New Techniques for Fast and Shallow FHE Bootstrapping and Beyond

Aayush Jain[1], Huijia Lin[2], Zeyu Liu[3], Sagnik Saha[1]

[1] Carnegie Mellon University
[2] University of Washington
[3] Yale University

Abstract

The main barrier to practical fully homomorphic encryption remains the latency and cost of bootstrapping, the ciphertext refresh step that enables unbounded computation. We design new methods that reduce both the latency and the circuit depth of bootstrapping in the FHEW/TFHE framework, which represents the state-of-the-art for lightweight bootstrapping and for computing deep and unstructured Boolean functions over encrypted data.

Our first contribution leverages LWE with a sparse small-norm secret, an assumption known to be equivalent to standard LWE and already widely used in FHE constructions. For an LWE secret of dimension n and Hamming weight h, we obtain bootstrapping procedures whose arithmetic complexity decreases from $\tilde{O}(n^2)$ to $\tilde{O}(nh)$ $\mathbb{Z}_q$ multiplications while preserving the same asymptotic number of additions. Concretely, this yields a 4.5–7.5× practical speedup for gate and functional bootstrapping over the state-of-the-art OpenFHE implementation.

Our second contribution introduces a new RLWE variant with structured secrets, called *Binary NTT RLWE*, and uses it to significantly reduce the circuit depth of FHEW/TFHE bootstrapping via a new relinearization-free BV multiplication technique. In concrete parameter settings, this reduces the number of sequential NTT/INTT layers required for bootstrapping to just 3, compared to more than 500 in standard FHEW/TFHE, while keeping the overall number of unit 32 or 64-bit word operations comparable to standard FHEW/TFHE bootstrapping. This substantial depth reduction suggests the potential for significantly lower bootstrapping latency on parallel, high-throughput architectures such as GPUs.

Finally, we analyze the security of the new RLWE assumption underlying our depth reduction, including worst-case-to-average-case and search-to-decision reductions, as well as evaluations against concrete attacks.

1 Introduction

Fully homomorphic encryption (FHE) enables arbitrary computation on encrypted data [95] and is a central goal of modern cryptography, with wide-ranging applications [63, 54, 64, 56, 32, 5, 3, 84, 40, 104, 52, 73, 78, 79, 66, 77, 71, 41, 25, 24, 30, 13, 101, 72]. In a breakthrough work, Gentry [43] constructed the first FHE scheme, followed by several new designs and substantial efficiency improvements, e.g., [19, 36, 44]. A key technique introduced by Gentry and underlying all FHE constructions is *bootstrapping*, whereby a ciphertext is homomorphically decrypted to produce a refreshed ciphertext encrypting the same message. This procedure enables the homomorphic evaluation of computations of unbounded depth and thus realizes the full vision of FHE. However, this crucial capability comes at a significant cost: bootstrapping remains

the primary performance bottleneck of practical FHE systems. For FHE to achieve widespread adoption, both the latency and throughput of bootstrapping must be substantially improved.

A large body of work has focused on improving bootstrapping efficiency, e.g., [34, 29, 69, 28, 49, 42, 89, 27, 62, 87, 105, 4, 61]. One line of research, initiated by BGV [18] and followed by BFV [16, 36], and CKKS [28, 22, 51, 14, 50, 65, 68, 58, 27], improves *amortized bootstrapping efficiency* via batching. While batching can dramatically reduce the amortized cost per ciphertext, it does not reduce latency. When only one or a small number of plaintexts need to be processed, the cost of handling an entire batch is still incurred, both for bootstrapping and for regular homomorphic operations.

A different approach, developed in the FHEW/TFHE line of work [34, 29], focuses on improving the efficiency of *single-ciphertext bootstrapping*. This approach provides low latency, flexibility for bootstrapping ciphertexts encrypting a single message, and the ability to exploit parallel architectures such as GPUs for high throughput by distributing independent gate bootstrapping operations across processors. Moreover, bootstrapping in this setting can be *functional*, meaning that an arbitrary function can be evaluated as part of the refresh procedure. Significant progress has been made in optimizing the FHEW/TFHE approach [87] (see Section 1.1 for further discussion).

Despite this encouraging progress, the efficiency of single-gate bootstrapping still leaves considerable room for improvement. In general, bootstrapping requires a large number of homomorphic additions and multiplications to refresh a ciphertext by homomorphically decrypting it using an encrypted secret key. In FHEW/TFHE, an LWE ciphertext is homomorphically decrypted using a secret encrypted under a different RLWE-based scheme. The dominant cost arises from a sequence of n homomorphic multiplications between a basic RLWE ciphertext and a ring variant of GSW ciphertext [44] (which we refer to as RLWE and RGSW ciphertexts, respectively), where n is the dimension of the LWE secret. Homomorphic decryption also involves $O(n)$ additions and multiplications by constant monomials, which can be implemented much more efficiently. As a result, this *RGSW-by-RLWE multiplication chain* dominates both the running time and the circuit depth of bootstrapping, and therefore directly limits both the latency and the throughput of FHEW/TFHE bootstrapping.

The central question we address in this work is the following:

Can the arithmetic cost and depth of FHEW/TFHE gate and functional bootstrapping be reduced?

Our Results in a Nutshell. In this work, we present new techniques for reducing both the *arithmetic cost* and the *circuit depth* of FHEW/TFHE gate bootstrapping by exploiting structured secrets underlying LWE and RLWE.

At the LWE layer, leveraging sparse secrets, which are already widely used in the FHE literature, we reduce the number of required RGSW-by-RLWE multiplications from n, the dimension of a dense LWE secret, to $h \ll n$, the Hamming weight of a sparse LWE secret. This yields substantial reductions in arithmetic cost and concrete speedups of $4.5\times$ –$7.5\times$ in bootstrapping time.

At the RLWE layer, under a new binary-NTT RLWE hardness assumption with a specialized secret distribution, we introduce a relinearization-free variant of BV-like multiplications. This enables TFHE bootstrapping to be evaluated at dramatically smaller circuit depth, requiring only 3 layers of NTT/INTT operations (compared to over 500 in the original FHEW/TFHE), while maintaining arithmetic costs comparable to FHEW/TFHE when combined with the LWE-layer technique. The depth reduction relies on this new RLWE-type assumption, whose security and supporting cryptanalysis are discussed in Section 6.

Next, we examine these two improvements in more detail.

Reducing arithmetic cost via sparse LWE secrets. The FHEW/TFHE decryption circuit consists primarily of $O(n)$ RGSW-by-RLWE multiplications, $O(n)$ monomial-by-RLWE multiplications, and $O(n)$ additions. The cost is dominated by the RGSW-by-RLWE multiplications.

We show that using a sparse LWE secret of dimension $n' = O(n)$ and sparsity (Hamming weight) h reduces the number of RGSW-by-RLWE multiplications to $O(h)$. This comes at the cost of introducing $O(n)$ monomial-by-RGSW multiplications and RGSW additions. This trade-off is highly favorable because these latter operations can be implemented very efficiently via coefficient rotations and point-wise additions.

To quantify the improvement, we consider instantiations over NTT-friendly RLWE rings, where the basic unit operations are the NTT, inverse NTT (INTT), point-wise multiplication, and point-wise addition. Among these, NTT/INTT operations dominate both runtime and circuit depth, and they are required only for RGSW-by-RLWE multiplications. Consequently, the arithmetic cost of bootstrapping decreases from $O(n)$ NTT/INTT, $O(n)$ point-wise multiplications, $O(n)$ point-wise additions to $O(h)$ NTT/INTT, $O(h)$ point-wise multiplications, $O(n)$ point-wise additions.

Theorem 1 (Sparse-secret bootstrapping cost, informal)**.** *Assume circular security for LWE over $\mathbb{Z}_q$ with a sparse secret and RLWE over an NTT-friendly ring $\mathcal{R}_Q$, where the RLWE ring dimension is $q/2|N$ and both q and Q are polynomially large. Then the bootstrapping procedure in Algorithm 2 performs, per ciphertext bootstrapping,*

$$O(h) \text{ NTT/INTT}, \qquad O(h) \text{ point-wise multiplications}, \qquad O(n') \text{ point-wise additions},$$

where n' is the LWE dimension and h is the sparsity (Hamming weight) of the LWE secret, assuming a probabilistic batch code with $O(1)$ overhead. All NTT/INTT, point-wise multiplication, and point-wise addition operations are over $\mathcal{R}_Q$. (We omit lower-cost bit-level operations such as bit decomposition and rotations.)

From a security perspective, restricting to sparse secrets incurs no essential loss. Results on entropic LWE show that secrets with sufficiently high entropy retain hardness comparable to that of dense secrets [46, 45, 17]. Concrete parameter estimates further confirm that sparse secrets achieve the same level of concrete security as dense secrets when evaluated using the LWE estimator [1]. For example, at the 120-bit security level and modulus $p = 512$, dense secrets with dimension $n = 503$ are comparable to sparse secrets with dimension $n' = 1024$ and sparsity $h = 43$. For modulus $p = 2^{12}$, dense secrets with dimension $n = 1350$ are comparable to sparse secrets with $n' = 2048$ and $h = 70$. Such sparse secrets are therefore already widely used in both theoretical and practical lattice-based constructions (e.g., [27, 14, 49, 83, 81, 23, 92]).

Finally, we validate these improvements through implementation. Our optimized design achieves approximately a 4.5× speedup for standard gate bootstrapping and up to a 7.5× speedup for 3-bit functional bootstrapping compared to OpenFHE [8]. See Section 7 for full evaluation details.

Reducing circuit depth via NTT-efficient multiplication. We next address the circuit depth of FHEW/TFHE bootstrapping. At a high level, FHEW/TFHE evaluates homomorphic decryption through a sequence of RGSW-by-RLWE multiplications. This results in an NTT/INTT depth proportional to the LWE dimension n. Since each NTT/INTT has depth $\log(N)$ for ring dimension N, these transformations dominate the overall circuit depth and form a primary bottleneck for hardware acceleration (see Section 5.2).

A natural idea is to organize these multiplications in a tree-like fashion, which would reduce the depth to $O(\log n)$. However, RGSW-by-RLWE homomorphic multiplication is not directly compatible with such tree-style evaluation. Switching instead to alternative multiplication procedures (such as multiplying two RGSW ciphertexts or two BGV ciphertexts) can

reduce depth but significantly increases arithmetic cost, often resulting in unfavorable trade-offs in practice. When combined with our sparsity-based improvement from the previous section, a tree-style construction would reduce the NTT/INTT depth to $O(\log h)$ while requiring only $O(h)$ expensive multiplications.

To further reduce both depth and arithmetic cost, we investigate NTT-efficient homomorphic encryption schemes. We introduce a new hardness assumption, which we call the *binary-NTT RLWE assumption* (Section 3.2). Based on this assumption, we construct a *relinearization-free* BV-like ciphertext multiplication procedure. Crucially, eliminating relinearization allows ciphertext components to remain in the NTT representation throughout w successive multiplications, thereby avoiding repeated NTT/INTT transformations. This reduces the circuit depth of these multiplications from $\Omega(w \log N)$ to $O(w)$. Moreover, we make the procedure remain compatible with BGV-style noise reduction via modulus switching [18], which can be applied periodically (e.g., after every w multiplications) to control noise at the cost of additional NTT/INTT layers.

Applying tree-style multiplication with this relinearization-free procedure yields a bootstrapping construction with NTT/INTT depth

$$D = O(\log h / \log w),$$

and modulus size $n^{O(wD)}$, for a tunable parameter w that captures the trade-off between depth and arithmetic cost.

Concretely, combining this RLWE-layer optimization with the sparsity-based improvements described above reduces the NTT/INTT depth of TFHE bootstrapping to just 3 layers, compared to over 500 layers in the original TFHE construction. At the same time, the total number of $\mathbb{Z}_q$ multiplications is smaller than in TFHE, at the cost of operating over 64-bit machine words rather than the 32-bit words used in TFHE. These properties make the resulting construction particularly well suited to GPUs and other parallel architectures.

Theorem 2 (Low-depth bootstrapping, informal)**.** *Replace RLWE in Theorem 1 with the* binary-NTT RLWE *hardness assumption (Definition 7).*

Let $w \geq 2$ be an integer parameter. The bootstrapping procedure in Algorithm 3 that evaluates the homomorphic decryption computation using a w-ary multiplication tree (instantiated via the relinearization-free BV scheme) performs, per ciphertext bootstrapping,

$$O(h/w) \text{ NTT/INTT}, \qquad O(h) \text{ point-wise multiplications}, \qquad O(n') \text{ point-wise additions},$$

and achieves sequential NTT/INTT depth

$$D \;=\; O(\log h / \log w),$$

and the RLWE modulus $Q = n^{O(wD)}$, while the LWE modulus remains polynomial assuming a probabilistic batch code with $O(1)$ overhead.

We conduct initial cryptanalysis of Binary NTT RLWE, including worst-case-to-average-case search-to-decision reductions, as well as evaluations against standard lattice attacks and multivariate quadratic attacks. We emphasize, however, that this assumption warrants further cryptanalytic study.

