf{bfvct}_{\mathsf{res}} U^\intercal$, where

$$U := \begin{pmatrix} 1 & \zeta_0 & \zeta_0^2 & \cdots & \zeta_0^{N-1} \\ 1 & \zeta_1 & \zeta_1^2 & \cdots & \zeta_1^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \zeta_{\frac{N}{2}-1} & \zeta_{\frac{N}{2}-1}^2 & \cdots & \zeta_{\frac{N}{2}-1}^{N-1} \\ 1 & \bar{\zeta}_0 & \bar{\zeta}_0^2 & \cdots & \bar{\zeta}_0^{N-1} \\ 1 & \bar{\zeta}_1 & \bar{\zeta}_1^2 & \cdots & \bar{\zeta}_1^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \bar{\zeta}_{\frac{N}{2}-1} & \bar{\zeta}_{\frac{N}{2}-1}^2 & \cdots & \bar{\zeta}_{\frac{N}{2}-1}^{N-1} \end{pmatrix} \in \mathbb{Z}_t^{N \times N}$$

where $\bar{\zeta}_j := \zeta_j^{-1}$, and the resulting $\mathsf{bfvct}'_{\mathsf{res}}$ is thus encrypting the polynomial $m(X) = \sum_i m_i X^{i-1}$.

This process is first introduced as the $\mathsf{SlotToCoeff}$ procedure in [13] for the CKKS HE scheme, and we adapt it to the BFV scheme.

Ring-LWE to LWE. Now $\mathsf{bfvct}'_{\mathsf{res}} = (a_{\mathsf{bfv}}, b_{\mathsf{bfv}}) \in \mathcal{R}_Q^2$ and we have $b_{\mathsf{bfv}} - a_{\mathsf{bfv}}\mathsf{sk}_{\mathsf{BFV}} \approx \lfloor Q/t \rceil\, m$ where m is the polynomial $m(X)$ defined above with all the coefficients being either 0 or $\lfloor t/3 \rceil$. Our goal is to obtain N LWE ciphertexts $(\mathsf{ct}'_1 = (\vec{a}'_1, b'_1), \ldots, \mathsf{ct}'_N = (\vec{a}'_N, b'_N)) \in \mathbb{Z}_Q^{N\cdot(N+1)}$ such that $b'_i - \langle \vec{a}'_i, \vec{s} \rangle \approx \lfloor Q/t \rceil\, m_i$, where $\vec{s} = (s_1, \ldots, s_N)$ for $\vec{s}$ is the coefficient vector of $\mathsf{sk}_{\mathsf{BFV}}$.

This can be done by using the $\mathsf{SampleExtract}$ procedure in [15]. With ciphertext $\mathsf{bfvct}'_{\mathsf{res}} = (a_{\mathsf{bfv}}(X) = \sum_{i\in[N]} a_{\mathsf{bfv}_i} X^{i-1}, b_{\mathsf{bfv}}(X) = \sum_{i\in[N]} b_{\mathsf{bfv}_i} X^{i-1})$, we achieve the extraction from RLWE ciphertext $\mathsf{bfvct}'_{\mathsf{res}}$ to LWE ciphertexts $(\vec{a}'_i, b'_i)_{i\in[N]}$ by setting $\vec{a}'_i = (\alpha_{i,1}, \ldots, \alpha_{i,N})$ where $\alpha_{i,j} \leftarrow a_{\mathsf{bfv}_{i-j+1}}$ if $j \le i$ and $\alpha_{i,j} \leftarrow -a_{\mathsf{bfv}_{N-j+i+1}}$ if $j > i$, and $b'_i = b_{\mathsf{bfv}_i}$ for $i \in [N]$.

This procedure is formalized by the following lemma:

Lemma 4.2. For any ring elements $a(X) = \sum_{i\in[N]} a_i X^{i-1}, s(X) = \sum_{i\in[N]} s_i X^{i-1} \in \mathcal{R}_q$, let $b(X) = a \cdot s = \sum_{i\in[N]} b_i X^{i-1} \in \mathcal{R}_q$, it holds that $b_i = \sum_{j\in[N]} a'_{i,j} \cdot s_j \bmod q$ where $a'_{i,j} = a_{i-j+1}$ if $j \le i$ and $a'_{i,j} = q - a_{N-j+i+1}$ otherwise.

The correctness of the lemma is straightforward so we omit the details. The procedure is presented as $\mathsf{Extract}$ in Algorithm 2.

Key switching and modulus switching. After all these steps, we now obtain $(\mathsf{ct}'_1 = (\vec{a}'_1, b'_1), \ldots, \mathsf{ct}'_N = (\vec{a}'_N, b'_N)) \in \mathbb{Z}_Q^{N\cdot(n+1)}$ encrypting the NAND results under $\vec{s}$. To obtain N LWE ciphertexts encrypting the NAND results under sk with ciphertext modulus t, two final steps are needed: (1) use key switching $\mathsf{KeySwitch}$ introduced in Section 3.2 with the key switching key K generated as in Section 4.1 to change ct'_i into ciphertexts encrypted under sk instead of $\vec{s}$; (2) use modulus switching to change the ciphertext modulus from Q to t.

We thus complete our algorithm, and formally demonstrate the entire procedure in Algorithm 2.

Theorem 4.3. Let $(\mathsf{pp}_{\mathsf{lwe}} = (n, q, p = 3, \mathcal{D}_1, \chi_\sigma), \mathsf{pp}_{\mathsf{bfv}} = (N, Q, Q', t = q, \mathcal{D}_2, \chi_{\sigma'}), \mathsf{sk}, \mathsf{btk}) \leftarrow \mathsf{Setup}(1^\lambda)$ in Algorithm 2, for any input LWE ciphertexts $\vec{\mathsf{ct}_1} = (\mathsf{ct}_{1,1}, \ldots, \mathsf{ct}_{1,N}), \vec{\mathsf{ct}_2} = (\mathsf{ct}_{2,1}, \ldots, \mathsf{ct}_{2,N})$ parameterized by $\mathsf{pp}_{\mathsf{lwe}}$, encrypting $\{0,1\}$ under sk, and $\mathsf{err}(\mathsf{ct}_{i,j}) < \lfloor q/12 \rfloor$ for all $i \in [2], j \in [N]$; let $(\mathsf{ct}_{\mathsf{out}1}, \ldots, \mathsf{ct}_{\mathsf{out}N}) \leftarrow \mathsf{Boot}(\vec{\mathsf{ct}_1}, \vec{\mathsf{ct}_2}, \mathsf{btk})$ in Algorithm 2, it holds that: $\mathsf{Dec}(\mathsf{ct}_{\mathsf{out}i}) = \mathrm{NAND}(\mathsf{Dec}(\mathsf{ct}_{1,i}), \mathsf{Dec}(\mathsf{ct}_{2,i}))$, $\Pr[\mathsf{err}(\mathsf{ct}_{\mathsf{out}i}) < \lfloor q/12 \rfloor] \ge 1 - \mathsf{negl}(\lambda)$, and $\mathsf{ct}_{\mathsf{out}i}$ is encrypted under secret key sk, for all $i \in [N]$.

Proof sketch. We show this result from basic correctness (i.e., achieving NAND gate functionality) and noise analysis.

- (Correctness) Correctness (i.e., $\mathsf{Dec}(\mathsf{ct}_{\mathsf{out}i}) = \mathrm{NAND}(\mathsf{Dec}(\mathsf{ct}_{1,i}), \mathsf{Dec}(\mathsf{ct}_{2,i})))$ is intuitive. We first homomorphically evaluate a circuit using BFV, which includes the evaluation of Dec function of the LWE ciphertexts and the NAND mapping. Then we extract N LWE ciphertexts out from the resulting BFV ciphertext. The correctness of circuit design is given by Lemma 4.1, the correctness of the evaluation of the circuit is given by the correctness of BFV (and condition (4) in Setup), and the correctness of extraction (including RLWE to LWE transformation, key switching, and modulus switching) is guaranteed by Lemmas 3.1, 3.2 and 4.2.

- (Noise Analysis) By the correctness of BFV, with bfvct_5 obtained as on line 33 in Algorithm 2, we have $\mathsf{err}(\mathsf{bfvct}_5) \le (Q'/t)/2$, and this error is from Gaussian distribution χ_{σ_5} with mean 0 and standard deviation σ_5. The key switching procedure introduces an extra noise with a standard deviation of $\sigma_{\mathsf{ks}} = \sigma_{\mathsf{ksk}} N d_{\mathsf{ks}}$ where σ_{ksk} is the error standard deviation of the key switching key and $\sigma_{\mathsf{ksk}} = \sigma'$, where σ' is selected on line 4 in Algorithm 2, and d_{ks} is a key switching parameter, bounded by $\log(N)$ (see details in Section 3.2). Thus, after modulus switching, the resulting ciphertexts have error of standard deviation $\sigma = \sqrt{\left(\frac{t}{Q'}(\sigma_5 + \sigma_{\mathsf{ks}})\right)^2 + \sigma_{\mathsf{ms}}{}^2}$, where $\sigma_{\mathsf{ms}} = \sqrt{\frac{\|\mathsf{sk}\|_2^2+1}{3}}$ is the modulus switching error standard deviation, and $\|\mathsf{sk}\|_2$ is the ℓ-2 norm of the secret key of the LWE ciphertexts)
  Thus, by condition (6) on line 8 in Algorithm 2, $\mathsf{err}(\mathsf{ct}_{\mathsf{out}i}) < \lfloor q/12 \rfloor, \forall i \in [N]$ with overwhelming probability.

Algorithm 2 Batched FHEW/TFHE bootstrapping for NAND gate

```
1:  procedure $\mathsf{Setup}(1^\lambda)$
2:      Select $(n, q, \mathcal{D}_1, \chi_\sigma, N, Q, \mathcal{D}_2\chi_{\sigma'}, Q')$ minimizing the cost of the entire homomorphic circuit evaluation
        and also satisfying:
3:      (1) $\mathsf{LWE}_{n,q,\mathcal{D}_1,\chi_\sigma}$ holds, and $\Pr[e < \lfloor q/12 \rceil] \geq 1 - \mathsf{negl}(\lambda)$ where $e \leftarrow \chi_\sigma$.
4:      (2) $\mathsf{RLWE}_{N,Q,\mathcal{D}_2,\chi_{\sigma'}}$ hold.
5:      (3) $Q' \leq Q, \mathsf{LWE}_{n,Q',\mathcal{D}_1,\chi_{\sigma'}}$ holds.
6:      (4) BFV with parameters $N, Q, t = q, \mathcal{D}_2, \chi_{\sigma'}$ is enough to evaluate the circuit in procedure $\mathsf{Boot}$
7:      (5) $\mathsf{LWE}_{n,q,\mathcal{D}_1,\chi_{\sigma_{\mathsf{res}}}}$ holds, where $\sigma_{\mathsf{res}} = \sqrt{(\frac{t}{Q'}(\sigma_5 + \sigma_{\mathsf{ks}}))^2 + {\sigma_{\mathsf{ms}}}^2}$, where $\sigma_5$ is the noise distribution
        standard deviation of line 33
8:      (6) $\Pr[e' < \lfloor q/12 \rceil] \geq 1 - \mathsf{negl}(\lambda)$ where $e' \leftarrow \chi_{\sigma_{\mathsf{res}}}$        ▷ $\sigma_{\mathsf{ks}}, \sigma_{\mathsf{ms}}$ are as introduced in Section 3.2.
9:      $\mathsf{sk} \leftarrow \mathcal{D}_1$
10:     Generate BFV secret key $\mathsf{sk}_{\mathsf{BFV}} \leftarrow \mathcal{D}_2$.
11:     Generate bootstrapping keys $\mathsf{btk} = (\mathsf{BFV.pk}, \mathsf{BFV.evk}, \mathsf{BFV.rtk}, K, \mathsf{bfvct}_{\mathsf{sk}})$ as in Section 4.1.
12:                                                                ▷ $K$ generated with $\mathsf{sk}_{\mathsf{BFV}}, \mathsf{sk}, Q', \chi_{\sigma'}$.
13:     return $(\mathsf{pp} = \mathsf{pp}_{\mathsf{lwe}} = (n, q, p = 3, \mathcal{D}_1, \chi_\sigma), \mathsf{pp}_{\mathsf{bfv}} = (N, Q, Q', t = q, \mathcal{D}_2, \chi_{\sigma'}), \mathsf{sk}, \mathsf{btk})$
14: procedure $\mathsf{Extract}(\mathsf{bfvct} = (a_{\mathsf{bfv}}, b_{\mathsf{bfv}}))$        ▷ Extract $N$ LWE ciphertexts from one BFV ciphertext
15:     Initialize $\mathsf{ct}_i = (\vec{a}_i, b_i)$ for $i \in [N]$
16:     for $i \in [N]$ do
17:         for $j \in [N]$ do
18:             if $j \leq i$ then
19:                 $\vec{a}_i[j] \leftarrow a_{\mathsf{bfv}}[i - j + 1]$
20:             else
21:                 $\vec{a}_i[j] \leftarrow -a_{\mathsf{bfv}}[N - j + i + 1]$
22:                                  ▷ Negative number is still in $\mathbb{Z}_q$ where $q$ is the $\mathsf{bfvct}$ ciphertext modulus
23:         $b_i = b_{\mathsf{bfv}_i}$
24:     return $(\mathsf{ct}_1, \ldots, \mathsf{ct}_N)$
25: procedure $\mathsf{Boot}((\mathsf{ct}_{1,1}, \ldots, \mathsf{ct}_{1,N}), (\mathsf{ct}_{2,1}, \ldots, \mathsf{ct}_{2,N}), \mathsf{btk} = (\mathsf{BFV.pk}, \mathsf{BFV.evk}, \mathsf{BFV.rtk}, K, \mathsf{bfvct}_{\mathsf{sk}}))$
26:                                                        ▷ $\mathsf{ct}_{j,i} = (\vec{a}_{j,i}, b_{j,i})$ for $j \in [2], i \in [N]$
27:     Compute $\mathsf{ct}_i = (\vec{a}_i, b_i) \leftarrow (\vec{a}_{1,i} + \vec{a}_{2,i}, b_{1,i} + b_{2,i} + \lfloor q/6 \rceil), \forall i \in [N]$
28:     $A \leftarrow (\vec{a}_1^\intercal, \ldots, \vec{a}_N^\intercal)$
29:     $\mathsf{bfvct}_1 \leftarrow \mathsf{LT}(\mathsf{BFV.rtk}, A, \mathsf{bfvct}_{\mathsf{sk}})$
30:     $\mathsf{bfvct}_2 \leftarrow (b_1, \ldots, b_N) - \mathsf{bfvct}_1$                ▷ Homomorphically computes $\vec{b} - \mathsf{sk}A$
31:     $\mathsf{bfvct}_3 \leftarrow \mathsf{BFV.Eval}(\mathsf{BFV.evk}, \mathsf{bfvct}_2, \mathsf{DRaMpoly})$
32:     $\mathsf{bfvct}_4 \leftarrow \mathsf{LT}(\mathsf{BFV.rtk}, U^\intercal, \mathsf{bfvct}_3)$           ▷ Recall that $U$ is the matrix defined for packing
33:     $\mathsf{bfvct}_5 \leftarrow \mathsf{BFV.ModSwitch}(\mathsf{bfvct}_4, Q')$          ▷ $\mathsf{BFV.ModSwitch}$ is described in Section 3.3
34:     $(\mathsf{ct}'_1, \ldots, \mathsf{ct}'_N) \leftarrow \mathsf{Extract}(\mathsf{bfvct}_5)$
35:     $\mathsf{ct}''_i \leftarrow \mathsf{KeySwitch}(K, \mathsf{ct}'_i), \forall i \in [N]$
36:                                  ▷ $\mathsf{KeySwitch}$ is defined in Section 3.2, and $K$ is the key switching key
37:     $\mathsf{ct}_{\mathsf{out}_i} \leftarrow \mathsf{ModSwitch}(\mathsf{ct}''_i, q), \forall i \in [N]$             ▷ $\mathsf{ModSwitch}$ is defined in Section 3.2
38:     return $(\mathsf{ct}_{\mathsf{out}_1}, \ldots, \mathsf{ct}_{\mathsf{out}_N})$
```

□

Efficiency analysis. As explained, the BFV circuit needs $O(\sqrt{n}+\sqrt{N})$ rotations, $O(\sqrt{t})$ ciphertext-by-ciphertext multiplications, and $n+t+N$ plaintext-by-ciphertext multiplications, with multiplicative depth $\ell = \log(t)+3$ (where we have $\log(t)+1$ levels for polynomial evaluation, one level for the inner product, and one level for Ring LWE to LWE extraction). Note that all these costs are amortized over N LWE ciphertexts bootstrapping.

All the homomorphic operations take at most $O(\mathsf{poly}(\ell))$ polynomial multiplications (see Section 3.3). The only constraint for t is that $t > 2N+1$ to guarantee that there is a primitive $2N$-th root of unity. Thus, the total cost is $\tilde{O}(N)$ polynomial multiplications per bootstrapping for N LWE ciphertexts. The amortized cost is thus quasi-constant number of polynomial multiplications (which can be done with $O(N\log(N))$ $\mathbb{Z}_Q$ operations using NTT). [‖]

Security analysis. Security analysis, on the other hand, is more involved. By condition (1), the input ciphertexts are semantically secure. To make the whole process secure, we need to make sure that the ciphertexts $\mathsf{bfvct}_1, \mathsf{bfvct}_2, \mathsf{bfvct}_3, \mathsf{bfvct}_4, \mathsf{bfvct}_5, (\mathsf{ct}'_i)_{i\in[N]}$ are all semantically secure. These ciphertexts are secure as long as the keys in $\mathsf{btk} = (\mathsf{BFV.pk}, \mathsf{BFV.evk}, \mathsf{BFV.rtk}, K, \mathsf{bfvct}_{\mathsf{sk}})$ are secure (as all these ciphertexts are obtained by performing operations over the input ciphertexts using these keys). By condition (2) on line 4 in Algorithm 2, together with the semantic security of BFV and the security of key switching process, $\mathsf{bfvct}_1, \mathsf{bfvct}_2, \mathsf{bfvct}_3, \mathsf{bfvct}_4, \mathsf{bfvct}_5, (\mathsf{ct}'_i)_{i\in[N]}$ are semantically secure. Based on the security of BFV, $\mathsf{BFV.pk}, \mathsf{BFV.evk}, \mathsf{BFV.rtk}, \mathsf{bfvct}_{\mathsf{sk}}$ are all secure. By condition (3) on line 5, K is also secure. Thus, the whole process is secure.