We present the asymptotic complexity of both of our main results in terms of 32-bit and 64-bit word operations in Table 1, and report concrete performance in Section 7.

Finally, our relinearization-free multiplication technique yields relinearization-free variants of BGV and CKKS, which may be of independent interest. We discuss these extensions in more detail in Section 5.3.

Scheme	# of $\mathbb{Z}_Q$ Operations		Depth	Security Assumption (Circular)		Cost per
	Multiplications	Additions	NTT/INTT	LWE component	RLWE component	$\mathbb{Z}_Q$ Operation
FHEW/TFHE	$N \cdot n \cdot \log(N)$	$N \cdot n \cdot \log(N)$	n	Binary/Ternary	Ternary	$\frac{\log(n)}{32}$ of 32-bit
Main result 1 (Algorithm 2)	$N \cdot h \cdot \log(N)$	$N \cdot (n + h \cdot \log(N))$	h	Sparse Binary/Ternary	Ternary	$\frac{\log(n)}{32}$ of 32-bit
Main result 2 (Algorithm 3)	$N \cdot h \cdot \log(N)$	$N \cdot (n + h \cdot \log(N))$	$\frac{\log(h)}{\log(w)}$	Sparse Binary/Ternary	Binary NTT RLWE (Section 5.1)	$\frac{\log(h) \cdot w \cdot \log(N)}{\log(w) \cdot 64}$ of 64-bit

Table 1: Asymptotic comparisons (omitting constants). n and h denote the LWE secret key length and the Hamming weight of the sparse secret key, respectively, while N represents the RLWE ring dimension (practically the same order as n). The parameter w ($1 \leq w \leq h$) can be set arbitrarily with the indicated tradeoff. The cost per $\mathbb{Z}_Q$ operation is measured by the number of word-size integer operations (either 32-bit or 64-bit). Like other FHE schemes, all results presume circular security.

1.1 Related Work

Gate bootstrapping. Gate bootstrapping was first introduced in FHEW [34] and later improved in TFHE [29] and [69]. There have been works attempting to reduce the number of RGSW-by-RLWE multiplications by having more additions in uniform binary key [105, 15, 55] (called key-unrolling). However, due to such a constraint, the number of RGSW-by-RLWE multiplications can only be reduced by a factor of 2, at the cost of having $4\times$ more additions, thereby providing limited concrete efficiency improvement. Two recent works also explored the sparsity of the LWE keys [61, 10]. In fact, they focus on the structured sparse secret *same* as our warm-up construction in Section 4.2. However, as discussed in more detail there, such a special type of sparse secret is not well-studied and is susceptible to more efficient attacks. Thus, they achieve relatively limited improvement over standard FHEW/TFHE ($< 1.5\times$).

Another very recent work [53] explores a smoother parameter setting in gate bootstrapping, which can be used in complement to our techniques to offer better performance under more careful parameter selections. Some other works focus on using the NTRU assumption instead of LWE/RLWE assumptions [102, 70, 39].

Large-precision functional bootstrapping. Besides gate bootstrapping, FHEW/TFHE framework also supports functional bootstrapping. To ensure large precision for such functional bootstrappings, there have been many attempts [47, 76, 9, 67, 11]. One state-of-the-art in this line [11] attempts to reduce the number of RGSW-by-RLWE multiplications by exploring the structure inside the function, thereby does not directly apply to gate bootstrapping as our focus. Two other state-of-the-art results [33, 100] attempt to explore different algebraic structures to improve large-precision bootstrappings. In general, their techniques can potentially be used together with ours for large precision, since they are mostly orthogonal to each other. This is left for future work to explore.

Other lines of work in FHE. Another line of work attempts to improve the amortized efficiency of FHE, such as BGV [18], BFV [16, 36], CKKS [28], and batched FHEW/TFHE bootstrapping [89, 48, 90, 74, 75, 80, 81, 99]. However, the latency of bootstrapping still remains high. In this line of work, sparsity of secrets are widely explored [27, 14, 49, 83, 81, 23, 92], but mainly to reduce the noise during modulus switching. by using batching [89, 48, 90, 74, 75, 80, 81, 99]. However, the latency of FHEW/TFHE bootstrapping still remains high.

2 Technical Overview

This section assumes basic familiarity with ring elements (such as their NTT and coefficient representations), LWE/RLWE, and homomorphic operations, which we recall in Section 3.

2.1 Bootstrapping with Sparse LWE Secret

Recalling FHEW/TFHE. The main technical component of FHEW/TFHE is the following task: given an LWE ciphertext $(\vec{a}, b) \in \mathbb{Z}_q^{n+1}$ encrypting a message under $\mathsf{sk}_{\mathsf{LWE}} \in \{0,1\}^n$, compute

$$\mathsf{tarCT} := \mathsf{RLWE}_{\mathsf{msb}}(X^{-b+\langle \vec{a}, \mathsf{sk}_{\mathsf{LWE}} \rangle}) = (c, d) \in \mathcal{R}_Q^2.$$

By $\mathsf{RLWE}_{\mathsf{msb}}$ we mean that

$$-d + c \cdot \mathsf{sk}_{\mathsf{RLWE}} = \lfloor Q/2 \rceil \cdot X^{-b+\langle \vec{a}, \mathsf{sk}_{\mathsf{lwe}} \rangle} + e$$

for some small error e.

To achieve this goal, TFHE relies primarily on RGSW-by-RLWE multiplications: given an RLWE ciphertext $\mathsf{RLWE}_{\mathsf{msb}}(m_1)$ and an RGSW ciphertext $\mathsf{RGSW}(m_2)$, we obtain

$$\mathsf{RLWE}_{\mathsf{msb}}(m_1) \times \mathsf{RGSW}(m_2) \to \mathsf{RLWE}_{\mathsf{msb}}(m_1 m_2).$$

Since $\mathsf{sk}_{\mathsf{LWE}}$ is binary, we can write

$$X^{-b+\langle \vec{a}, \mathsf{sk}_{\mathsf{lwe}} \rangle} = X^{-b} \prod_{i \in [n]} X^{\vec{a}_i \cdot \mathsf{sk}_{\mathsf{LWE}}[i]} = X^{-b} \prod_i \left(\mathsf{sk}_{\mathsf{LWE}}[i] \cdot (X^{\vec{a}_i} - 1) + 1 \right).$$

To obtain tarCT homomorphically, publish bootstrapping keys $\mathsf{btk}_i = \mathsf{RGSW}(\mathsf{sk}_{\mathsf{LWE}}[i])$. Then, compute

$$\mathsf{btk}'_i \leftarrow (X^{\vec{a}[i]} - 1) \cdot \mathsf{btk}_i + 1 = \mathsf{RGSW}(\mathsf{sk}_{\mathsf{LWE}}[i] \cdot (X^{\vec{a}_i} - 1) + 1) \ ,$$

followed by

$$\mathsf{tarCT} = (0, \lfloor Q/2 \rceil \cdot X^{-b}) \times \mathsf{btk}'_0 \times \cdots \times \mathsf{btk}'_{n-1} \ ,$$

where $(0, \lfloor Q/2 \rceil \cdot X^{-b})$ is a trivial ciphertext $\mathsf{RLWE}_{\mathsf{msb}}(X^{-b})$.

This second step contains n sequential RGSW-by-RLWE multiplications, which form the main performance bottleneck of TFHE, while the first step has n monomial-by-RGSW multiplications, admitting efficient implementation[1].

After computing tarCT, the remainder of TFHE bootstrapping is comparatively efficient and remains unchanged in our construction (see Section 4.1). In this overview, we focus on how to compute tarCT using significantly fewer RGSW-by-RLWE multiplications.

Warmup: Structured sparse binary keys. [2] Assume first that the LWE secret key is binary, sparse, and furthermore structured. Specifically, $\mathsf{sk}_{\mathsf{LWE}} \in \{0,1\}^n$ consists of h concatenated chunks, each being a one-hot vector. Let $\mathcal{I} := \{\mathsf{id}_0, \ldots, \mathsf{id}_{h-1}\}$ denote the indices where $\mathsf{sk}_{\mathsf{LWE}}[\mathsf{id}_j] = 1$ for $j \in [h]$.

Since

$$\langle \vec{a}, \mathsf{sk}_{\mathsf{LWE}} \rangle = \sum_{j \in [h]} \vec{a}[\mathsf{id}_j],$$

[1] It can be optimized to n monomial-by-RLWE multiplications; see Algorithm 1. Jumping ahead, in our construction, we keep btk_i in coefficient representation and monomial multiplication can be implemented very efficiently, using e.g., coefficient rotation, or keeping a pointer.

[2] This warm-up is essentially the same as [61] as discussed in Section 1.1.

we show that only h RGSW-by-RLWE multiplications are needed. Suppose we have

$$\mathsf{btk}'_j := \mathsf{RGSW}(X^{\vec{a}[\mathsf{id}_j]\cdot \mathsf{sk}_{\mathsf{LWE}}[\mathsf{id}_j]}),$$

then we can compute

$$(0, \lfloor Q/2 \rceil X^{-b}) \times \mathsf{btk}'_0 \times \cdots \times \mathsf{btk}'_{h-1}.$$

The key question is how to obtain btk'_j from the original bootstrapping keys $\mathsf{btk}_i = \mathsf{RGSW}(\mathsf{sk}_{\mathsf{LWE}}[i])$. For structured secrets, this can be achieved using only *linear* operations:

$$\mathsf{btk}'_j \leftarrow \sum_{\ell \in [n/h]} X^{\vec{a}[\ell + j\cdot n/h]} \cdot \mathsf{btk}_{\ell + j\cdot n/h}.$$

This involves only monomial-by-RGSW multiplications and additions, which are significantly cheaper than RGSW-by-RLWE multiplications (see Section 4.3).

Compared to prior work, this entire procedure reduces the number of expensive multiplications from n to h, yielding substantial concrete improvements since $h \ll n$ even at equivalent security levels.

General sparse binary keys. Although entropic LWE results imply that any sufficiently high-entropy secret distribution remains hard [46, 45, 17], reductions may incur some security loss. Moreover, such structured sparsity may be vulnerable to concretely more efficient attacks. On the other hand, random sparse secrets are widely studied, particularly in the FHE setting (see Section 1.1), and can be evaluated directly using the LWE-estimator [1]. Therefore, it would be more robust to design our construction using such keys instead.

The technique above for structured sparse secret does not directly extend to arbitrary sparse secrets: The difficulty is that linear computation of btk'_j fails without the one-hot structure. Our key observation is that *probabilistic batch codes (PBC)* [5, 103] allow us to transform a general sparse key into an extended key with the one-hot structure. PBC is tool for parallel batch access to a database and can be implemented for example using cuckoo hashing. It enables compiling a database DB into an slightly extended one $\mathsf{DB}' \leftarrow \mathsf{PBC.Ecd}(\mathsf{DB})$ consisting $\tilde{h}$ chunks, such that, h (for simplicity $\tilde{h} = h$ here) arbitrary access $\mathcal{I} \subseteq [|DB|]$ to the original database, can be converted into $\tilde{h}$ parallel access, $\mathcal{I}' \leftarrow \mathsf{PBC.Sched}(\mathcal{I})$, one access per chunk of DB'.

We view the coefficient vector $\vec{a}$ as the database, and PBC encode it $\vec{a}' := \mathsf{PBC.Ecd}(\vec{a})$. By viewing the h-sparse LWE secret $\mathsf{sk}_{\mathsf{LWE}}$ as indicating access to $\vec{a}$ at indices $\mathcal{I} = \{\mathsf{id} \ : \ \mathsf{sk}_{\mathsf{LWE}}[\mathsf{id}] = 1\}$, we can convert it to parallel $\mathcal{I}'$ to chunks of $\vec{a}'$. This view allows us to obtain a new secret $\mathsf{sk}'_{\mathsf{LWE}}$, with $\mathsf{sk}'_{\mathsf{LWE}}[\mathsf{id}] = 1$ iff $\mathsf{id} \in \mathcal{I}'$, which then has the crucial one-hot structure in each chunk. The correctness of PBC ensures that $\langle \vec{a}', \mathsf{sk}'_{\mathsf{LWE}} \rangle = \langle \vec{a}, \mathsf{sk}_{\mathsf{LWE}} \rangle$. Bootstrapping then proceeds as in the warm-up case. Above, we made the simplifying assumption that $h = \tilde{h}$. In the general case $h \leq \tilde{h}$, the conversions then have more technical subtleties that we handle in Section 4.2.

The total cost becomes $\tilde{h}$ RGSW-by-RLWE multiplications and $\tilde{n}$ linear operations. As shown in [5], choosing $\tilde{h} = 1.5h$ and $\tilde{n} = 3n$ yields negligible failure probability and negligible security loss. More aggressive parameters can further improve performance at the cost of $\ll 1$ bit of security loss (see Section 7).