4.5 Optimizations

Efficiency of DRaMpoly. For a function

$$f(x) = \begin{cases} 0 & \text{if } x \in (-r, r) \\ y & \text{otherwise} \end{cases}$$

where $r \in [2, \lfloor t/2 \rfloor]$ and $y \in \mathbb{Z}_t$, the function $f'(X) = f(0) - \sum_{i=1}^{t-1} X^i \sum_{a=0}^{t-1} f(a) a^{t-1-i}$ has about half of its coefficients being 0. This means that when homomorphically evaluating f', only half of the plaintext-by-ciphertext multiplications are needed, and only half of the powers are needed, which means fewer ciphertext-by-ciphertext multiplications. Thus, we modify DRaM to be

$$\mathsf{DRaM}(x) = \begin{cases} 0 & \text{if } x \in (\lfloor -t/6 \rfloor, \lfloor t/6 \rfloor) \\ \lfloor t/3 \rfloor & \text{otherwise} \end{cases}$$

and shift the $\mathsf{ct}_i = (\vec{a}_i, b_i)$ (from Section 4.2) by $-(2\lfloor t/3 \rfloor + \lfloor t/6 \rfloor)$ (i.e., $b_i \leftarrow b_i - 2\lfloor q/3 \rfloor - \lfloor t/6 \rfloor$). This reduces the complexity of evaluating $\mathsf{DRaMpoly}$ while remains everything else the same (recall that $t = q$).

Generating rotations in advance. Note that for line 29 in Algorithm 2, we need to rotate $\mathsf{bfvct}_{\mathsf{sk}}$ $\sqrt{n}$ times, as on line 6 in Algorithm 1. However, instead of doing the rotations when evaluating Boot, we can compute those rotations during btk generation and include $\mathsf{bfvct}_{\mathsf{rot}_i}$ in btk for all $i \in [\sqrt{n}]$. This can save $\sqrt{n}$ rotations during bootstrapping.

Level-specific rotation keys. With the optimization above, we only need $\sqrt{n}$ rotation keys with full level. After the deep circuit evaluation of $\mathsf{DRaMpoly}$, we also need rotation keys to compute line 31 in Algorithm 2. Instead of generating the rotation keys with modulus Q, we modulus switch bfvct_3 to modulus Q' and generate rotation keys with modulus $Q' \ll Q$ to greatly reduce the bootstrapping key size.

[‖] LWE ciphertexts addition has the cost of $O(1)$ $\mathbb{Z}_q$ operations per LWE ciphertext. LWE key switching has the cost of $\tilde{O}(N)$ $\mathbb{Z}_{Q'}$ operations per LWE ciphertext. LWE modulus switching has the cost of $O(n)$ $\mathbb{Z}_{Q'}$ operations per LWE ciphertext. Thus, their costs do not affect the asymptotic behavior. Note that the prior works (e.g., [44, 37]) use a similar way to compute the asymptotic costs. Concretely, their costs are also much smaller than the BFV circuit evaluation.

Using BFV key switching and modulus switching. Note that BFV also supports key switching procedure and modulus switching procedure. Hence, instead of performing these two procedures for each extracted LWE ciphertext, we process directly on the single BFV ciphertext. We first create a polynomial $s'(X) = \sum_{i \in [N]} s'_i X^{i-1} \in \mathcal{R}_Q$ where $s'_i = \mathsf{sk}[i]$ if $i \leq n$ and $s'_i = 0$ if $i > n$, for $i \in [N]$. [**] We use $s'(X)$ as the new key to generate the BFV key switching key K together with $\mathsf{sk}_{\mathsf{BFV}}$. The modulus switching procedure remains the same. The security guarantee of BFV key switching is the same as ring switching introduced in [22] as long as n is a power-of-two. Since the key switching procedure and modulus switching procedure are relatively fast, especially for modulus switching, the runtime may not be majorly affected.

4.6 Additional Discussion

Further tuning *n*. Note that as on line 5 in Algorithm 2, we need $\mathsf{LWE}_{n,Q',\mathcal{D}_1,\chi_{\sigma'}}$ to hold. However, since Q' is relatively large, although σ' is also huge, n might need to be relatively large (e.g., concretely 1024). While this can be sufficient for many applications, we introduce the following way to even reduce n.

Suppose we choose some n such that $\mathsf{LWE}_{n,Q',\mathcal{D}_1,\chi_{\sigma_5}}$ breaks. To work around this issue, we introduce an intermediate $n' > n$ and first perform a $\mathsf{KeySwitch}$ to the intermediate secret key with length n', such that $\mathsf{LWE}_{n',Q',\mathcal{D}_1,\chi_{\sigma_5}}$ holds. Then, we modulus switch the result to $q' < Q'$, with the error standard deviation $\sigma_{\mathsf{tmp}} = \sqrt{(\frac{q'}{Q'}(\sigma_5 + \sigma_{\mathsf{ks}}))^2 + {\sigma_{\mathsf{ms}}}^2}$, such that $\mathsf{LWE}_{n,q',\mathcal{D}_1,\chi_{\sigma_{\mathsf{tmp}}}}$ holds. Finally, we perform another key switching to n and modulus switching to q. The resulted error distribution standard deviation is then $\sqrt{(t/q')(\sigma_{\mathsf{tmp}} + {\sigma_{\mathsf{ks}}'}^2) + {\sigma_{\mathsf{ms}}'}^2}$, which can be dominated by ${\sigma_{\mathsf{ms}}}'$ with careful parameter choosing. One can of course repeat this intermediate step for arbitrary times to find an optimal n.

Use BGV instead of BFV. Since BGV also evaluates circuits over $\mathbb{Z}_t$, our construction can use BGV instead of BFV. Note that BGV encodes messages using least significant bits (LSBs), and FHEW/TFHE ciphertexts (i.e., LWE ciphertexts) encrypt messages using most significant bits (MSBs). Therefore, can simply use the technique introduced in [3, Appendix A] to convert a BGV ciphertext to a BFV ciphertext before switching back to the LWE ciphertexts. This minor change does not affect the overall functionality or the security analysis.

5 Multi-binary-gate Bootstrapping

NAND gate itself is a universal gate, and thus our bootstrapping for NAND gate evaluation already achieves the functionality requirement of FHE. However, the efficiency is still limited.

To evaluate a circuit, one needs to first translate the circuit to have only NAND gates. This might introduce a lot of overhead on the circuit size. Moreover, it is restrictive that all the N pairs of input LWE ciphertexts are of the same gate.

Thus, in this section, we propose a construction to evaluate an *arbitrary* binary logic gate (including OR, NOR, AND, NAND, XOR, XNOR). Moreover, the batched bootstrapping procedure can take N different gates, and evaluate the N pairs of LWE ciphertexts with respect to these N input gates in parallel, instead of evaluating the same gate for all the N pairs of input ciphertexts. This enhanced flexibility of our scheme does not introduce any overhead [††].

5.1 Construction

Recall that to evaluate the NAND gate, at a high level, we proceed as follows: given two bits $\gamma_1, \gamma_2 \in \{0, 1\}$, step (1): lift them into $\mathbb{Z}_3$, and compute $r \leftarrow \gamma_1 + \gamma_2 \mod 3$; step (2): if $r = 2$, $\gamma' \leftarrow 0$, otherwise $\gamma' \leftarrow 1$. This procedure gives us $\gamma' = \text{NAND}(\gamma_1, \gamma_2)$ as expected.

[**] Recall that sk is ternary so it can be tranformed in $\mathbb{Z}_q$ easily

[††] For XOR and XNOR, prior constructions have an extra overhead in terms of error, as instead of $\mathsf{ct}_1 + \mathsf{ct}_2$, they need to perform $2(\mathsf{ct}_1 - \mathsf{ct}_2)$ before applying bootstrapping. We refer the readers to [43, Sec 3.2] for details.

Recall that step (2) is performed homomorphically using BFV. Since all the slots in a BFV ciphertext need to be evaluated using the same polynomial function by the SIMD nature of BFV, one main challenge is that we cannot modify step (2), i.e., step (2) needs to be shared among all different gates. Hence, our construction focuses on modifying step (1) and adding a step (3).

OR gate. For the OR gate, we change step (1) to the following: compute $r \leftarrow (\gamma_1 + \gamma_2) - 1 \mod 3$. In this case, if γ_1 and γ_2 are both 0, r is 2, and we get $\gamma' = 0$ in step (2). Otherwise, r is 0 or 1, and we get $\gamma' = 1$.

XNOR gate. For the XNOR gate, we change step (1) to the following: compute $r \leftarrow (\gamma_1 + \gamma_2) - 2 \mod 3$. In this case, if only one of γ_1 and γ_2 is 1, r is 2, and we get $\gamma' = 0$ as needed. Otherwise, r is 0 or 1, and $\gamma' = 1$.

NOR, AND, XOR gates. For those three gates, we add a step (3). We first evaluate the result γ' as OR, NAND, and XNOR gate correspondingly using steps (1) and (2). We then perform $\gamma' \leftarrow 1 - \gamma'$, as NOR, AND, and XOR are simply negations of the result of OR, NAND, XNOR.

Translation to LWE ciphertexts. Recall that our $\mathsf{DRaM}(x)$ function outputs 0 when $x \in (2\lfloor q/3 \rfloor, q)$ and outputs $\lfloor q/3 \rfloor$ otherwise (see Eq. (1)). We show how to evaluate the modification of step (1) described above in $\mathbb{Z}_q$ for different gates, such that this DRaM function in step (2) could be shared.

- (OR gate) Given a pair of ciphertexts $(\mathsf{ct}_1 = (\vec{a}_1, b_1), \mathsf{ct}_2 = (\vec{a}_2, b_2))$, compute $\mathsf{ct} = (\vec{a}, b) \leftarrow (\vec{a}_1 + \vec{a}_2, b_1 + b_2 - \lfloor q/6 \rfloor)$. In this case, iff $\mathsf{Dec}(\mathsf{ct}_1) = 0$ and $\mathsf{Dec}(\mathsf{ct}_2) = 0$, we have $b - \langle \vec{a}, \mathsf{sk} \rangle \in (2\lfloor q/3 \rfloor, q)$.
- (XNOR gate) Given a pair of ciphertexts $(\mathsf{ct}_1 = (\vec{a}_1, b_1), \mathsf{ct}_2 = (\vec{a}_2, b_2))$, compute $\mathsf{ct} = (\vec{a}, b) \leftarrow (\vec{a}_1 + \vec{a}_2, b_1 + b_2 - \lfloor q/3 \rfloor - \lfloor q/6 \rfloor)$. In this case, iff $\mathsf{Dec}(\mathsf{ct}_1) = 1 \wedge \mathsf{Dec}(\mathsf{ct}_2) = 0$ or $\mathsf{Dec}(\mathsf{ct}_1) = 0 \wedge \mathsf{Dec}(\mathsf{ct}_2) = 1$, we have $b - \langle \vec{a}, \mathsf{sk} \rangle \in (2\lfloor q/3 \rfloor, q)$.
- (Negation) After obtaining $\mathsf{ct} = (\vec{a}, b)$ encrypting γ', which is either $\lfloor q/3 \rfloor$ or 0, compute $\mathsf{ct} \leftarrow (-\vec{a}, \lfloor q/3 \rfloor - b)$.

Combining all these, we construct a more general binary gate bootstrapping. We formally show the entire procedure in Algorithm 3.

Theorem 5.1. Let $(\mathsf{pp_{lwe}} = (n, q, p = 3, \mathcal{D}_1, \chi_\sigma), \mathsf{pp_{bfv}} = (N, Q, Q', t = q, \mathcal{D}_2, \chi_{\sigma'}), \mathsf{sk}, \mathsf{btk}) \leftarrow \mathsf{Setup}(1^\lambda)$ in Algorithm 3, for any input LWE ciphertexts $\vec{\mathsf{ct}_1} = (\mathsf{ct}_{1,1}, \ldots, \mathsf{ct}_{1,N}), \vec{\mathsf{ct}_2} = (\mathsf{ct}_{2,1}, \ldots, \mathsf{ct}_{2,N})$ parameterized by $\mathsf{pp_{lwe}}$, encrypting $\{0, 1\}$ under sk, and $\mathsf{err}(\mathsf{ct}_{i,j}) < \lfloor q/12 \rfloor$ for all $i \in [2], j \in [N]$ and a vector of gates $(g_1, \ldots, g_N) \in \{\text{OR, NOR, AND, NAND, XOR, XNOR}\}^N$; let $(\mathsf{ct_{out}}_1, \ldots, \mathsf{ct_{out}}_N) \leftarrow \mathsf{Boot}(\vec{\mathsf{ct}_1}, \vec{\mathsf{ct}_2}, (g_i)_{i \in [N]}, \mathsf{btk})$ in Algorithm 3, it holds that: $\mathsf{Dec}(\mathsf{ct_{out}}_i) = g_i(\mathsf{Dec}(\mathsf{ct}_{1,i}), \mathsf{Dec}(\mathsf{ct}_{2,i}))$, $\Pr[\mathsf{err}(\mathsf{ct_{out}}_i) < \lfloor q/12 \rfloor] \geq 1 - \mathsf{negl}(\lambda)$, and $\mathsf{ct_{out}}_i$ is encrypted under secret key sk, for all $i \in [N]$.

Proof sketch. Correctness and noise analysis follow similarly as in the proof of Theorem 4.3. □

Efficiency and security analysis. This remains exactly the same as the NAND gate analysis in Section 4.4. The costs of preprocessing and postprocessing the LWE ciphertexts are only at most $O(n)$ $\mathbb{Z}_q$ operations, and thus do not affect the asymptotic behavior. Concretely, their costs are also much smaller than the BFV circuit evaluation.

Optimizations. All the optimizations introduced in Section 4.5 can still be applied in a similar way.

6 Functional Bootstrapping for Arbitrary Functions

In this section, we discuss an even more general bootstrapping: functional bootstrapping for arbitrary function evaluation. At a high level, functional bootstrapping allows one to evaluate an arbitrary function over an FHE ciphertext without increasing the error of the FHE ciphertext. More formally, the process takes a ciphertext encrypting $m \in \mathbb{Z}_p$ with error $< \left\lfloor \frac{q}{2p} \right\rfloor$, and outputs a ciphertext encrypting $m' \leftarrow f(m)$ for an arbitrary function $f : \mathbb{Z}_p \rightarrow \mathbb{Z}_p$, with the error of the output ciphertext also $< \left\lfloor \frac{q}{2p} \right\rfloor$.

Algorithm 3 Batched FHEW/TFHE bootstrapping for arbitrary binary gates

```
1: procedure Setup($1^\lambda$)
2:     Select $(n, q, \mathcal{D}_1, \chi_\sigma, N, Q, \mathcal{D}_2 \chi_{\sigma'}, Q')$ minimizing the cost of the entire homomorphic circuit evaluation and also satisfying:
3:     (1) $\mathsf{LWE}_{n,q,\mathcal{D}_1,\chi_\sigma}$ holds, and $\Pr[e < \lfloor q/12 \rfloor] \geq 1 - \mathsf{negl}(\lambda)$ where $e \leftarrow \chi_\sigma$.
4:     (2) $\mathsf{RLWE}_{N,Q,\mathcal{D}_2,\chi_{\sigma'}}$ hold.
5:     (3) $Q' \leq Q, \mathsf{LWE}_{n,Q',\mathcal{D}_1,\chi_{\sigma'}}$ holds.
6:     (4) BFV with parameters $N, Q, \mathcal{D}_2, t = q, \chi_{\sigma'}$ is enough to evaluate the circuit in procedure Boot
7:     (5) $\mathsf{LWE}_{n,q,\mathcal{D}_1,\chi_{\sigma_{\mathsf{res}}}}$ holds, where $\sigma_{\mathsf{res}} = \sqrt{(\frac{t}{Q'}(\sigma_5 + \sigma_{\mathsf{ks}}))^2 + {\sigma_{\mathsf{ms}}}^2}$, where $\sigma_5$ is the noise distribution standard deviation of line 35
8:     (6) $\Pr[e' < \lfloor q/12 \rfloor] \geq 1 - \mathsf{negl}(\lambda)$ where $e' \leftarrow \chi_{\sigma_{\mathsf{res}}}$
9:                                             ▷ $\sigma_{\mathsf{ks}}, \sigma_{\mathsf{ms}}$ are as introduced in Section 3.2.
10:    $\mathsf{sk} \leftarrow_\$ \{-1, 0, 1\}^n$
11:    Generate BFV secret key $\mathsf{sk}_{\mathsf{BFV}} \leftarrow \mathcal{D}_2$.
12:    Generate bootstrapping keys $\mathsf{btk} = (\mathsf{BFV.pk}, \mathsf{BFV.evk}, \mathsf{BFV.rtk}, K, \mathsf{bfvct}_{\mathsf{sk}})$ as in Section 4.1.
13:                                             ▷ $K$ generated with $\mathsf{sk}_{\mathsf{BFV}}, \mathsf{sk}, Q', \chi_{\sigma'}$.
14:    return $(\mathsf{pp} = \mathsf{pp}_{\mathsf{lwe}} = (n, q, p = 3, \mathcal{D}_1, \chi_\sigma), \mathsf{pp}_{\mathsf{bfv}} = (N, Q, Q', t = q, \mathcal{D}_2, \chi_{\sigma'}), \mathsf{sk}, \mathsf{btk})$
15: procedure GateOps($\mathsf{ct}_1 = (\vec{a}_1, b_1), \mathsf{ct}_2 = (\vec{a}_2, b_2), g, q$)
16:    $\mathsf{ct} = (\vec{a}, b) \leftarrow (\vec{a}_1 + \vec{a}_2, b_1 + b_2)$        ▷ Operations in $\mathbb{Z}_q$.
17:    if $g$ is AND or NAND then
18:        $b \leftarrow b + \lfloor q/6 \rfloor$
19:    else if $g$ is OR or NOR then
20:        $b \leftarrow b - \lfloor q/6 \rfloor$
21:    else                                     ▷ $g$ is XOR or XNOR
22:        $b \leftarrow b - \lfloor q/3 \rfloor - \lfloor q/6 \rfloor$
23:    return ct
24: procedure Negation($\mathsf{ct} = (\vec{a}, b), g, q$)
25:    if $g$ is NOR or AND or XOR then
26:        $\vec{a} \leftarrow -\vec{a}, b \leftarrow \lfloor q/3 \rfloor - b$        ▷ Operations in $\mathbb{Z}_q$.
27:    return $\mathsf{ct} = (\vec{a}, b)$
28: procedure Boot($(\mathsf{ct}_{1,i})_{i\in[N]}, (\mathsf{ct}_{2,i})_{i\in[N]}, (g_i)_{i\in[N]}, \mathsf{btk} = (\mathsf{BFV.pk}, \mathsf{BFV.evk}, \mathsf{BFV.rtk}, K, \mathsf{bfvct}_{\mathsf{sk}})$)
29:    $\mathsf{ct}_i \leftarrow \mathsf{GateOps}(\mathsf{ct}_{1,i}, \mathsf{ct}_{2,i}, g_i, q), \forall i \in [N]$
30:    $A \leftarrow (\vec{a}_1^\intercal, \ldots, \vec{a}_N^\intercal)$
31:    $\mathsf{bfvct}_1 \leftarrow \mathsf{LT}(\mathsf{BFV.rtk}, A, \mathsf{bfvct}_{\mathsf{sk}})$
32:    $\mathsf{bfvct}_2 \leftarrow (b_1, \ldots, b_N) - \mathsf{bfvct}_1$        ▷ Homomorphically computes $\vec{b} - \mathsf{sk}A$
33:    $\mathsf{bfvct}_3 \leftarrow \mathsf{BFV.Eval}(\mathsf{BFV.evk}, \mathsf{bfvct}_2, \mathsf{DRaMpoly})$
34:    $\mathsf{bfvct}_4 \leftarrow \mathsf{LT}(\mathsf{BFV.rtk}, U^\intercal, \mathsf{bfvct}_3)$        ▷ Recall that $U$ is the matrix defined for packing
35:    $\mathsf{bfvct}_5 \leftarrow \mathsf{BFV.ModSwitch}(\mathsf{bfvct}_4, Q')$        ▷ $\mathsf{BFV.ModSwitch}$ is described in Section 3.3
36:    $(\mathsf{ct}'_1, \ldots, \mathsf{ct}'_N) \leftarrow \mathsf{Extract}(\mathsf{bfvct}_5)$        ▷ $\mathsf{Extract}$ same as in Algorithm 2.
37:    $\mathsf{ct}''_i \leftarrow \mathsf{KeySwitch}(K, \mathsf{ct}'_i), \forall i \in [N]$
38:                        ▷ $\mathsf{KeySwitch}$ is defined in Section 3.2, and $K$ is the key switching key
39:    $\mathsf{ct}_{\mathsf{out}_i} \leftarrow \mathsf{ModSwitch}(\mathsf{ct}''_i, q), \forall i \in [N]$        ▷ $\mathsf{ModSwitch}$ is defined in Section 3.2
40:    $\mathsf{ct}_{\mathsf{out}_i} \leftarrow \mathsf{Negation}(\mathsf{ct}_{\mathsf{out}_i}, g_i, q), \forall i \in [N]$
41:    return $(\mathsf{ct}_{\mathsf{out}_1}, \ldots, \mathsf{ct}_{\mathsf{out}_N})$
```