Ternary Keys and Beyond. Our construction naturally extends to more general sparse LWE secrets. Nonzero entries may be drawn from a larger domain (e.g., ternary) at the cost of additional linear operations. Importantly, the number of RGSW-by-RLWE multiplications remains unchanged (see Section 4.2).

Monomial multiplications. Since typically $n \gg h$, linear operations, especially monomial multiplications, may themselves become non-negligible. We describe two optimizations for accelerating monomial-by-RGSW multiplication in Section 4.3.

2.2 Reducing NTT/INTT Depth

FHEW/TFHE bootstrapping requires n levels of NTT/INTT, due to n sequential RGSW-by-RLWE multiplications. This large NTT/INTT depth is a major barrier for optimizing TFHE using modern hardware accelerators: for parameters used by FHEW/TFHE, NTT/INTT has even comparable runtime under GPU and single-core CPU (with AVX-512 acceleration) [106, 12].

Tree-shaped multiplications. One natural idea is to perform the multiplications in a tree shape instead of sequentially. This is, unfortunately, not supported by RGSW-by-RLWE multiplications since they are not commutative. One attempt is to replace these multiplications by BV-like multiplications (recalled in Section 3.3.2), which are commutative, thereby reducing the NTT/INTT depth to $\log(n)$. However, this is often unfavorable in practice, since it comes with a cost, that the number of arithmetic operations becomes significantly larger for two reasons: (1) with $\log(n)$ levels of multiplications, the ciphertext modulus Q becomes $\Omega(n^{\log(n)})$ (making each $\mathbb{Z}_Q$ operation a lot more expensive and the ring dimension larger to accommodate such a large modulus), and (2) such multiplications have an operation called relinearization, which is not only expensive by itself, but also requires the ciphertexts to transform between NTT and coefficient representations, requiring calls to NTT/INTTs.

Utilizing our sparse secret above immediately mitigates the first issue: the depth becomes $\log(h)$ instead of $\log(n)$. However, the second problem remains. Another independent question is to see whether we can further reduce the NTT/INTT depth beyond $\log(h)$.

Relinarization-free multiplications. One natural thought is to get rid of relinearization. However, naively getting rid of relinearization causes the ciphertext size to grow exponentially with the multiplicative depth. To properly remove the need for relinearization, we introduce a new variant of RLWE:

Definition 1 (Binary-NTT RLWE, Informal)**.** *RLWE remains secure even when the secret s is sampled uniformly subject to the restriction $s = s^2$.*

To see why this assumption helps: given two RLWE ciphertexts $\mathsf{RLWE}_{\mathsf{lsb}}(m_1 \in \mathcal{R}_t) = (a_1, b_1) \in \mathcal{R}_Q^2$ satisfying $a_1 s - b_1 = m_1 + t \cdot e_1$, and similarly $\mathsf{RLWE}_{\mathsf{lsb}}(m_2 \in \mathcal{R}_t) = (a_2, b_2) \in \mathcal{R}_Q^2$ satisfying $a_2 s - b_2 = m_2 + t \cdot e_2$, it holds that $b_1 b_2 - (a_1 b_2 + a_2 b_1)s + a_1 a_2 s^2 = m_1 m_2 + E\Delta$ where $E = \Omega(e_1 e_2 + \sqrt{N}(e_1 + e_2) + t(e_1 + e_2))$ (see Appendix A.1 for details on the error E). Therefore, we have $(a_3, b_3) = \mathsf{RLWE}_{\mathsf{lsb}}(m_1 \cdot m_2) \in \mathcal{R}_Q^2$ where $a_3 = a_1 a_2 - a_1 b_2 - a_2 b_1$ and $b_3 = b_1 b_2$. This gets rid of the need for relinearization. Furthermore, note that all these operations can be computed in their NTT representation, thereby, no NTT/INTT is needed in each of such multiplication.

Error control. One issue remaining is that we cannot directly perform such multiplications for $\log(h)$ levels, since the error growth is $\Omega(N^{\log(h)})$, making the parameters unbearable. The standard way of solving this problem is via modulus switching introduced in BGV [18], after every multiplication. However, the error growth during modulus reduction is proportional to the norm of the secret key in its coefficient form, which is small in regular BGV. However, our secret key has to satisfy $s = s^2$ and will therefore have a large norm in the coefficient domain. One way to resolve this problem is to perform key-switching between our secret and secrets with a small norm, before applying modulus switching. Furthermore, we can perform *delayed modulus switching*: only modulus switching after multiplying w ciphertexts for some tunable parameter $w \geq 2$, which allows our ciphertexts to remain in their NTT representation. Then, for h multiplications, $\sim h/w$ key-switching are needed. While $\sim h/w$ can already be tolerable in practice for certain w, we introduce an additional way to get further rid of these key-switching: using the following new assumption, which is (somewhat surprisingly) equivalent to Binary NTT RLWE.

Definition 2 (QH-SS-RLWE, Informal)**.** *RLWE remains secure even when the secret s has small norm and we release a random pair of ring elements (u, v) satisfying $s^2 = us + v$.*

Modulus switching now works well since the secret is once again small in the coefficient domain. To avoid relinearization, we simply use the tuple (u, v) to compute

$$(a_3 \leftarrow a_1a_2u - a_1b_2 - a_2b_1,\ b_3 \leftarrow b_1b_2 - a_1a_2v).$$

Each multiplication gets slightly more expensive than before, but we can still perform the entire procedure in the NTT representation, thereby efficiently. Note that to perform modulus switching, we need to transform the ciphertexts into coefficient representation, but only after multiplying w ciphertexts. Asymptotically, we have an NTT/INTT depth of $\log(h)/\log(w)$. Concretely, the depth is only 3,compared to > 500 for the original FHEW/TFHE, while keeping the number of arithmetic operations similar, thereby a very appealing solution for hardware acceleration.

Cryptanalysis. Lastly, we perform cryptanalysis on our new assumptions. Following a similar idea as in [6], we show that the two new assumptions introduced above are actually equivalent, as long as the small secret in Definition 2 is sampled from the error distribution. We then show a search-to-decision reduction and a worst-case to average-case reduction for these assumptions. Finally, we present evidence that existing attacks against LWE/RLWE do not have any obvious advantage against our variant of RLWE. We also investigate and rule out a few other natural approaches to take advantage of the extra structure of the secret key in our variant. These are investigated in more detail in Section 6.

3 Preliminaries

3.1 Notation

We denote the security parameter by $\lambda \in \mathbb{N}$; all other parameters implicitly depend on λ. The modulus will be denoted by $q \in \mathbb{N}$. We use χ to describe the error distribution for a single coordinate, so it corresponds to a sampling procedure on $\mathbb{Z}_q$. Unless otherwise specified, it is a discrete Gaussian distribution with a mean of 0 and a certain standard deviation. We define rings $\mathcal{R} := \mathbb{Z}[X]/(X^N + 1)$ and $\mathcal{R}_q := \mathcal{R}/q\mathcal{R}$ for ring dimension N and $2N|q - 1$ for modulus q unless otherwise specified. Let $[x] := (0, \ldots, x - 1)$ for $x \in \mathbb{Z}^+$.

3.2 Standard Assumptions

In this section, we describe some standard hardness assumptions and some of their variants our work relies.

For all the assumptions below, we denote the number of samples by $m \in \mathbb{N}$ in assumptions. For the LWE assumptions, we use $n \in \mathbb{N}$ to be the dimension of the secret. For the RLWE assumptions, we use $N \in \mathbb{N}$ as the ring dimension. We assume that n, m, N are polynomial in λ and that $2N|q - 1$.

Definition 3 (Learning With Errors (LWE))**.** *The $\mathsf{LWE}_{\lambda,n,m,q,\mathcal{D},\chi}$ assumption states that no PPT adversary can distinguish between $(\mathbf{A}, \vec{s}\mathbf{A} + \vec{e} \mod q)$ and $(\mathbf{A}, \vec{r})$ where $A \leftarrow_\$ \mathbb{Z}_q^{n\times m}$, $\vec{s} \leftarrow \mathcal{D}$, $\vec{e} \leftarrow \chi^{m\times 1}$ and $\vec{r} \leftarrow \mathbb{Z}_q^{m\times 1}$ with noticeable probability.*

If $\mathcal{D}$ is uniform over $\mathbb{Z}_q^n$, this assumption is the standard LWE assumption. It has been shown in [46, 45, 17], any distribution with sufficient entropy for LWE (the entropic LWE) can be reduced to the standard LWE. For simplicity, we define entropic LWE as LWE. The distribution $\mathcal{D}$ used will be specified.

Definition 4 (Ring Learning with Errors)**.** *The* $\mathsf{RLWE}_{\lambda,N,m,q,\chi}$ *assumption states that no PPT adversary can distinguish between* $\{(a_i, a{\cdot}s{+}e)\}_{i\in[m]}$ *and* $\{(a_i, b_i)\}_{i\in[m]}$ *where* $\{a_i\}_{i\in[m]}, b, s \leftarrow_{\$} \mathcal{D}$ *and* $\vec{e} \leftarrow_{\$} \chi^N; e = \sum \vec{e}[i]X^i \in \mathcal{R}_q$ *with noticeable advantage.*

Similarly, we define (entropic) RLWE, and if $\mathcal{D}$ is uniform over $\mathcal{R}_q$, it is the standard RLWE. Unlike LWE, the relationship between entropic RLWE and standard RLWE is unclear. However, in practice, for most FHE constructions, $\mathcal{D}$ is defined to be ternary (i.e., s coefficients uniform over $\{-1, 0, 1\}^N$). Later, when we use a different $\mathcal{D}$, we will specify it accordingly.

3.3 Basics of FHE

This section provides a quick summary of several different encryption schemes used for FHE and how certain homomorphic operations are performed.

3.3.1 Encryption schemes

LWE ciphertexts. An LWE sample can be used as a one time pad to encrypt a message. To ensure correctness, the error term should be separated from the message. Thus, an LWE ciphertext encrypts a message $m \in \mathbb{Z}_t$ for plaintext space $t \ll q$ under secret s has the form $(\vec{a}, b = \langle \vec{a}, s\rangle + \Delta \cdot m + e) \in \mathbb{Z}_q^{n+1}$ where $\vec{a} \leftarrow_{\$} \mathbb{Z}_q^n, e \leftarrow_{\$} \chi$ for some error distribution χ, and $\Delta := \lfloor q/t \rceil$. We use $\mathsf{LWE}^{s,q}_{\mathsf{msb}}(m)$ to represent such a ciphertext, and omit s and/or q if clear.

RLWE ciphertexts. Similarly, a RLWE ciphertext encryps a message $m \in \mathcal{R}_t$ under secret s has the form $(a, b = as + e + \Delta \cdot m) \in \mathcal{R}_q^2$ where $a \leftarrow_{\$} \mathcal{R}_q$, $e \leftarrow_{\$} \chi$, and $\Delta := \lfloor q/t \rceil$. We use $\mathsf{RLWE}^{s,q}_{\mathsf{msb}}(m)$ to represent such a ciphertext.

Instead of encoding the messages in the most significant bits, we could alternatively use least significant bits, which results in a ciphertext of the form $(a, b = as + t \cdot e + m) \in \mathcal{R}_q^2$. We use $\mathsf{RLWE}^{s,q}_{\mathsf{lsb}}(m)$ to represent such a ciphertext.

In some cases, it is easier to work with a message in $m \in \mathcal{R}_q$ directly, which gives a ciphertext of the form $(a, b = as{+}e{+}m) \in \mathcal{R}_q^2$. We use $\mathsf{RLWE}^{s,q}(m)$ to represent such a ciphertext. Again, s, q might be omitted if clear.

RLWE' ciphertexts. This serves as a stepping stone for the later RGSW ciphertexts. Let $2 \le B \le q \in \mathbb{Z}$ be a fixed parameters and $\mathsf{dig} := \lfloor \log_B(q) \rfloor$. An RLWE' ciphertexts is a vector of RLWE ciphertexts, $(\mathsf{RLWE}(m), \mathsf{RLWE}(Bm), \ldots, \mathsf{RLWE}(B^{\mathsf{dig}-1}m))$, and we use $\mathsf{RLWE}'(m)$ to denote such a ciphertext.

RGSW ciphertexxts. An RGSW ciphertext contains two RLWE' ciphertexts of form $(\mathsf{RLWE}'(-s{\cdot}m), \mathsf{RLWE}'(m))$, where s is the secret key used for encrypt the RLWE' ciphertexts (thereby requiring a circular security assumption [88]). We use $\mathsf{RGSW}(m)$ to denote such a ciphertext.

3.3.2 Homomorphic Operations

Now we recall some of the most common operations used in FHE.

Number Theoretic Transform (NTT) and its Inverse (INTT). All of the ring-based encryption schemes above use ring elements in their ciphertexts. The representation of these ring elements affects the efficiency of the various operations below.

Two representations are commonly used in FHE schemes. We can represent $a \in \mathcal{R}_q$ (again, $2N|q-1$ for ring dimension N) as $\sum_{i=0}^{N-1} a_i X^i$, and use the vector $\{\vec{a}_i\}$ to describe a. This is called the *coefficient representation/form.* Alternatively, we can identify a by the values it takes at N distinct points. If ζ is a primitive $2N^{th}$ root unity modulo q, the *NTT representation/form* of a is the vector $[a(\zeta), a(\zeta^3), a(\zeta^5), \ldots, a(\zeta^{2N-1})]$.

We use $\hat{x}$ to represent the NTT form of a ring element $x \in \mathcal{R}_q$, and use $\mathsf{NTT}(x) \to \hat{x}$ and $\mathsf{INTT}(\hat{x}) \to x$ as the interfaces for NTT/INTT. The cost of NTT and INTT is $N\log(N)/2$ of $\mathbb{Z}_q$ multiplications and $N\log(N)$ of $\mathbb{Z}_q$ additions.

For any operation involving two ring elements, we need them to be represented in the same format. Two elements can be added efficiently in $O(N)$ time in either representation. However, multiplication takes $O(N^2)$ (using the naive schoolbook algorithm) or $O(N^{1.5})$ (using Karatsuba algorithm) operations in the coefficient representation but only N $\mathbb{Z}_q$ operations in the NTT representation. On the other hand, sampling the error term in any of the ring-based cryptosystems is feasible only in the coefficient representation. Unless stated otherwise, assume all the following ring elements are in coefficient form. During multiplication, by default, we transform the inputs from coefficient representation to NTT representation, multiply them, and transform back to coefficient form using NTT and INTT functions above.
For all the operations below, unless otherwise stated, the ciphertext modulus and secret keys remain unchanged.

Additions. We use $\mathsf{Enc}(m_1) + \mathsf{Enc}(m_2) \to \mathsf{Enc}(m_1 + m_2)$ as the interface to additions, adding two ciphertexts together, where $\mathsf{Enc} \in \{\mathsf{LWE}, \mathsf{RLWE}_{\mathsf{msb}}, \mathsf{RLWE}_{\mathsf{lsb}}, \mathsf{RLWE}', \mathsf{RGSW}'\}$. Additions can be done in simply with $|\mathsf{Enc}(\cdot)|$ operations where $|\mathsf{Enc}(\cdot)|$ is the number of $\mathbb{Z}_q$ elements a ciphertext $\mathsf{Enc}(\cdot)$ has.

Monomial multiplications. One can multiply an RLWE ciphertext encrypting message m with a monomial $X^c \in \mathcal{R}$ where $c \in [N]$, which generates a ciphertext encrypting $X^c \cdot m$, denoted as $X^c \times \mathsf{Enc}(m) \to \mathsf{Enc}(X^c \cdot m)$, where $\mathsf{Enc} \in \{\mathsf{RLWE}, \mathsf{RLWE}_{\mathsf{msb}}, \mathsf{RLWE}_{\mathsf{lsb}}, \mathsf{RLWE}', \mathsf{RGSW}\}$. The cost of monomial multiplication is $|\mathsf{Enc}(\cdot)|$ of $\mathbb{Z}_q$ multiplications, if both inpts are in NTT forms. If both are in coefficient forms, it can be achieved essentially for free for modern computers, which we discuss in more detail in Section 4.3.

Plaintext multiplications. Directly multiplying a plaintext $c \in \mathcal{R}_q$ with an RLWE ciphertext as above can incur as large as $O(q \cdot N)$ of noise growth, which breaks correctness. Thus, either $||c||_\infty$ needs to be small such that the noise is still sufficiently small, or we can multiply c (in coefficient form) by RLWE' (in either coefficient or NTT form) to generate a RLWE ciphertext: (1) decompose c into $(c_0, \ldots, c_{\mathsf{dig}-1})$ satisfying $c = \sum_{i\in[\mathsf{dig}]} c_i \cdot B^i$. All of the c_i have infinity norm bounded by B; (2) apply NTT to each c_i and obtain $\hat{c}_i$ and transform $\mathsf{RLWE}'(m)$ to its NTT form if necessary; and (3) compute $\mathsf{RLWE}(c\cdot m) \leftarrow \sum_{i\in[\mathsf{dig}]} \hat{c}_i \times \mathsf{RLWE}'(m)[i]$; (4) an INTT is needed if the output is in coefficient form.

We use $c \odot \mathsf{RLWE}'(m) \to \mathsf{RLWE}(c \cdot m)$ to represent such plaintext multiplications. This incurs a noise growth of only $O(\mathsf{dig} \cdot B \cdot \sqrt{N})$. The cost of this operation is dominated by the NTT/INTTs, which contains $3\mathsf{dig}+2$ NTT/INTTs if both inputs and outputs are in coefficient forms, and dig of NTTs if both are in NTT forms.

BV-like ciphertext multiplications. We can multiply two ciphertexts by following the recipe of [19]. This strategy requires an additional input called an evaluation key, which is defined as $\mathsf{evk} := \mathsf{RLWE}'(s^2)$ where s is the secret key. The evaluation key is always represented in the NTT domain. Given two input ciphertexts $\mathsf{RLWE}_{\mathsf{lsb}}(m_1) = (a_1, b_1)$, $\mathsf{RLWE}_{\mathsf{lsb}}(m_2) = (a_2, b_2)$ and $\mathsf{evk} = \mathsf{RLWE}'(s^2)$ (for simplicity, in NTT form, but can also be in coefficient form), the multiplication: (1) computes $(x = a_1a_2, y = a_1b_2 + a_2b_1, z = b_1b_2) \in \mathcal{R}_q^3$ and (2) computes $(y, z) + x \odot \mathsf{evk}$, which yields $\mathsf{RLWE}_{\mathsf{lsb}}(m_1m_2)$. This gives the interface $\mathsf{RLWE}_{\mathsf{lsb}}(m_1) \times \mathsf{RLWE}_{\mathsf{lsb}}(m_2) \to \mathsf{RLWE}_{\mathsf{lsb}}(m_1m_2)$.

Since this operation is relatively complicated, we show more details, including why this is correct in Appendix A.1, for the interested readers. The summary above, however, is sufficient for understanding our protocols. The cost is dominated by $\mathsf{dig}+1$ of NTT/INTTs (transforming x back to its coefficient representation, decomposing it, and transforming it into its NTT

representation). If the two inputs are in their coefficient forms, 4 additional NTTs are needed.

GSW-like multiplication. The RGSW ciphertexts can be used for a different style of ciphertext multiplication [44], also called the RGSW-by-RLWE multiplications. Given $\mathsf{RLWE}_{\mathsf{msb}}(m_1) = (a, b)$ (in coefficient form) and $\mathsf{RGSW}(m_2) = (\mathsf{RLWE}'(-s \cdot m_2), \mathsf{RLWE}'(s \cdot m_2))$ (in either form), we aim to produce $\mathsf{RLWE}_{\mathsf{msb}}(m_1 \cdot m_2)$ (in coefficient form). This is achieved by computing $a \odot \mathsf{RLWE}'(-s \cdot m_2) + b \odot \mathsf{RLWE}'(m_2)$, which produces $\mathsf{RLWE}_{\mathsf{msb}}(m_1 \cdot m_2)$. Again, please see Appendix A.1 for details.

The cost is dominated by $6\mathsf{dig} + 4$ NTT/INTTs if RGSW is in coefficient form (simply achieved by two plaintext multiplications defined above), and $2\mathsf{dig} + 4$ NTT/INTTs if RGSW is in NTT form.

Key switching. We can change the secret key used for a ciphertext. This requires an additional input called a key switching key which encrypts the old secret key s_1 using the new secret key s_2 as follows: $\mathsf{ksk} := \mathsf{RLWE}'^{s_2}(s_1)$. The key switching key is always represented in the NTT domain. Given input ciphertext $\mathsf{RLWE}^{s_1}(m) = (a_1, b_1)$, $\mathsf{KeySW}(\mathsf{ksk}, \mathsf{RLWE}^{s_1}(m)) \to \mathsf{RLWE}^{s_2}(m)$ does the following: compute $(0, b) + -a \odot \mathsf{ksk}$. Please see Appendix A.1 for correctness. The cost is essentially a plaintext multiplication described above (with ciphertext in NTT form), which is thus $\mathsf{dig} + 1$ of NTT/INTTs.

Modulus switching. We can change the modulus of ciphertexts without affecting the plaintext. We define $\mathsf{ModSW}(\mathsf{Enc}^q(m)) \to \mathsf{Enc}^{q'}(m)$ where $\mathsf{Enc} \in \{\mathsf{RLWE}_{\mathsf{msb}}, \mathsf{RLWE}_{\mathsf{lsb}}\}, q' < q$. For $\mathsf{RLWE}_{\mathsf{msb}}$, it is simply $\lfloor \frac{q'}{q} \mathsf{Enc}^q(m) \rceil$. For $\mathsf{RLWE}_{\mathsf{lsb}}$, it is more complicated. Please see Appendix A.1 for more details. The cost is dominated by 2N real number multiplications and rounding.

Extract. Lastly, we recall "extract", which transforms an RLWE ciphertext encrypting $m := \sum_{j \in [N]} m_j X^j \in \mathcal{R}_t$ under secret $s := \sum_{j \in [N]} s_j X^j \in \mathcal{R}$ into an LWE ciphertext encrypting $m_i \in \mathbb{Z}_t$ under $\vec{s} := (s_0, \ldots, s_{N-1})$ which is the vector containing, for $i \in [N]$, with the same ciphertext modulus. The procedure is denoted as $\mathsf{Extract}(\mathsf{RLWE}^s_{\mathsf{msb}}(m), i) \to \mathsf{LWE}^{\vec{s}}_{\mathsf{msb}}(m_i)$. $i = 0$ is omitted by default.

3.4 Probablistic Batch Code

Probablistic batch code (PBC) was introduced in [5] as a way to compile a private information retrieval (PIR) procedure to a multi-query PIR procedure. Here, we recall its definition [5, 103].

Definition 5. *A (n, c, h, k, ϵ)-Probabilistic batch code (PBC) has the following four PPT algorithms:*

- $C_1, \ldots, C_k \leftarrow \mathsf{Ecd}(\mathsf{pp}, \mathsf{DB})$: *takes a public parameter* pp *and a database* DB *of n entries and output k buckets such that the total size of all buckets is at most $c \cdot n$. Furthermore, each $C_i[j]$ must be one of the n entries in* DB.
- $i_1, \ldots, i_k \leftarrow \mathsf{Sched}(\mathsf{pp}, H)$: *takes a public parameter* pp *and a query H of h distinct elements in $[n]$, and outputs k scheduled indices: i.e., $i_j \in [|C_j|]$ or $i_j = \perp$.*
- $\mathsf{res} \leftarrow \mathsf{Dcd}(\mathsf{pp}, H, C_1[i_1], \ldots, C_k[i_k])$*: takes a query H of h distinct elements in $[n]$, and the encoded elements indexed by the scheduled indices(and if $i_j = \perp, C_j[i_j] = \perp$).*

Furthermore, the PBC has error at most ϵ if, for all database DB *and queries H of h distinct elements, it holds that: let $C_1, \ldots, C_k \leftarrow \mathsf{Ecd}(\mathsf{pp}, \mathsf{DB})$, $i_1, \ldots, i_k \leftarrow \mathsf{Sched}(\mathsf{pp}, H)$, $\mathsf{res} \leftarrow \mathsf{Dcd}(\mathsf{pp}, H, C_1[i_1], \ldots, C_k[i_k])$, then: $\Pr[\mathsf{res} = H] \geq 1 - \epsilon$.*

Lastly, we say that PBC is canonically decodable if, for the same quantifiers above, there exists a permutation Π such that: let C be a vector of all the non-$\perp$ element of $C_1[i_1], \ldots, C_k[i_k]$, $\mathsf{res} = \Pi(C)$.

Compared to the definition in [103], we fix one parameter $t = 1$ (where t is to allow Sched to output $I_1, \ldots, I_k$ being k sets each with at most t indices), and define an additional "canonically decodable" property. Both of which are useful for our later use.

Realization of PBC using cuckoo hashing. In [5, 103], PBC is realized using cuckoo hashing. In summary, it works as follows: with c hash functions (which do not need to be cryptographic), Ecd duplicates each of the n database elements c times and assign them into one of the k buckets according to the hash functions. Given the query H, the schedule function uses cuckoo hashing (with these c functions) to find $i_1, \ldots, i_k$ such that for all $h \in H$ there exists j such that $C_j[i_j] = h$. Note that in this case, only h out of k indices are valid and the rest are just $\perp$.

Since this idea is widely used for batch PIR [5, 3, 103, 21], and we use only PBC as a black-box, we refer the readers to [5, Section 4.2] and [103, Section 7.1] for details.

4 Faster FHEW/TFHE Bootstrapping with Sparse Keys

This section shows how (sparse) binary secrets can help improve the efficiency of FHEW/TFHE bootstrapping.

4.1 Recalling the FHEW/TFHE Bootstrapping Paradigm

We begin by recalling the FHEW/TFHE cryptosystems [34, 29, 69].