In regular FHEW/TFHE bootstrapping, this function f is required to be negacyclic [‡‡]. There have been several recent works trying to allow arbitrary functions [16, 33, 40]. However, all of them require at least two bootstrapping operations (or equivalent overhead), which is not efficient (not to mention the increasing parameters due to algorithm change, inducing an even larger cost). Moreover, the two works [33, 40] with implementation only tolerate small precision ([40] benchmarks for 3 bits, and [33] takes tens seconds for 7 bis of precision, see Section 7 for more details).

In this section, we show how to improve our batched multi-binary-gate bootstrapping construction to a bootstrapping construction to evaluate an *arbitrary function*.

6.1 Construction

Recall that for LWE encryption, a message $m \in \mathbb{Z}_p$ is encoded to a message $x \in \mathbb{Z}_q$ by computing $x \leftarrow m \cdot \alpha$, where $\alpha = \lfloor q/p \rfloor$, p is the plaintext modulus of the LWE ciphertext, and q is the ciphertext modulus of the LWE ciphertext. Also recall that we use $t = q$, where t is the plaintext modulus of the BFV scheme.

Our main observation is that given function $f : \mathbb{Z}_p \rightarrow \mathbb{Z}_p$, we can create a look-up table $\mathsf{LUT} : \mathbb{Z}_t \rightarrow \mathbb{Z}_t$ such that $\mathsf{LUT}(x) = (f(\lfloor x/\alpha \rceil)) \cdot \alpha$, where $\alpha = \lfloor q/p \rfloor$. Let BFV plaintext space $t = q$, we create the following polynomial function

$$\mathsf{fpoly}(x) = \mathsf{LUT}(0) - \sum_{i=1}^{t-1} x^i \sum_{a=0}^{t-1} \mathsf{LUT}(a) a^{t-1-i} \ . \tag{2}$$

Then, we replace $\mathsf{DRaMpoly}$ evaluated homomorphically using BFV on line 33 in Algorithm 3 with fpoly, and the other procedures remain exactly the same. This achieves our goal without any cost [§§], unlike prior works [16, 33, 40] (concretely are all at least 5x slower than the binary gate bootstrapping with their implementation, see Section 7 for details).

We formalize our constructions in Algorithm 4.

Theorem 6.1. For any $p = \mathsf{poly}(\lambda)$, let $(\mathsf{pp}_{\mathsf{lwe}} = (n, q, p, \mathcal{D}_1, \chi_\sigma), \mathsf{pp}_{\mathsf{bfv}} = (N, Q, Q', t = q, \mathcal{D}_2, \chi_{\sigma'}), \mathsf{sk}, \mathsf{btk}) \leftarrow \mathsf{Setup}(1^\lambda, p)$ in Algorithm 4, for any input LWE ciphertexts $\vec{\mathsf{ct}_1} = (\mathsf{ct}_{1,1}, \ldots, \mathsf{ct}_{1,N}), \vec{\mathsf{ct}_2} = (\mathsf{ct}_{2,1}, \ldots, \mathsf{ct}_{2,N})$ parameterized by $\mathsf{pp}_{\mathsf{lwe}}$, $\vec{\mathsf{ct}} = (\mathsf{ct}_1, \ldots, \mathsf{ct}_N)$ encrypting $\vec{m} = (m_1, \ldots, m_N)$, for $m_{i \in [N]} \in \mathbb{Z}_p$ under sk, and $\mathsf{err}(\mathsf{ct}_i) < \left\lfloor \frac{q}{2p} \right\rfloor$ for all $i \in [N]$ and a vector of gates $(g_1, \ldots, g_N) \in \{\text{OR, NOR, AND, NAND, XOR, XNOR}\}^N$; let $(\mathsf{ct}_{\mathsf{out}_1}, \ldots, \mathsf{ct}_{\mathsf{out}_N}) \leftarrow \mathsf{Boot}(f, \vec{\mathsf{ct}}, \mathsf{btk})$ in Algorithm 4, it holds that: $\mathsf{Dec}(\mathsf{ct}_{\mathsf{out}_i}) = f(\mathsf{Dec}(\mathsf{ct}_i)), \mathsf{err}(\mathsf{ct}_{\mathsf{out}_i}) < \left\lfloor \frac{q}{2p} \right\rfloor$, and $\mathsf{ct}_{\mathsf{out}i}$ is encrypted under secret key sk, for all $i \in [N]$.

Proof sketch. We prove from the basic correctness (i.e., evaluating f correctly over the encrypted messages) and noise analysis.

- (Correctness) Correctness (i.e., $\mathsf{Dec}(\mathsf{ct}_{\mathsf{out}_i}) = f(\mathsf{Dec}(\mathsf{ct}_i))))$ is intuitive. Most of the parts remain exactly the same as the proof for Theorem 4.3. The only thing we need to argue that $\mathsf{LUT} : \mathbb{Z}_t \rightarrow \mathbb{Z}_t$ defined as $\mathsf{LUT}(x) = (f(\lfloor x/\alpha \rceil)) \cdot \alpha$, where $\alpha = \lfloor q/p \rfloor$ and $t = q$, correctly represents an arbitrary function $f : \mathbb{Z}_p \rightarrow \mathbb{Z}_p$. This can be demonstrated as follow: given LWE parameters $(n, q, p, \mathcal{D}_1, \chi)$, for all $x \in \mathbb{Z}_q$, let $y \leftarrow \left\lceil \frac{x}{\lfloor q/p \rfloor} \right\rfloor \in \mathbb{Z}_p$, for any LWE secret key $\mathsf{sk} \leftarrow \mathcal{D}_1$, any $\vec{a} \leftarrow_\$ \mathbb{Z}_q^n$, let $\mathsf{ct} \leftarrow (\vec{a} \in \mathbb{Z}_q^n, \langle a, \mathsf{sk} \rangle + \mathsf{LUT}(x) + e)$ where $e \leftarrow \chi$, it holds that $\Pr[\mathsf{Dec}(\mathsf{ct}) = f(y)] \geq 1 - \mathsf{negl}(\lambda)$ (probability over the error sampling), by the correctness of the underlying LWE scheme.

[‡‡] More precisely, they require the function to be first transformed into a function $\mathbb{Z}_q \rightarrow \mathbb{Z}$ and this transformed function needs to be negacyclic. For details, see [43]. However, either way, this constraint is very strong and makes the functionality much more limited.

[§§] Note that concretely, the number of zero coefficients increases as discussed in Section 4.5, but this only incurs a small overhead. See Section 7 for more details.