Workflow. The core component of the FHEW/TFHE cryptosystem is the bootstrapping procedure. It takes as input an LWE ciphertext $\mathsf{ct} = \mathsf{LWE}_{\mathsf{msb}}(m) \in \mathbb{Z}_q^{n+1} = (\vec{a}, b)$, where q is a power of two, encrypting $m \in \{0, 1, 2\}$ embedded in $\mathbb{Z}_4$, with error e, under a secret key $\mathsf{sk}_{\mathsf{LWE}} \in \mathbb{Z}_q^n$ (typically short, e.g., binary or ternary). The procedure outputs another LWE ciphertext $\mathsf{ct}' = \mathsf{LWE}_{\mathsf{msb}}(m') \in \mathbb{Z}_q^{n+1}$, where $m' = f(m)$ for some negacyclic[3] function f representing the NAND gate with respect to $m_1, m_2 \in \{0, 1\}$ such that $m_1 + m_2 = m$: $f(0) = 1$, $f(1) = 1$, $f(2) = 0$, and $f(3) = 0$. Indeed, when $m = 0$ or $m = 1$, at least one of m_1, m_2 is zero, implying $\mathsf{NAND}(m_1, m_2) = 1$.[4] The output ciphertext ct' is also encrypted under $\mathsf{sk}_{\mathsf{LWE}}$ and has error e' that is *independent* of the error e.

Bootstrapping procedure. We now describe the core bootstrapping procedure in more detail. We first introduce the required (public) keys. The bootstrapping key btk consists of a vector of ciphertexts encrypted under a different secret key $\mathsf{sk}_{\mathsf{RLWE}} \in \mathcal{R}_Q$, for some modulus $Q \gg q$ and a ring $\mathcal{R}$ of dimension $N = q/2$ (or any $N \geq q/2$ such that $\frac{q}{2} \mid N$): $\mathsf{btk}[i] = \mathsf{RGSW}(\mathsf{sk}_{\mathsf{LWE}}[i])$ under $\mathsf{sk}_{\mathsf{RLWE}}$. In addition, a key switching key ksk is required to switch ciphertexts from $\mathsf{sk}_{\mathsf{RLWE}}$ to $\mathsf{sk}_{\mathsf{LWE}}$ (see Section 3.3.2).

The main objective of bootstrapping is to obtain

$$\mathsf{tarCT} := \mathsf{RLWE}_{\mathsf{msb}}\big(X^{-b+\langle \vec{a}, \mathsf{sk}_{\mathsf{LWE}} \rangle}\big) \in \mathcal{R}_Q^2.$$

In short, TFHE encrypts $\mathsf{btk}_i \leftarrow \mathsf{RGSW}(\mathsf{sk}_{\mathsf{LWE}}[i])$ and computes $\mathsf{tarCT} \leftarrow (0, X^{-b}) \times \mathsf{btk}'_0 \times \cdots \times \mathsf{btk}'_{n-1}$, where $\mathsf{btk}'_i \leftarrow (X^{\vec{a}[i]} - 1) \cdot \mathsf{btk}_i + 1$, $(0, X^{-b})$ is a trivial RLWE ciphertext of X^{-b}.[5] It is easy to verify that btk'_i encrypts $X^{\vec{a}[i]}$ if $\mathsf{sk}_{\mathsf{LWE}}[i] = 1$ and 1 otherwise. which therefore gives the requisite tarCT ciphertext. In total, this step takes n RGSW-by-RLWE multiplications (Section 3.3.2), thereby costly: taking $\Omega(nN \log(N))$ of $\mathbb{Z}_q$ multiplications. Note that further,

[3] Meaning that $f(x) = -f(x + p/2) \in \mathbb{Z}_p$ for f being a function of x over $\mathbb{Z}_p$.

[4] The same framework applies to other binary gates. We focus on NAND since it is universal.

[5] Note that this step contains n of monomial-by-RGSW multiplications, which can be simplified to n monomial-by-RLWE multiplications as shown in Algorithm 1.

Algorithm 1 FHEW/TFHE Bootstrapping

procedure $\mathsf{KeyGen}(\mathsf{sk}_{\mathsf{LWE}}, \mathsf{sk}_{\mathsf{RLWE}})$
  $\mathsf{btk}_i \leftarrow \mathsf{RGSW}^{\mathsf{sk}}_{\mathsf{RLWE}}(\mathsf{sk}_{\mathsf{LWE}}[i])$ for $i \in [n]$ (with parameters B, dig, see Section 3.3.1).
  $\mathsf{ksk} \leftarrow \mathsf{KeySW}(\mathsf{sk}_{\mathsf{LWE}}, \mathsf{sk}_{\mathsf{RLWE}}) \in \mathcal{R}^{\mathsf{dig}\times 2}$ (where $\mathsf{sk}_{\mathsf{LWE}}$ is embedded into $\mathcal{R}$ and see Section 3.3.1 for dig)
procedure $\mathsf{BlindRot}((\vec{a}, b)\mathbb{Z}_q^{n+1}, (\mathsf{btk}_i)_{i\in[n]}, \mathsf{ksk})$
  $P(X) := \sum_{i\in[N]} p_i \cdot X^i$ where $p_i = 1$ if $i \in [-Q/8, 3Q/8)$ and $p_i = 0$ otherwise.
  $\mathsf{ACC} \leftarrow (0, P(X) \cdot X^{-b}) \in \mathcal{R}_Q^2$
  for $j \in [n]$ **do**
    $\mathsf{ACC} \leftarrow (X^{\vec{a}[j]} - 1) \times \mathsf{ACC} \times \mathsf{btk}'[j] + \mathsf{ACC}$
  return ACC
procedure $\mathsf{Boot}(\mathsf{ct} = \mathsf{LWE}_{\mathsf{msb}}(m) \in \mathbb{Z}_q^{n+1}, \mathsf{btk} := (\mathsf{btk}_i)_{i\in[n]}, \mathsf{ksk})$
  $\mathsf{tarCT} \leftarrow \mathsf{BlindRot}(\mathsf{ct}, \mathsf{btk})$
  $\mathsf{ct}_{\mathsf{res}} \leftarrow \mathsf{tarCT} + Q/8$
  return $\mathsf{ct}' \leftarrow \mathsf{ModSW}(\mathsf{Extract}(\mathsf{KeySW}(\mathsf{ksk}, \mathsf{ct}_{\mathsf{res}}), 0), q) = \mathsf{LWE}_{\mathsf{msb}}(m) \in \mathbb{Z}_q^{n+1}$

since RGSW-by-RLWE multiplications are not commutative, these n multiplications are done sequentially, resulting in n of NTT/INTT depth. Later, we introduce our own method for obtaining tarCT, which constitutes the core improvement over existing approaches.

In fact, this tarCT is a simplification. Instead, the goal is to compute $\mathsf{ct}_{\mathsf{res}} = \mathsf{RLWE}_{\mathsf{msb}}(m' + r(X))$, where $m' = f(m)$ is a constant term in $\mathbb{Z}_4$ (mapping $0, 1$ to 1 and 2 to 0, representing the NAND gate), and $r(X)$ is a polynomial in $\mathcal{R}_4$ with zero constant coefficient. To this end, we define

$$P(X) = \sum_{i\in[N]} p_i X^i,$$

where $p_i = Q/8$ if $i \in [-N/8, 3N/8)$ and $p_i = -Q/8$ otherwise. Then, instead of computing tarCT using $(0, X^{-b})$, use $(0, P(X) \cdot X^{-b})$ as the initial trivial RLWE ciphertext. In this case, $\mathsf{ct}_{\mathsf{res}} = \mathsf{tarCT} + Q/8$ under $\mathsf{sk}_{\mathsf{RLWE}}$, as desired (see our proof of Theorem 3 for details why).

The final step is to obtain an LWE ciphertext encrypting m' under $\mathsf{sk}_{\mathsf{LWE}}$. To compute $\mathsf{ct}' := \mathsf{LWE}_{\mathsf{msb}}(m')$, we proceed as follows: (1) key switch $\mathsf{ct}_{\mathsf{res}}$ from $\mathsf{sk}_{\mathsf{RLWE}}$ to $\mathsf{sk}_{\mathsf{LWE}}$ using KeySW; (2) extract the constant term of the resulting ring ciphertext using $\mathsf{Extract}$; and (3) modulus switch the result from Q to q using ModSW. Formally,

$$\mathsf{ct}' \leftarrow \mathsf{ModSW}(\mathsf{Extract}(\mathsf{KeySW}(\mathsf{ksk}, \mathsf{ct}_{\mathsf{res}} \in \mathcal{R}_Q^2), 0), q) \in \mathbb{Z}_q^{n+1}$$

is an $\mathsf{LWE}_{\mathsf{msb}}(m')$ ciphertext under $\mathsf{sk}_{\mathsf{LWE}}$ whose noise is independent of the error of the input ciphertext ct. This completes the construction. We summarize this procedure in Algorithm 1.

4.2 Leveraging the Sparsity of the LWE Secret Key

In the FHE literature, uniform sparse secret keys are very often used in CKKS (e.g., [27, 14]) and BGV/BFV (e.g., [49, 23]) to ensure small noise growth after modulus switching, whose error grows linearly with the Hamming weight of the secret key. This, in turn, improves the efficiency of bootstrapping. Interestingly, to the best of our knowledge, such keys has not been exploited in FHEW/TFHE. Indeed, as shown in the paradigm above, a sparse secret key does not appear to offer significant benefits beyond the final modulus switching step, whose impact on the overall efficiency of FHEW/TFHE bootstrapping is relatively minor.

This naturally raises the question of whether the sparsity of the secret key can be leveraged in a different manner to improve efficiency. In this section, we introduce such an approach.

Warm-up: structured sparse binary keys. We begin by considering a simplified setting.[6] Suppose that the secret key is not only sparse, but also highly structured. Specifically, consider a binary secret key $\mathsf{sk}_{\mathsf{LWE}}$ of length n and Hamming weight $h \ll n$ (e.g., $h = 42$ and $n = 1024$ for > 120-bit security per LWE-estimator [1]). Furthermore, assume that the secret key is *guaranteed* to have exactly one non-zero entry in each block of size n/h (e.g., $\mathsf{sk}_{\mathsf{LWE}} := \underline{\mathsf{sk}_{\mathsf{LWE}}}_0 \,\|\, \underline{\mathsf{sk}_{\mathsf{LWE}}}_1 \,\|\, \ldots \,\|\, \underline{\mathsf{sk}_{\mathsf{LWE}}}_{31}$, where each $\underline{\mathsf{sk}_{\mathsf{LWE}}}_i \in \{0,1\}^{1024/32}$ is a uniformly random one-hot vector).

Let $\mathcal{I} := \{\mathsf{id}_1, \ldots, \mathsf{id}_h\}$ denote the set of indices corresponding to the non-zero entries of $\mathsf{sk}_{\mathsf{LWE}}$, i.e., $\mathsf{sk}_{\mathsf{LWE}}[\mathsf{id}_j] = 1$. The goal is to obtain

$$\mathsf{tarCT} := \mathsf{RLWE}\big(X^{-b+\langle \vec{a}, \mathsf{sk}_{\mathsf{LWE}}\rangle}\big) = X^{-b} \times \prod_{j\in[h]} \mathsf{ct}\big(X^{\vec{a}[\mathsf{id}_j]\cdot \mathsf{sk}_{\mathsf{LWE}}[\mathsf{id}_j]}\big) = X^{-b} \times \prod_{j\in[h]} \mathsf{ct}\big(X^{\vec{a}[\mathsf{id}_j]}\big),$$

since $\mathsf{sk}_{\mathsf{LWE}}[\bar{\mathsf{id}}] = 0$ for all $\bar{\mathsf{id}} \in [n] \setminus \mathcal{I}$.

Observe that as long as we can obtain $\mathsf{btk}'_j = \mathsf{RGSW}(X^{\vec{a}[\mathsf{id}_j]\cdot \mathsf{sk}_{\mathsf{LWE}}[\mathsf{id}_j]})$ for each $\mathsf{id}_j \in \mathcal{I}$, then tarCT can be computed with only h RGSW-by-RLWE multiplications:

$$\mathsf{tarCT} \leftarrow \mathsf{RLWE}_{\mathsf{msb}}(X^{-b}) \times \mathsf{btk}'_1 \times \cdots \times \mathsf{btk}'_h = \mathsf{RLWE}_{\mathsf{msb}}\big(X^{-b+\sum_{j\in[h]} \vec{a}[\mathsf{id}_j]\cdot \mathsf{sk}_{\mathsf{LWE}}[\mathsf{id}_j]}\big),$$

where $\mathsf{RLWE}_{\mathsf{msb}}(X^{-b})$ is a trivial ciphertext of the form $(0, \Delta \cdot X^{-b}) \in \mathcal{R}_q^2$.

The remaining question is how to obtain btk'_j. Since each block of size n/h contains exactly one non-zero secret coefficient, we can compute btk'_j via summation:

$$\mathsf{btk}'_j \leftarrow \sum_{\ell\in[n/h]} X^{\vec{a}[\ell + j\cdot n/h]} \cdot \mathsf{btk}_{\ell + j\cdot n/h},$$

which yields $\mathsf{btk}'_j = \mathsf{RGSW}(X^{\vec{a}[\mathsf{id}_j]\cdot \mathsf{sk}_{\mathsf{LWE}}[\mathsf{id}_j]})$. The remainder of the bootstrapping process remains identical to prior work: key switching, extraction of the constant term, and modulus switching.

Compared to prior approaches, which require n ciphertext multiplications and n scalar multiplications (monomial multiplications; see Section 3.3.2), , this structured secret key reduces the complexity to h ciphertext multiplications, n scalar multiplications, and n ciphertext additions. Furthermore, as we will discuss in Section 4.3, the n scalar multiplications can be further eliminated, resulting in a total cost of only h ciphertext multiplications together with n ciphertext additions. As mentioned in Section 3.3.2, ciphertext additions are essentially free compared to ciphertext multiplications. Thus, the overall saving is roughly a factor of n/h (approximately 5–20×, as shown in Section 7.1).

From a theoretical standpoint, this already suffices, since, as discussed in Section 3.2, LWE with such a secret distribution (or general entropic LWE) is as secure as standard LWE. Nevertheless, choosing concrete parameters becomes much more delicate: since this secret key distribution is somewhat specialized, a more systematic study might be necessary for its concrete security. Thus, we next describe how to extend our approach to general sparse binary secrets, a much more widely used assumption.

General sparse binary secrets. We now consider general sparse binary secret keys without the additional structure assumed above. The main challenge of general sparse secrets comes from the fact that the h non-zero elements are uniformly distributed over the n secret elements.

[6] This warm-up is essentially the same as [61] as discussed in Section 1.1.

This makes it hard to use additions to reduce multiplications as before. However, if a general sparse key can somehow be reduced to a structured sparse key, we can use the exact same idea above. Luckily, we observe that this is indeed possible: a general sparse binary secret can be "encoded" into a secret key that is *almost* of the structured form described above.

Concretely, we adopt the (canonically decodable) probabilistic batch code (PBC) technique introduced in [5], which we recall in Section 3.4. At a high level, PBC maps a database DB of n elements into h' smaller databases with total size $n' = c \cdot n$ for a small constant c. To retrieve h elements from DB, one retrieves at most one element from each smaller database. This functionality closely matches our needs: partitioning $\mathsf{sk}_{\mathsf{LWE}}$ into h' smaller vectors, each containing *at most* one non-zero entry (as opposed to exactly one non-zero entry in the structured case, which we address below).

More formally, given $\mathsf{sk}_{\mathsf{LWE}} \in \{0,1\}^n$ with Hamming weight h, with $\mathcal{I}$ denoting the set of indices where $\mathsf{sk}_{\mathsf{LWE}}$ equals one. We interpret $[n]$ as a database DB (which is later used to assign $\mathsf{sk}_{\mathsf{LWE}}$ with length n accordingly) in PBC and encode it into h' buckets $C_1, \ldots, C_{h'} \leftarrow \mathsf{Ecd}(\mathsf{pp}, \mathsf{sk}_{\mathsf{LWE}})$, where each bucket is a vector of indices. Then, $i_1, \ldots, i_{h'} \leftarrow \mathsf{Sched}(\mathsf{pp}, \mathcal{I})$ produces h' indices such that for every $\mathsf{id} \in \mathcal{I}$, there exists a bucket C_j with $C_j[i_j] = \mathsf{id}$. The remaining $h' - h$ indices are set to $\perp$ (i.e., there exist indices i_j such that $i_j = \perp$). For each bucket C_j, we construct a vector $\underline{\mathsf{sk}_{\mathsf{LWE}}}_j$, such that, $\underline{\mathsf{sk}_{\mathsf{LWE}}}_j[i_j] = 1$ and $\underline{\mathsf{sk}_{\mathsf{LWE}}}_j[i] = 0$ for all other coordinates $i \neq i_j$ (and if $i_j = \perp$, $\underline{\mathsf{sk}_{\mathsf{LWE}}}_j = \vec{0}$). We also need to construct h' of $\underline{\vec{a}}_j$'s such that $\sum_j \langle \underline{\vec{a}}_j, \underline{\mathsf{sk}_{\mathsf{LWE}}}_j \rangle = \langle \vec{a}, \mathsf{sk}_{\mathsf{LWE}} \rangle$, which is straightforward since C_j's are public: $\underline{\vec{a}}_j[i] = \vec{a}[C_j[i]]$ for $i \in [|C_j|]$.

If there is no i_j equal to $\perp$ (i.e., $h' = h$), this encoded secret key has exactly the same format as the structured sparse key considered earlier, and the previous technique applies directly. The remaining challenge is handling the case where some $i_j = \perp$.

For any bucket j with $i_j = \perp$, recall that $\underline{\mathsf{sk}_{\mathsf{LWE}}}_j = \vec{0}$. For each bucket j, we have $\mathsf{btk}_{j,i} = \mathsf{RGSW}(\underline{\mathsf{sk}_{\mathsf{LWE}}}_j[i])$. Applying the same method as in the structured case yields

$$\mathsf{btk}'_j \leftarrow \sum_i \mathsf{btk}_{j,i} \cdot X^{\vec{a}_{C_j}[i]}.$$

Consequently, $\mathsf{btk}'_j = \mathsf{RGSW}(0)$ whenever $i_j = \perp$, which would cause the final tarCT to always encrypt zero. To avoid this, we instead need $\mathsf{btk}'_j = \mathsf{RGSW}(1)$ in this case. To achieve this, for each bucket $(\mathsf{btk}_{j,i})_{i \in |C_j|}$, we append a dummy key $\mathsf{btk}_{j,\mathsf{dummy}}$, generated as $\mathsf{RGSW}(1)$ if $\underline{\mathsf{sk}_{\mathsf{LWE}}}_j = \vec{0}$ and as $\mathsf{RGSW}(0)$ otherwise. When computing btk'_j, this dummy key is added at the end, ensuring that $\mathsf{btk}'_j = \mathsf{RGSW}(1)$ when $i_j = \perp$, while leaving btk'_j unchanged when $i_j \neq \perp$.

This completes the construction of our PBC-based compiler for general sparse binary secrets. In terms of efficiency, the resulting construction requires h' ciphertext multiplications, together with $n' = c \cdot n$ scalar multiplications and ciphertext additions. In practice, parameters such as $h' = 1.5h$ and $c = 3$ are commonly used [5, 3, 103] to ensure negligible failure probability, which remains significantly smaller than the cost of the original FHEW/TFHE paradigm. In Section 7, we further discuss how to reduce h' at the cost of leaking less than one bit of information (and, as noted in Section 3.2, LWE is leakage-resistant).

We formalize the resulting compiled paradigm with a sparse binary secret key in Algorithm 2.

Theorem 3. *Let $n, N, q, Q > 0$, for all $\mathsf{sk}_{\mathsf{LWE}} \in \{0,1\}^n$ with hamming weight h, for any $m \in \{0,1,2\} \subset \mathbb{Z}_4$, and let $\mathsf{ct} \leftarrow (\vec{a}, b := \langle \vec{a}, \mathsf{sk}_{\mathsf{LWE}} \rangle - m \cdot \lfloor q/p \rceil + e)$ for any $e \leq \lfloor q/8 \rfloor$; then, let $\mathcal{R} := \mathbb{Z}[X]/(X^N + 1)$ and for all $\mathsf{sk}_{\mathsf{RLWE}} \in \mathcal{R}_2$, for all $(\mathsf{btk}, \mathsf{ksk}, r)$ in the support of $\mathsf{KeyGen}(\mathsf{sk}_{\mathsf{LWE}}, \mathsf{sk}_{\mathsf{RLWE}})$, $\mathsf{ct}' = (\vec{a}', b') \leftarrow \mathsf{FHEWTFHEBoot}(\mathsf{ct}, \mathsf{btk}, \mathsf{ksk}, r)$ (both algorithms in Algorithm 2), then let $E \leftarrow [b' + \langle \vec{a}', \mathsf{sk}_{\mathsf{LWE}} \rangle - f(m) \cdot \lfloor q/4 \rceil \in \mathbb{Z}_q$ where $f(0) = f(1) = 1$ and*

$f(2) = 0$; *it holds that* $|E| = O(\frac{q}{Q}(\sigma_{br} + \sigma_{ks}) + \sigma_{ms})$ *where* $\sigma_{br} = O(\sqrt{\mathsf{dig} \cdot k \cdot N} \cdot B \cdot \sigma')$, $\sigma_{ks} = O(\sqrt{\mathsf{dig} \cdot N} \cdot B \cdot \mathsf{dig} \cdot \sigma)$, *and* $\sigma_{ms} = O(\sqrt{h})$.

Proof. The correctness is straightforward, ignoring the noise: by the fact that PBC is a canonically decodable PBC (Definition 5), it holds that: let $\mathcal{I}$ denote the set of indices at which $\mathsf{sk}_{\mathsf{LWE}}$ is non-zero, and let $\mathcal{I}' := C_j[i_j] | j \in [h'], i_j \neq \perp$; then $\mathcal{I} = \mathcal{I}'$ except for $\mathsf{negl}(\lambda)$ probability by the correctness of PBC. Then,

$$\begin{aligned}\mathsf{tarCT} &= \mathsf{RLWE}_{\mathsf{msb}}(P(X) \cdot X^{-b+\sum_{i\in\mathcal{I}} \vec{a}[i]\cdot \mathsf{sk}_{\mathsf{LWE}}[i]}) \\ &= \mathsf{RLWE}_{\mathsf{msb}}(P(X) \cdot X^{-b+\sum_{i\in[n]} \vec{a}[i]\cdot \mathsf{sk}_{\mathsf{LWE}}[i]})\end{aligned}$$

under $\mathsf{sk}_{\mathsf{RLWE}}$ where m is the input value and $m_i \in \{0,1\}$.Specifically, if $m \in \{0,1\}$, $-b + \langle \vec{a}, \mathsf{sk}_{\mathsf{LWE}} \rangle \in (-q/8, 3q/8]$ thereby $P(X) \cdot X^{-b+\sum_{i\in[n]} \vec{a}[i]\cdot \mathsf{sk}_{\mathsf{LWE}}[i]} = Q/8$ at the constant term. Otherwise, $-b + \langle \vec{a}, \mathsf{sk}_{\mathsf{LWE}} \rangle \in (3q/8, 7q/8]$, thereby $P(X) \cdot X^{-b+\sum_{i\in[n]} \vec{a}[i]\cdot \mathsf{sk}_{\mathsf{LWE}}[i]} = -Q/8$ at the constant term. Then, by adding the result by $Q/8$, we have $\mathsf{ct}_{\mathsf{res}} = \mathsf{RLWE}_{\mathsf{msb}}(f(m) + \sum_{i\in[1,N-1]} m_i \cdot X^i) \in \mathcal{R}_Q^2$

Lastly, using KeySW, we switch $\mathsf{ct}_{\mathsf{res}}$ to to be under $\mathsf{sk}_{\mathsf{LWE}}$, and $\mathsf{Extract}$ extracts out an LWE ciphertext encrypting the constant term $f(m)$, and ModSW switches the ciphertext to be in $\mathbb{Z}_q$, as the regular TFHE construction [29].

The only thing left is the noise analysis. Let the $\mathsf{btk}_j[i]$'s and ksk error have standard deviation of σ. Then, it is easy to see that $\mathsf{btk}'[j]$ has error of standard deviation bounded $\sigma'_j = \sqrt{|C_j|} \cdot \sigma \leq \sqrt{n \cdot c} \cdot \sigma$ (which is in practice smaller since each bucket has $\approx c \cdot n/h$ elements instead of $c \cdot n$ elements) and let $\sigma' = \max(\sigma_j)$. The rest can be seen as a standard TFHE bootstrapping with LWE secret key length k instead of n, which means that the final noise of ct' has standard deviation of $\sigma_{output} = \frac{q}{Q}(\sigma_{br} + \sigma_{ks}) + \sigma_{ms}$ where $\sigma_{br} = O(\sqrt{\mathsf{dig} \cdot k \cdot N} \cdot B \cdot \sigma')$ (where $B^{\mathsf{dig}} \geq Q$ is the parameters for generating RGSW ciphertexts as discussed in Section 3.3.2) being the accumulated noise for the RGSW and $\mathsf{RLWE}_{\mathsf{msb}}$ multiplications, $\sigma_{ks} = O(\sqrt{\mathsf{dig} \cdot N} \cdot B \cdot \mathsf{dig})$ being the key switching error, and $\sigma_{ms} = O(\sqrt{h})$ being the modulus switching error [29]. Thus, the final error is bounded by $O(\sigma_{output})$ except for $\mathsf{negl}(\lambda)$ probability. □

Sparse ternary secrets. Lastly, we briefly comment on how our compiler extends to ternary secrets. For simplicity, we again begin with a structured ternary secret key $\mathsf{sk}_{\mathsf{LWE}}$ of length n and Hamming weight $h \ll n$. Furthermore, the secret key is *guaranteed* to have exactly one non-zero entry per block of size n/h, denoted as $\mathsf{sk}_{\mathsf{LWE}} := \underline{\mathsf{sk}_{\mathsf{LWE}}}_0 \,\|\, \underline{\mathsf{sk}_{\mathsf{LWE}}}_1 \,\|\, \ldots \,\|\, \underline{\mathsf{sk}_{\mathsf{LWE}}}_{h-1}$, as before. Let $\mathcal{I} := \{\mathsf{id}_0, \ldots, \mathsf{id}_{h-1}\}$ denote the indices of the non-zero entries, each of which is either 1 or -1.