Algorithm 4 Batched FHEW/TFHE bootstrapping for arbitrary function over $\mathbb{Z}_p$

```
1: procedure $\mathsf{Setup}(1^\lambda, p)$
2:     Select $(n, q, \mathcal{D}_1, \chi_\sigma, N, Q, \mathcal{D}_2, \chi_{\sigma'}, Q')$ minimizing the cost of the entire homomorphic circuit evaluation and also satisfying:
3:     (1) $\mathsf{LWE}_{n,q,\mathcal{D}_1,\chi_\sigma}$ holds, and $\Pr[e < \left\lfloor \frac{q}{2p} \right\rfloor] \geq 1 - \mathsf{negl}(\lambda)$ where $e \leftarrow \chi_\sigma$.
4:     (2) $\mathsf{RLWE}_{N,Q,\mathcal{D}_2,\chi_{\sigma'}}$ hold.
5:     (3) $Q' \leq Q, \mathsf{LWE}_{n,Q',\chi_{\sigma'}}$ holds.
6:     (4) BFV with parameters $N, Q, t = q, \mathcal{D}_2, \chi_{\sigma'}$ is enough to evaluate the circuit in procedure $\mathsf{Boot}$
7:     (5) $\mathsf{LWE}_{n,q,\mathcal{D}_1,\chi_{\sigma_{\mathsf{res}}}}$ holds, where $\sigma_{\mathsf{res}} = \sqrt{(\frac{t}{Q'}(\sigma_5 + \sigma_{\mathsf{ks}}))^2 + {\sigma_{\mathsf{ms}}}^2}$, where $\sigma_5$ is the noise distribution standard deviation of line 34
8:     (6) $\Pr[e' < \left\lfloor \frac{q}{2p} \right\rfloor] \geq 1 - \mathsf{negl}(\lambda)$ where $e' \leftarrow \chi_{\sigma_{\mathsf{res}}}$
9:                                         ▷ $\sigma_{\mathsf{ks}}, \sigma_{\mathsf{ms}}$ are as introduced in Section 3.2.
10:    $\mathsf{sk} \leftarrow_\$ \{-1, 0, 1\}^n$
11:    Generate BFV secret key $\mathsf{sk}_{\mathsf{BFV}}$.
12:    Generate bootstrapping keys $\mathsf{btk} = (\mathsf{BFV.pk}, \mathsf{BFV.evk}, \mathsf{BFV.rtk}, K, \mathsf{bfvct}_{\mathsf{sk}})$ as in Section 4.1.
13:                                         ▷ $K$ generated with $\mathsf{sk}_{\mathsf{BFV}}, \mathsf{sk}, Q', \chi'$.
14:    return $(\mathsf{pp} = \mathsf{pp}_{\mathsf{lwe}} = (n, q, p, \mathcal{D}_1, \chi_\sigma), \mathsf{pp}_{\mathsf{bfv}} = (N, Q, t = q, \mathcal{D}_2, \chi_{\sigma'}), \mathsf{sk}, \mathsf{btk})$
15: procedure $\mathsf{Extract}(\mathsf{bfvct} = (a_{\mathsf{bfv}}, b_{\mathsf{bfv}}))$    ▷ Extract $N$ LWE ciphertexts from one BFV ciphertext
16:    Initialize $\mathsf{ct}_i = (\vec{a}_i, b_i)$ for $i \in [N]$
17:    for $i \in [N]$ do
18:        for $j \in [N]$ do
19:            if $j \leq i$ then
20:                $\vec{a}_i[j] \leftarrow a_{\mathsf{bfv}}[i - j + 1]$
21:            else
22:                $\vec{a}_i[j] \leftarrow -a_{\mathsf{bfv}}[N - j + i + 1]$
23:                         ▷ Negative number is still in $\mathbb{Z}_q$ where $q$ is the $\mathsf{bfvct}$ ciphertext modulus
24:        $b_i = b_{\mathsf{bfv}_i}$
25:    return $(\mathsf{ct}_1, \ldots, \mathsf{ct}_N)$
26: procedure $\mathsf{Boot}(f : \mathbb{Z}_p \to \mathbb{Z}_p, (\mathsf{ct}_1, \ldots, \mathsf{ct}_N), \mathsf{btk} = (\mathsf{BFV.pk}, \mathsf{BFV.evk}, \mathsf{BFV.rtk}, K, \mathsf{bfvct}_{\mathsf{sk}}))$
27:    Compute $\mathsf{LUT}(x) = (f(\lfloor x/\alpha \rceil)) \cdot \alpha$, where $\mathsf{LUT} : \mathbb{Z}_t \to \mathbb{Z}_t$
28:    Compute $\mathsf{fpoly}(x) = \mathsf{LUT}(0) - \sum_{i=1}^{t-1} x^i \sum_{a=0}^{t-1} \mathsf{LUT}(a) a^{t-1-i}$
29:    $A \leftarrow (\vec{a}_1^{\mathsf{T}}, \ldots, \vec{a}_N^{\mathsf{T}})$
30:    $\mathsf{bfvct}_1 \leftarrow \mathsf{LT}(\mathsf{BFV.rtk}, A, \mathsf{bfvct}_{\mathsf{sk}})$
31:    $\mathsf{bfvct}_2 \leftarrow (b_1, \ldots, b_N) - \mathsf{bfvct}_1$        ▷ Homomorphically computes $\vec{b} - \mathsf{sk}A$
32:    $\mathsf{bfvct}_3 \leftarrow \mathsf{BFV.Eval}(\mathsf{BFV.evk}, \mathsf{bfvct}_2, \mathsf{fpoly})$
33:    $\mathsf{bfvct}_4 \leftarrow \mathsf{LT}(\mathsf{BFV.rtk}, U^{\mathsf{T}}, \mathsf{bfvct}_3)$        ▷ Recall that $U$ is the matrix defined for packing
34:    $\mathsf{bfvct}_5 \leftarrow \mathsf{ModSwitch}(\mathsf{bfvct}_4, Q')$        ▷ $\mathsf{ModSwitch}$ is described in Section 3.3
35:    $(\mathsf{ct}'_1, \ldots, \mathsf{ct}'_N) \leftarrow \mathsf{Extract}(\mathsf{bfvct}_5)$
36:    $\mathsf{ct}''_i \leftarrow \mathsf{KeySwitch}(K, \mathsf{ct}'_i), \forall i \in [N]$
37:                         ▷ $\mathsf{KeySwitch}$ is defined in Section 3.2, and $K$ is the key switching key
38:    $\mathsf{ct}_{\mathsf{out}_i} \leftarrow \mathsf{ModSwitch}(\mathsf{ct}''_i, q), \forall i \in [N]$        ▷ $\mathsf{ModSwitch}$ is defined in Section 3.2
39:    return $(\mathsf{ct}_{\mathsf{out}_1}, \ldots, \mathsf{ct}_{\mathsf{out}_N})$
```

- (Noise Analysis) The noise analysis remains the same as in Theorem 4.3. Each resulting ciphertext has an error with $\sigma = \sqrt{(\frac{t}{Q'}(\sigma_5 + \sigma_{\mathsf{ks}}))^2 + {\sigma_{\mathsf{ms}}}^2}$ as the standard deviation. Thus, by condition (6) in Setup, $\mathsf{err}(\mathsf{ct}_{\mathsf{out}_i}) < \left\lfloor \frac{q}{2p} \right\rfloor, \forall i \in [N]$.

□

Efficiency and security analysis. The efficiency and security analysis remain exactly the same as the NAND gate bootstrapping.

6.2 Size of p

Practically, we can achieve p of 9 bits given the parameters we choose (see Section 7), without introducing much overhead compared to the binary gate evaluation. This is already an improvement compared to prior state-of-the-art works.

This precision can be useful in many applications like machine learning using FHE [30, 33]. However, some other applications might need to further enlarge p. To accommodate a larger p, we need to increase the degree of the underlying polynomial. Note that the number of plaintext-by-ciphertext multiplications grows linearly with p,[¶¶] and the number of ciphertext-by-ciphertext multiplications grows with $\sqrt{p}$. Hence, our scheme still remains practical for some applications for p being 10-15 bits [***].

To more efficiently achieve even larger precision evaluation, one can do a digit decomposition and evaluate the function using additional techniques (e.g., the chain-based or tree-based methods) as explained in more detail in [40, Section 5]. The chain-based and tree-based methods are first introduced in [23] and later optimized by [39]. These techniques can potentially be integrated with our scheme, but we leave this for future works to further explore.

7 Evaluation

We implemented our algorithms proposed in Sections 4 to 6 in a C++ library. We use Palisade [46] for our LWE ciphertexts implementation, and SEAL [48] for BFV scheme. We benchmark these schemes on several parameter settings on a Google Compute Cloud `e2-standard-4` with 16GB RAM.

Binary gate bootstrapping parameter selection. We choose LWE parameters as follows: $n = 1024, q = 65537, p = 3, \sigma = 3.2$, and BFV parameters as follows: $N = 32768, \log Q \approx 673, t = 65537, \sigma' = 3.2$. We use Gaussian secret keys with a standard deviation of 3.2 for our LWE ciphertexts. These parameters guarantee $>$ 128-bit security by LWE estimator [2].

Arbitrary function evaluation parameter selection. n, σ, N, σ' remain unchanged. Set $(t = q = 256^2 + 1, \log Q \approx 673)$ for $p = 2^9$, and $(t = q = 768 \cdot 1024 + 1, \log Q \approx 900)$ for $p = 2^{12}$. We use ternary secret keys for arbitrary function evaluation to reduce the error of modulus switching. These parameters guarantee $>$ 125-bit security by LWE estimator [2].

We choose q, p such that the error of the output ciphertext is larger than $\left\lfloor \frac{q}{2p} \right\rfloor$ with probability $< 2^{-30}$. By performing 1000 trials of bootstrapping, which is of total 32,768,000 LWE ciphertexts, we calculate the standard deviation of the output error to be ∼10. Thus, we choose q, p accordingly such that $\left\lfloor \frac{q}{2p} \right\rfloor \geq 64$.

Binary gate bootstrapping. From Table 2, we can see that our performance is more than 30x faster than the state-of-the-art C++ implementation of the non-batched construction. We are also more than 3x faster compared to the rust implementation by TFHE-rs [50]. Moreover, our construction supports Gaussian secret keys without any loss in performance, while the TFHE construction does not. With a secret key generated

[¶¶] Each plaintext-by-ciphertext multiplication only requires N $\mathbb{Z}_Q$ multiplications as the "plaintext" is a scalar.