As before, the ultimate goal is to obtain $\mathsf{RLWE}\big(P(X) \cdot X^{-b+\sum_{j\in[h]} \vec{a}[\mathsf{id}_j]\cdot \mathsf{sk}_{\mathsf{LWE}}[\mathsf{id}_j]}\big)$. Accordingly, our immediate objective is to compute $\mathsf{btk}'_j = \mathsf{RLWE}\big(X^{\vec{a}[\mathsf{id}_j]\cdot \mathsf{sk}_{\mathsf{LWE}}[\mathsf{id}_j]}\big)$. However, the original construction

$$\mathsf{btk}'_j \leftarrow \sum_{\ell\in[n/h]} X^{\vec{a}[\ell+j\cdot n/h]} \cdot \mathsf{btk}_{\ell+j\cdot n/h}$$

no longer suffices, since it yields $\mathsf{RLWE}(-X^{\vec{a}[\mathsf{id}_j]})$ instead of $\mathsf{RLWE}(X^{\vec{a}[\mathsf{id}_j]})$ when $\mathsf{sk}_{\mathsf{LWE}}[\mathsf{id}_j] = -1$.

To address this issue, as in [86], we generate two bootstrapping keys for each index $i \in [n]$: $\mathsf{btk}_{i,1}$ and $\mathsf{btk}_{i,-1}$, where $\mathsf{btk}_{i,1} = \mathsf{RLWE}(1)$ if $\mathsf{sk}_{\mathsf{LWE}}[i] = 1$ (and $\mathsf{RLWE}(0)$ otherwise), and $\mathsf{btk}_{i,-1} = \mathsf{RLWE}(1)$ if $\mathsf{sk}_{\mathsf{LWE}}[i] = -1$ (and $\mathsf{RLWE}(0)$ otherwise). We then compute

$$\mathsf{btk}'_j \leftarrow \sum_{\ell\in[n/h]} \big(X^{\vec{a}[\ell+j\cdot n/h]} \cdot \mathsf{btk}_{\ell+j\cdot n/h,1} + X^{-\vec{a}[\ell+j\cdot n/h]} \cdot \mathsf{btk}_{\ell+j\cdot n/h,-1}\big),$$

Algorithm 2 Our New FHEW/TFHE Bootstrapping with Sparse Keys

Given a cononically decodable (n, c, h, h')-PBC scheme PBC with $\mathsf{negl}(\lambda)$ error probability (Definition 5).
procedure $\mathsf{KeyGen}(\mathsf{sk}_{\mathsf{LWE}}, \mathsf{sk}_{\mathsf{RLWE}})$
  $C_0, \ldots, C_{h'-1} \leftarrow \mathsf{PBC.Ecd}([n]; r)$ with randomness r
  $i_0, \ldots, i_{k-1} \leftarrow \mathsf{PBC.Sched}(\mathcal{I})$ where $\mathcal{I} := \{\mathsf{id}_1, \ldots, \mathsf{id}_h\}$ contains all the indices id_j where $\mathsf{sk}_{\mathsf{LWE}}[\mathsf{id}_j] = 1$.
  for $j \in [h']$ **do**
    for $i \in C_j$ **do**
      if $i = \mathsf{id}_j$ **then**
        $\mathsf{btk}_{j,i} \leftarrow \mathsf{RGSW}^{\mathsf{sk}_{\mathsf{RLWE}}}(1)$ ▷ With parameters B, dig, see Section 3.3.1
      else
        $\mathsf{btk}_{j,i} \leftarrow \mathsf{RGSW}^{\mathsf{sk}_{\mathsf{RLWE}}}(0)$
    if $i_j = \perp$ **then**
      $\mathsf{btk}_{j,\mathsf{dummy}} \leftarrow \mathsf{RGSW}^{\mathsf{sk}_{\mathsf{RLWE}}}(1)$ ▷ With parameters B, dig, see Section 3.3.1
    else
      $\mathsf{btk}_{j,\mathsf{dummy}} \leftarrow \mathsf{RGSW}^{\mathsf{sk}_{\mathsf{RLWE}}}(0)$
  $\mathsf{ksk} \leftarrow \mathsf{KeySW}(\mathsf{sk}_{\mathsf{LWE}}, \mathsf{sk}_{\mathsf{RLWE}})$ (where $\mathsf{sk}_{\mathsf{LWE}}$ is embeded into $\mathcal{R}$)
  return $\mathsf{btk} := ((\mathsf{btk}_{j,i})_{i \in C_j}, \mathsf{btk}_{j,\mathsf{dummy}})_{j \in [h']}, \mathsf{ksk}, r$
procedure $\mathsf{BlindRot}((\vec{a}, b)\mathbb{Z}_q^{n+1}, \mathsf{btk}, \mathsf{ksk}, r)$
  $C_0, \ldots, C_{h'-1} \leftarrow \mathsf{PBC.Ecd}([n]; r)$ with randomness r
  $\mathsf{ACC} \leftarrow (0, \frac{Q}{4} \cdot X^{-b}) \in \mathcal{R}_Q^2$
  for $j \in [h']$ **do**
    $\mathsf{btk}'_j \leftarrow (\sum_{i \in \vec{C}_j} X^{\vec{a}[i]} \cdot \mathsf{btk}_{i,j}) + \mathsf{btk}_{j,\mathsf{dummy}}$
    $\mathsf{ACC} \leftarrow \mathsf{ACC} \times \mathsf{btk}'_j$
  return ACC
procedure $\mathsf{Boot}(\mathsf{ct} = \mathsf{LWE}_{\mathsf{msb}}(m) \in \mathbb{Z}_q^{n+1}, \mathsf{btk} := (\mathsf{btk}_i)_{i \in [n]}, \mathsf{ksk})$
  $\mathsf{tarCT} \leftarrow \mathsf{BlindRot}(\mathsf{ct}, \mathsf{btk}, r)$ ▷ The rest is the same as Algorithm 1.
  $P(X) := \sum_{i \in [N]} p_i \cdot X^i$ where $p_i = 1$ if $i \in [-\frac{Q}{8}, \frac{3Q}{8})$ and $p_i = 0$ otherwise.
  $\mathsf{ct}_{\mathsf{res}} \leftarrow \mathsf{tarCT} + Q/8$
  return $\mathsf{ct}' \leftarrow \mathsf{ModSW}(\mathsf{Extract}(\mathsf{KeySW}(\mathsf{ksk}, \mathsf{ct}_{\mathsf{res}}), 0), q) = \mathsf{LWE}_{\mathsf{msb}}(m) \in \mathbb{Z}_q^{n+1}$

which yields the desired btk'_j.

The PBC-based compiler extends to general sparse ternary secrets in a straightforward manner. Compared to the binary case, the number of scalar multiplications and ciphertext additions doubles, while the number of ciphertext multiplications remains unchanged. Since this extension from binary to ternary secrets is conceptually straightforward (and similar ideas have appeared in prior work, e.g., [86]), we omit the formal pseudocode.

4.3 Concrete Optimizations for n Scalar Multiplications

As mentioned, in our new construction, bootstrapping consists of h ciphertext multiplications, n scalar multiplications, and n additions.

At first glance, each plaintext multiplication takes the form $X^{\vec{a}[i]} \cdot \mathsf{btk}_i$ for some i, which appears to be almost free: suppose btk_i is in its coefficient form, the monomial multiplications can be implemented by shifting polynomial coefficients by $\vec{a}[i]$, together with coefficient negation. However, in practice, this intuition can still be costly, for most practical parameter choices (e.g.,

those used in [8, 97]), as it requires $\Omega(n)$ memory copying and negation.

Free monomial multiplications. For simplicity, we discuss the task of a simple monomial by polynomial multiplication, $X^i \cdot p(X)$ for $i \in [N]$ and $p(X) \in \mathcal{R}$, and a monomial by ciphertext multiplication can be achieved by repeating this simpler operation. To avoid the aforementioned overhead, for $p(X) = p_0 + p_1 X + \cdots + p_{N-1} X^{N-1}$, instead of storing a single coefficient vector $\vec{p} = [p_0, \ldots, p_{N-1}]$, we store an extended vector

$$\vec{p}' = [-p_0, \ldots, -p_{N-1},\ p_0, \ldots, p_{N-1}].$$

Then, for any $i \leq N-1$, the product $X^i \cdot p(X)$ can be obtained by directly accessing the contiguous length-N segment from $\vec{p}'[N-i]$ to $\vec{p}'[2N-i-1]$, which corresponds exactly to the shifted polynomial.

Since we require $q \mid 2N$, the exponent i may satisfy $i \geq N$. For $i \in [N, 2N-1]$, we observe that $X^i \cdot p(X) = X^{i-N} \cdot (-p(X))$. Thus, it suffices to also store $-p(X)$, which can be represented as $-\vec{p}'$. Notably, the second half of $\vec{p}'$ and the first half of $-\vec{p}'$ are identical. This allows us to compactly store both using

$$\vec{p}'' = [-p_0, \ldots, -p_{N-1},\ p_0, \ldots, p_{N-1},\ -p_0, \ldots, -p_{N-1}].$$

With this representation, for $i \leq N-1$ the access pattern remains unchanged, and for $i \in [N, 2N-1]$, $X^i \cdot p(X)$ can be obtained by accessing $\vec{p}''[2N-i]$ through $\vec{p}''[3N-i-1]$.

Under this layout, the subsequent addition step following the monomial by ciphertext multiplication can directly read these N consecutive coefficients, eliminating the need for any explicit memory copying. This optimization comes at the cost of a $3\times$ increase in the bootstrapping key size, which remains on the order of megabytes.

Tradeoff between space and runtime. While the technique above enables (nearly) free monomial multiplications, it requires both inputs and outputs to be represented in coefficient form. For the subsequent RGSW-by-RLWE ciphertext multiplications, it is preferable to store $\mathsf{btk}'[j]$ in NTT form for all $j \in [k]$, since otherwise each $\mathsf{btk}'[j]$ must first be converted into NTT form, incurs additional computational costs.

This requirement implies that $X^{\vec{a}[i]} \cdot \mathsf{btk}_i$ must also be produced in NTT form, which in turn requires the input $\mathsf{btk}_{j,i}$ to already be in NTT form. As a result, plaintext multiplication must be carried out directly in the NTT domain, rather than via coefficient shifting and negation. Consequently, a single monomial–ciphertext multiplication costs $\Omega(N)$ operations in $\mathbb{Z}_Q$, instead of being essentially free. Overall, the n plaintext multiplications contribute $\Omega(c \cdot n \cdot N)$ $\mathbb{Z}_Q$ multiplications.

To further improve the runtime efficiency of bootstrapping, we present an approach that eliminates all n plaintext multiplications even when operating in the NTT domain, at the expense of increased bootstrapping key size.

For simplicity, we again consider the case of a structured sparse binary secret key. The first computation step is $X^{\vec{a}[i]} \cdot \mathsf{btk}_i$ for some i. To avoid performing this multiplication online, observe that when $q = 2N$, there are at most $2N$ distinct monomials of the form $X^{\vec{a}[i]}$. Thus, for each btk_i, we can precompute and store $X^\ell \cdot \mathsf{btk}_i$ for all $\ell \in [2N]$. With this precomputation, the n monomial by ciphertext multiplications can be entirely avoided, even in the NTT representation, at the cost of increasing the bootstrapping key size by a factor of $2N$.

This optimization extends directly to general sparse binary secret keys, with the same multiplicative overhead in bootstrapping key size.

5 Low Depth Bootstrapping from Binary NTT Secrets

This section introduces our low-depth bootstrapping construction. To start, we introduce several new variants of the RLWE assumption (Definition 4), which are (somewhat surprisingly) equivalent. Then, we show how to leverage these new variants to build an efficient low-depth bootstrapping. Lastly, we show some initial cryptanalysis including their equivalence, worse-to-average search-to-decision reduction, and attacks in Section 6.

5.1 Our Assumptions

Our first assumption postulates that Ring LWE remains secure when the secret s satisfies $s = s^2$, which means that the NTT form (instead of the coefficient form) of s is a binary vector.