[***] Asymptotically, the cost of our scheme is dominated by the number of scalar-by-ciphertext multiplications, which grows linear in p. Thus, our scheme becomes impractical when p is too large (e.g., 20 bits or more).

	TFHE (OpenFHE)	TFHE (TFHE-rs)	**Ours (Single Gate)**	**Ours (Mixed Gates)**
Amortized time per LWE ciphertext (ms)	190	17.5	**4.7**	**4.7**
Total time (sec)	0.19	0.0175	**155**	**155**
# of ciphertexts per bootstrapping	1	1	**32768**	**32768**
LWE ciphertexts secret key type	Ternary	Binary	**Gaussian**	**Gaussian**

Table 2: Runtime comparison between our construction and state-of-the-art implementation of non-batched FHEW/TFHE-like cryptosystems. Mixed gates means the 32768 input gates contain a mix of AND/NAND/OR/NOR/XOR/XNOR gates (∼32768/6 for each type). For OpenFHE, we benchmark it with Intel HEXL optimizaiton and their own NATIVEOPT. For Concrete, we benchmarked it with AVX512 optimization on.

Plaintext space	3-bit	7-bit		8-bit			9-bit	
Scheme(s)	LMP22 [40]	GPL23 [24]	FDFB[33]	GBA21[23]	LXDX23 [39]	LMP22 [40]	PEGASUS[30]	**Ours**
Amortized time (ms) per LWE ciphertext	1192	1205	35169	2203	409	1793	3476	**6.7**
Total time (sec)	1.192	1234	35.169	2.203	0.409	1.793	3.476	**220**
# of ciphertexts per bootstrapping	1	1024	1	1	1	1	1	**32768**
Input assumption	No	No	No	A vector of LWE ct's with small precision	A vector of LWE ct's with small precision	Only works for sign function and CT decomposition	MSB of the input ciphertext being 0	**No**

Table 3: Runtime comparison between our work and prior or concurrent works to evaluate an arbitrary function with ≤ 9 bits of precision (i.e., plaintext space ≤ 9 bits). Note that GPL23 [24] and LXDX23 [39] do not have an implementation publicly available so we directly reuse the number from their papers. GBA21 [23] only has the number for 6-bit, so we directly take the numbers reported in [39] for [23]. [24] has environment: Intel Xeon Platinum 8252C CPU at 4.5 GHz with 192 GB of RAM at 2933 MHz, and [39] has environment: i7-10700F at 2.90GHZ. Both environments are better than our running environment and thus not under-estimating their performance. All other numbers are based on the experiment with the same environment as ours.

under a Gaussian distribution (i.e., each secret key element is sampled from a Gaussian distribution $(0, \sigma)$), the scheme is more secure than the one with a ternary or binary secret key.

We only compare with the TFHE construction, not FHEW or Lee et al. [36], as to our knowledge, if simply considering the bootstrapping runtime, TFHE is the most efficient one.

Arbitrary function evaluation. Through our experiment, we losslessly support 9 bits of precision with $t = 65537$ for arbitrary function evaluation with bootstrapping. In terms of amortized cost, our construction is about two to three orders of magnitude faster than all the other schemes providing 9-bits or less as demonstrated in Table 3. For even larger precision, 12-bits, as shown in Table 4, our construction achieves even better results: all more than two orders of magnitude faster than any other existing schemes. All the existing schemes benchmark at most ≤ 12 bits of precision, for an arbitrary function evaluation, so we follow this convention.

Furthermore, we make no assumption on the inputs. On the other hand, PEGASUS [30] (implementing TFHE functional bootstrapping) requires the input ciphertexts MSB to be 0; LMP22 [40] can only perform sign function and ciphertext decomposition (i.e., decomposing a large precision ciphertext into a vector of small precision ciphertexts) for larger precision (> 3-bits); LXDX23 [39] follows the route of [23], and thus both [39] and [23] require the input to be a vector of LWE ciphertexts with small precision, instead of a single LWE ciphertext with large precision [†††]. Hence, our scheme is not only much faster, but also much

[†††] To decompose a large precision ciphertext into a vector of small precision ciphertexts, one may use LMP22[40], which

Plaintext space	10-bit		12-bit			
Scheme(s)	GBA21[23]	LXDX23 [39]	LMP22 [40]	GBA21[23]	LXDX23[39]	**Ours**
Amortized time (ms) per LWE ciphertext	23667	1779	3998	51085	8092	**39.1**
Total time (sec)	23.667	1.779	3.998	51.085	8.092	**1280**
# of ciphertexts per bootstrapping	1	1	1	1	1	**32768**
Input assumption	A vector of LWE ct's with small precision	A vector of LWE ct's with small precision	Only works for sign function and CT decomposition	A vector of LWE ct's with small precision	A vector of LWE ct's with small precision	**No**

Table 4: Runtime comparison between our work and prior or concurrent works to evaluate an arbitrary function with larger bits of precision (i.e., plaintext space > 9 bits). For experiment environment, same as Table 3, we run the experiments using the code for LMP22 [40] and take the numbers from [39] for [23, 39].

stronger and more flexible in terms of functionality [‡‡‡].

Bootstrapping key size. Our btk size is as follows. The numbers are for $t = 256^2 + 1$ and $t = 768 \cdot 1024 + 1$ for $p = 2^9$ and $p = 2^{12}$ respectively. Our construction requires 32 rotation keys with full-level, ∼65MB and ∼107MB each; 128 rotation keys with 2 levels, both ∼1MB each; 32 BFV ciphertexts with full-level, ∼5.2MB and ∼7MB each; 1 BFV public key, ∼5.2MB and ∼7MB; and 1 BFV relinerization key, ∼65MB and ∼107MB. In total, the btk size is ∼2.38GB for binary gates and 9-bit LUT. For 12-bit LUT, it is ∼3.80GB.

8 Extension

8.1 Scheme Switching

Another important application of our scheme is FHEW/TFHE and BFV/BGV scheme switching. Scheme switching was first introduced by Chimera [6] and later improved by PEGASUS [30].

The main goal of scheme switching is to change the ciphertexts of one FHE scheme to the other and switch back. During the encryption, the same plaintext is encrypted under different schemes. The motivation is that BFV/BGV/CKKS only support polynomial evaluation but when switching to FHWE/TFHE, we can evaluate an arbitrary LUT, such as comparisons.

Both of the prior works mainly focus on FHEW/TFHE-CKKS scheme switching, while FHEW/TFHE-BFV/BGV scheme switching is only discussed, without a thorough study or implementation. Another work [11] converts LWE ciphertexts to RLWE ciphertexts and back. However, in their conversion, the output ciphertexts depend on the error of the input ciphertexts, which makes the conversion not quite compatible with the motivation of scheme switching. Ideally, when transforming LWE to RLWE ciphertexts, the noise of the output should be independent of the input noise, so that the application can take full advantage of the leveled HE property without enlarging the LWE ciphertext's modulus or dimension. We refer the readers to [11, 6, 30] for more details.

In contrast, our work achieves the conversion between FHEW/TFHE ciphertexts and BFV/BGV ciphertexts. The same process as in Section 4.3 can be adapted for FHEW/TFHE-BFV scheme switching. The only change is to replace DRaMpoly with a function according to the underlying plaintext modulus p (instead of 3) of the input LWE ciphertexts. Section 4.4 can be directly applied for BFV-FHEW/TFHE scheme switching [§§§].

introduces another 5-6 seconds of overhead for a 12-bit precision LWE ciphertext.

[‡‡‡] Note that we do not compare with a concurrent and independent work [41], as it focuses on optimizing the schemes in [40] and achieving a 2-3x runtime improvement. We believe that this improvement does not affect our overall comparison, and to our knowledge, there is no open-sourced code available. However, note that this work shows a great improvement over [40] and may be preferred over our result when the number of bootstrapping needed is small.

[§§§] Note that here the output LWE ciphertexts have error dependent on the input BFV ciphertexts. This is fine in many applications, as FHEW/TFHE evaluation itself involves bootstrapping.

Our scheme switching is similar to the scheme switching between FHEW/TFHE and BFV/BGV described in [6, Sec 3.1]. However, directly applying the schemes they introduce is relatively impractical. Our scheme, on the other hand, practically realizes this functionality by carefully matching the parameters and introducing optimizations to fully use the power of BFV. Moreover, our scheme can evaluate an arbitrary function during the conversion, as introduced in Section 6.

Note that the LWE plaintext modulus p and BFV plaintext modulus t are different (as $t = q > p$). One may encrypt a message $m \in \mathbb{Z}_p$ using the LWE ciphertext, and then to switch to BFV, the output BFV ciphertext encrypts $m' \in \mathbb{Z}_t$ (i.e., the encrypted message is lifted to $\mathbb{Z}_t$). Thus, the subsequent circuit evaluation must be done via $\mathbb{Z}_t$.

Benchmark. We benchmark this extension with the following parameters: given $N = 32768$ LWE ciphertexts with $n = 1024, q = 65537, p = 2^9$, switching them into one BFV ciphertext with $N = 32768, Q \approx 2^{793}, t = 65537$ and with ~5 levels left (roughly 150 bits of noise budget left). Then, given a BFV ciphertext with one level left, for $N = 32768, t = 65537$, convert it two 32768 LWE ciphertexts.

- (FHEW/TFHE to BFV) ~296 seconds
- (BFV to FHEW/TFHE) ~17 seconds

8.2 Batched LWE Ciphertext Bootstrapping Based on CKKS

An interesting question is can we use CKKS to do batched LWE ciphertext bootstrapping? To our knowledge, the answer is both yes and no.

Evaluating NAND gates using CKKS is easy. Similarly, we first compute $b - \langle \vec{a}, s \rangle$. Then, since CKKS performs an approximate real number evaluation, we get a result of $kq + m + e$ for some integer k. To recover m, we need to remove kq and e at the same time and map m to 0 or 1. This can be done by using a sine function to approximate the process. This sine-based approximation for modular computation is first proposed in [13] and later greatly improved in a long line of work [10, 27, 35, 28, 5, 34, 32]. Further, since sine is not a polynomial function, one needs to use a polynomial function to approximate sine, which has also been extensively studied in this line of work.

In a nutshell, if $\|\mathsf{sk}\|_1$ (LWE ciphertext secret key) is small, k is small and thus this process is actually very efficient. Therefore, using CKKS to perform batched NAND gate bootstrapping can potentially be more efficient than what we introduce in Section 4.

Unfortunately, this process is hard to be extended to support arbitrary function evaluation. If we want to evaluate an arbitrary function f together with bootstrapping, we need to compose f with the sine functions in an efficient way. However, since f is based on $\mathbb{Z}_p$ where p can be relatively large (e.g., 2^9), how to efficiently use polynomial functions to approximate a composition between the sine function and the arbitrary function f remains to be an open problem.

Therefore, for simple f (i.e., either p is small or f has some specific structure like the NAND gate), using CKKS to do batched bootstrapping can potentially be a great alternative. Nevertheless, since evaluating arbitrary functions is an essential feature of FHEW/TFHE cryptosystems [30, 40] and is widely applied in many applications, we choose to use BFV. Exploring this direction in more detail is left for future work.

9 Concluding Remarks

In this work, we show a novel way to do batched bootstrapping for LWE ciphertexts. Based on the benchmark of the implementation, our method is orders of magnitude faster than the prior works when considering the amortized cost, thus adding a strong tool to the FHEW/TFHE line of work. It also opens other new and interesting directions; for example, how to efficiently extend our algorithm to support even larger plaintext space, or how to use CKKS as an alternative for batched functional bootstrapping. These are left for future works to explore.

Acknowledgements

We are grateful to Yuriy Polyakov for his insightful discussions and feedback, and to Wen-jie Lu for answering key-switching implementation questions regarding the SEAL library. We thank the Secure Decentralized Systems Lab at Yale University. We also thank all the reviewers for their insightful comments.

References

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

[2] 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.

[3] J. Alperin-Sheriff and C. Peikert. Practical bootstrapping in quasilinear time. pages 1–20, 2013.

[4] A. A. Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, I. Quah, Y. Polyakov, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca. Openfhe: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915, 2022. `https://eprint.iacr.org/2022/915`, commit: 122f470e0dbf94688051ab852131ccc5d26be934.

[5] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In A. Canteaut and F.-X. Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021*, pages 587–617, Cham, 2021. Springer International Publishing.

[6] 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.

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

[8] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Proceedings of the 32nd Annual Cryptology Conference on Advances in Cryptology — CRYPTO 2012 - Volume 7417*, page 868–886, Berlin, Heidelberg, 2012. Springer-Verlag.

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

[10] H. Chen, I. Chillotti, and Y. Song. Improved bootstrapping for approximate homomorphic encryption. In Y. Ishai and V. Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 34–54, Cham, 2019. Springer International Publishing.

[11] H. Chen, W. Dai, M. Kim, and Y. Song. Efficient homomorphic conversion between (ring) lwe ciphertexts. Cryptology ePrint Archive, Report 2020/015, 2020. `https://eprint.iacr.org/2020/015`.

[12] H. Chen and K. Han. Homomorphic lower digits removal and improved FHE bootstrapping, 2018.

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

[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] 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.

[16] I. Chillotti, D. Ligier, J.-B. Orfila, and S. Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for tfhe. In M. Tibouchi and H. Wang, editors, *Advances in Cryptology – ASIACRYPT 2021*, pages 670–699, Cham, 2021. Springer International Publishing.

[17] K. Cong, R. C. Moreno, M. B. da Gama, W. Dai, I. Iliashenko, K. Laine, and M. Rosenberg. Labeled psi from homomorphic encryption with reduced computation and communication. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21. Association for Computing Machinery, 2021.

[18] 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.

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

[20] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. `https://ia.cr/2012/144`.

[21] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[22] C. Gentry, S. Halevi, C. Peikert, and N. P. Smart. Ring switching in bgv-style homomorphic encryption. In I. Visconti and R. De Prisco, editors, *Security and Cryptography for Networks*, pages 19–37, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[23] 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.

[24] A. Guimarães, H. V. L. Pereira, and B. van Leeuwen. Amortized bootstrapping revisited: Simpler, asymptotically-faster, implemented. Cryptology ePrint Archive, Paper 2023/014, 2023. `https://eprint.iacr.org/2023/014`.

[25] S. Halevi and V. Shoup. Bootstrapping for HElib. Cryptology ePrint Archive, Report 2014/873, 2014. `https://eprint.iacr.org/2014/873`.

[26] S. Halevi and V. Shoup. Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive, Report 2020/1481, 2020. `https://eprint.iacr.org/2020/1481`.

[27] K. Han, M. Hhan, and J. H. Cheon. Improved homomorphic discrete fourier transforms and fhe bootstrapping. *IEEE Access*, 7:57361–57370, 2019.

[28] K. Han and D. Ki. Better bootstrapping for approximate homomorphic encryption. In *Cryptographers' Track at the RSA Conference*, pages 364–390. Springer, 2020.

[29] I. Iliashenko, C. Nègre, and V. Zucca. Integer functions suitable for homomorphic encryption over finite fields. Cryptology ePrint Archive, Report 2021/1335, 2021. WAHC 2021.

[30] 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`.

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

[32] S. Kim, M. Park, J. Kim, T. Kim, and C. Min. Evalround algorithm in ckks bootstrapping. Asiacrypt 2022, 2022. `https://eprint.iacr.org/2022/1256`.

[33] K. Kluczniak and L. Schild. Fdfb: Full domain functional bootstrapping towards practical fully homomorphic encryption. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(1):501–537, Nov. 2022.

[34] J.-W. Lee, E. Lee, Y. Lee, Y.-S. Kim, and J.-S. No. *High-Precision Bootstrapping of RNS-CKKS Homomorphic Encryption Using Optimal Minimax Polynomial Approximation and Inverse Sine Function*, pages 618–647. Springer International Publishing, 06 2021.

[35] Y. Lee, J.-W. Lee, Y.-S. Kim, and J.-S. No. Near-optimal polynomial for modulus reduction using l2-norm for approximate homomorphic encryption. *IEEE Access*, 8:144321–144330, 2020.

[36] Y. Lee, D. Micciancio, A. Kim, R. Choi, M. Deryabin, J. Eom, and D. Yoo. Efficient fhew bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption. In C. Hazay and M. Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 227–256, Cham, 2023. Springer Nature Switzerland.

[37] F.-H. Liu and H. Wang. Batch bootstrapping i: A new framework for simd bootstrapping in polynomial modulus. In C. Hazay and M. Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 321–352, Cham, 2023. Springer Nature Switzerland.

[38] F.-H. Liu and H. Wang. Batch bootstrapping i: Bootstrapping in polynomial modulus only requires o (1) fhe multiplications in amortization. In C. Hazay and M. Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 321–352, Cham, 2023. Springer Nature Switzerland.

[39] K. Liu, C. Xu, B. Dou, and L. Xu. Optimization of functional bootstrap with large lut and packing key switching. Cryptology ePrint Archive, Paper 2023/631, 2023. `https://eprint.iacr.org/2023/631`.

[40] Z. Liu, D. Micciancio, and Y. Polyakov. Large-precision homomorphic sign evaluation using fhew/tfhe bootstrapping. In *Advances in Cryptology – ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5–9, 2022, Proceedings, Part II*, page 130–160, Berlin, Heidelberg, 2023. Springer-Verlag.

[41] S. Ma, T. Huang, A. Wang, and X. Wang. Fast and accurate: Efficient full-domain functional bootstrap and digit decomposition for homomorphic computation. Cryptology ePrint Archive, Paper 2023/645, 2023. `https://eprint.iacr.org/2023/645`.

[42] S. J. Menon and D. J. Wu. Spiral: Fast, high-rate single-server pir via fhe composition. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 930–947, 2022.

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

[44] D. Miccianco and J. Sorrell. Ring Packing and Amortized FHEW Bootstrapping. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

[45] G. D. Micheli, D. Kim, D. Micciancio, and A. Suhl. Faster amortized fhew bootstrapping using ring automorphisms. Cryptology ePrint Archive, Paper 2023/112, 2023. `https://eprint.iacr.org/2023/112`.

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

[47] M. S. Paterson and L. J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM Journal on Computing*, 2(1):60–66, 1973.

[48] Microsoft SEAL, 2020. `https://github.com/Microsoft/SEAL`.

[49] N. Smart and F. Vercauteren. Fully homomorphic SIMD operations. Designs, Codes and Cryptography, 2011. `https://eprint.iacr.org/2011/133`.

[50] Zama-ai, tfhe-rs, 2023. https://github.com/zama-ai/tfhe-rs, commit: 509bf3e2846bc98dd42d0e8eeb7f27852e5b632a.

[51] 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.