Definition 6 (Binary NTT Distribution)**.** *We define the binary NTT distribution $\mathcal{D}_{N,q}^{\mathsf{Bin\text{-}NTT}}$ with the following sampling algorithm:*

- *Sample a random boolean vector $v \leftarrow_\$ \{0,1\}^N$*
- *Return the ring element $x \in \mathcal{R}_q$ whose NTT representation is v. In other words, it is the unique polynomial satisfying $x(g^{2i+1}) = v_i$ for all $0 \leq i < N$ where g is a primitive $2N$-th root of unity modulo q. To obtain the coefficient representation of x, we multiply v by the inverse NTT matrix for modulus q and degree N.*

Definition 7 (Binary NTT Ring Learning With Error Assumption)**.** *The Binary-NTT RLWE assumption states that no PPT adversary can distinguish between $\{(a_i, a_i s + e_i)\}_{i\in[m]}$ and $\{(a_i, r_i)\}_{i\in[m]}$ where $\{a_i\}_{i\in[m]}, \{r_i\}_{i\in[m]} \leftarrow_\$ \mathcal{R}_q$, $s \leftarrow \mathcal{D}_{N,q}^{\mathsf{Bin\text{-}NTT}}$ and $\{e_i\}_{i\in[m]} \leftarrow \chi$ with noticeable probability as long as the number of samples m is polynomial in λ.*

Next, we consider a more general assumption where the secret satisfies some publicly released quadratic equation $s^2 = us + v$ instead of $s = s^2$ (which is equivalent to setting $u = 1, v = 0$). In this case, we can sample s and u uniformly, and compute v to satisfy the above quadratic.

Definition 8 (Quadratic Hint Ring Learning With Error Assumption)**.** *The QH-RLWE assumption states that no PPT adversary can distinguish between $(u, v, \{a_i\}_{i\in[m]}, \{a_i s + e_i\}_{i\in[m]})$ and $(u, v, \{a_i\}_{i\in[m]}, \{r_i\}_{i\in[m]})$ where $\{a_i\}_{i\in[m]}, \{r_i\}_{i\in[m]}, u, s \leftarrow_\$ \mathcal{R}_q$, $v = s^2 - us$ and $\{e_i\}_{i\in[m]} \leftarrow \chi$ with noticeable probability.*

Finally, we combine this general assumption with the standard small-secret variant of RLWE widely used for constructing FHE. Specifically, we postulate that RLWE is secure even when (1) the secret is small (bounded by some $\beta > 0$ in the coefficient domain) and (2) we additionally publish the quadratic hint as before.

Definition 9 (Small Secret Distribution)**.** *We define $\mathcal{D}_\beta$ with the following sampling algorithm:*

- *Sample each element of vector v of size N from $[-\beta, \beta]$.*
- *Return the ring element*

$$x := \sum_{i=0}^{N-1} v_i X^i$$

.

Definition 10 (Quadratic Hint Small Secret Ring Learning With Error Assumption)**.** *The QH-SS-RLWE assumption states that no PPT adversary can distinguish between $(u, s^2 - us, \{a_i\}_{i\in[m]}, \{a_i s + e_i\}_{i\in[m]})$ and $(u, z^2 - uz, \{a_i\}_{i\in[m]}, \{r_i\}_{i\in[m]})$ where $\{a_i\}_{i\in[m]}, \{r_i\}_{i\in[m]}, u, z \leftarrow_\$ \mathcal{R}_q$, $s \leftarrow \mathcal{D}_\beta$ and $\{e_i\}_{i\in[m]} \leftarrow \chi$ with noticeable probability.*

5.2 Low-depth Gate Bootstrapping

In this section, we discuss how to reduce the circuit depth (i.e., the depth of $\mathbb{Z}_q$ multiplications and additions) of gate bootstrapping using our new assumption. Specifically, for standard FHEW/TFHE bootstrapping [34, 29] (recalled in Algorithm 1), each bootstrapping operation requires $\Omega(n)$ NTT/INTT operations.

This constitutes a major bottleneck even for a powerful server. To illustrate, consider a hypothetical server with unbounded parallel computational resources. Even in this idealized setting, performing a single gate bootstrapping still requires at least $\Omega(n \cdot \log(N))$ sequential steps, where N is the ring dimension, since each NTT or INTT has circuit depth $\Omega(\log(N))$. Moreover, this limitation is not merely theoretical. In practice, NTTs are indeed the dominant bottleneck for hardware acceleration. For GPUs with $N = 2^{12}$ (which is the ring dimension commonly used for gate bootstrapping in practice [8, 97]), GPU-based NTT implementations achieve essentially the same runtime as CPU-based implementations with AVX-512 acceleration, both being approximately 5–10 microseconds [12, 106]. Consequently, to achieve meaningful hardware acceleration, it is crucial to design a gate bootstrapping procedure with low circuit depth. This section presents such a construction, *without* significantly increasing the overall runtime.

Low-depth bootstrapping. We begin by introducing a natural alternative that reduces the bootstrapping depth from n to $\log(n)$ levels of NTT/INTTs. This approach is a simple modification of the original FHEW/TFHE bootstrapping procedure recalled in Algorithm 1. At a high level, we replace RGSW-by-RLWE multiplications with BV-like multiplications (Section 3.3.2) and evaluate the resulting product in a tree-based manner.

More concretely, recall that in FHEW/TFHE, the bootstrapping key is generated as $\mathsf{btk}_i = \mathsf{RGSW}(\mathsf{sk}_{\mathsf{LWE}}[i])$ for $i \in [n]$. Instead, we generate $\mathsf{btk}_i \leftarrow \mathsf{RLWE}_{\mathsf{lsb}}(\mathsf{sk}_{\mathsf{LWE}}[i])$. Subsequently, we compute $\mathsf{btk}'_j \leftarrow (X^{-\vec{a}[j]} - 1) \cdot \mathsf{btk}_j + 1$ for all $j \in [n]$. Now, rather than updating the accumulator sequentially via $\mathsf{ACC} \leftarrow \mathsf{ACC} \times \mathsf{btk}'[j]$, we directly compute

$$\mathsf{ACC} \times \Big(\prod_{j \in [n]} \mathsf{btk}'[j]\Big)$$

using BV-like multiplications between RLWE ciphertexts (as described in Section 3.3.2). Since BV-like multiplications are commutative, the entire product can be evaluated using a binary tree, resulting in only $\log(n)$ levels of NTT/INTT operations. However, this straightforward approach introduces several challenges, which we address next.

Sparse LWE key to reduce depth. First, BV-like multiplications incur significantly higher noise growth than RGSW-by-RLWE multiplications. As a result, they require a substantially larger ciphertext modulus and ring dimension, leading to a considerable increase in runtime. To mitigate this issue, we again leverage our sparse-key technique. By doing so, we reduce the multiplicative depth from $\log(n)$ to $\log(h)$, which substantially lowers the total noise needed and renders the parameters manageable.

Binary NTT RLWE to reduce runtime. Second, even after reducing the depth, the required ciphertext modulus and ring dimension remain larger than those used with RGSW-by-RLWE multiplications. Consequently, the per-operation cost is higher, potentially exceeding that of standard TFHE. We now show how our new assumption alleviates this overhead, by eliminating the need for relinearization and allowing ciphertexts to remain in NTT form throughout the computation, thereby enabling fast multiplications.

In more detail, as recalled in Section 3.3.2, multiplying two RLWE ciphertexts $\mathsf{ct}_1 = (a_1, b_1) = \mathsf{RLWE}_{\mathsf{lsb}}(m_1)$ and $\mathsf{ct}_2 = (a_2, b_2) = \mathsf{RLWE}_{\mathsf{lsb}}(m_2)$ starts by computing

$$(x = a_1 a_2,\ y = a_1 b_2 + a_2 b_1,\ z = b_1 b_2) \in \mathcal{R}_q^3.$$

Recall that $z - xs^2 - ys = m_1 m_2 + t \cdot E$, where E denotes the accumulated error. Now suppose that the secret key satisfies $s = s^2$. Letting $(a' \leftarrow x, b' \leftarrow y + z)$, it is straightforward to verify that $(a', b') = \mathsf{RLWE}_{\mathsf{lsb}}(m_1 \cdot m_2)$, up to increased noise. Crucially, this implies that the relinearization step (which is often the dominant bottleneck in practice) is no longer required.

Moreover, since relinearization is unnecessary, a product chain consists solely of ring multiplications and additions. Specifically, let $\mathsf{ct}_i = (a_i, b_i) = \mathsf{RLWE}_{\mathsf{lsb}}(m_i)$ for $i \in [w]$, and for simplicity assume that w is a power of two. The goal is to compute $\prod_{i\in[w]} \mathsf{ct}_i$. At the first level, we compute

$$\mathsf{ct}_j^{(1)} \leftarrow \mathsf{ct}_{2j} \times \mathsf{ct}_{2j+1}, \text{for } j \in [w/2],$$

where

$$\mathsf{ct}_j^{(1)} := (a_{2j}a_{2j+1} - a_{2j}b_{2j+1} - a_{2j+1}b_{2j},\ b_{2j}b_{2j+1}),$$

which involves only ring multiplications and additions. At the second level, we similarly compute $\mathsf{ct}_k^{(2)} \leftarrow \mathsf{ct}_{2k}^{(1)} \times \mathsf{ct}_{2k+1}^{(1)}$ for $k \in [w/4]$. Continuing in this manner, the final output is $\mathsf{ct}^{(\log(w))} = \mathsf{RLWE}_{\mathsf{lsb}}(\prod_{i\in[w]} m_i)$, as desired. Evidently, the entire process consists exclusively of algebraic ring operations.

As recalled in Section 3.3.2, fast multiplication of two ring elements $x, y \in \mathcal{R}_q$ (assuming $\mathcal{R}$ fully splits over q) normally proceeds in three steps: (1) transforming x and y into NTT form, (2) performing element-wise multiplications in NTT form, and (3) applying an inverse NTT to return to coefficient form. In standard BV-like multiplications, ring elements must remain in coefficient form, since relinearization requires digit decomposition[7]. Under our new assumption, however, relinearization is unnecessary. As a result, the entire product chain can be carried out while keeping all ring elements in NTT form, so that only step (2) is required. This reduces the per-multiplication cost to $O(N)$, whereas NTT and INTT each require $O(N \log N)$ operations for ring dimension N.

One caveat is that BV-like multiplications induce exponential noise growth for $\mathsf{RLWE}_{\mathsf{lsb}}$. Specifically, after w multiplications, the error magnitude satisfies $|E| = \Omega(2^w)$. Therefore, w cannot be too large. To control the noise, after a suitable number of multiplications, we apply modulus switching, which requires ciphertexts to be represented in coefficient form. Moreover, since modulus switching error is proportional to the norm of the secret key, ciphertexts must be encrypted under a secret key that is short in coefficient form. Accordingly, we either require the secret key itself to be short (e.g., by using QH-SS-RLWE as described in Section 5.2), or we perform key switching to re-encrypt ciphertexts under a different secret key sk' that is short, instead of binary in NTT slots, prior to modulus switching. More generally, we parameterize this process by a vector $\vec{w} = (w_0, \ldots, w_{\ell-1})$, where each $w_i \in \mathbb{N}$ specifies the number of multiplications performed before the i-th modulus switching, across ℓ levels of modulus switching. We formalize the resulting construction in Algorithm 3.

Theorem 4. *Let $n, N, q, Q > 0$, for all $\mathsf{sk}_{\mathsf{LWE}} \in \{0,1\}^n$ with hamming weight h, for any $m \in \{0,1\}, p \geq 2$, and $\mathsf{ct} \leftarrow (\vec{a}, b := \langle \vec{a}, \mathsf{sk}_{\mathsf{LWE}} \rangle + m \cdot \lfloor q/p \rceil)$; then, let $\mathcal{R} := \mathbb{Z}[X]/(X^N + 1)$ and for all $\mathsf{sk}_{\mathsf{RLWE}} \in \mathcal{R}_2$, for all $((\mathsf{btk}_i)_{i\in[n]}, \mathsf{ksk}, r)$ in the support of $\mathsf{KeyGen}(\mathsf{sk}_{\mathsf{LWE}}, \mathsf{sk}_{\mathsf{RLWE}})$ $\mathsf{ct}' = (\vec{a}', b') \leftarrow \mathsf{FHEWTFHEBoot}(\mathsf{ct}, (\mathsf{btk}_i)_{i\in[n]}, \mathsf{ksk}, r)$ (both algorithms in Algorithm 3), then let $E \leftarrow b' - \langle \vec{a}', \mathsf{sk}_{\mathsf{LWE}} \rangle - m \cdot \lfloor q/p \rceil \in \mathbb{Z}_q$; it holds that $|E| = O(\frac{q_1}{Q} \cdot (N^2 \cdot p \cdot \sigma_\ell)^{w_\ell} + \sigma_{ms})$ where $\sigma_i = \frac{q_1 \cdots q_{\ell-i+1}}{Q} \cdot (N^2 \cdot p \cdot \sigma_{i-1})^{w_i} + \sigma_{ms}$.*

Proof. The correctness proof is straightforward, similar to Theorem 3. In terms of error analysis, as shown in [57], given two ciphertexts each with error standard deviation σ, each multiplication

[7]There exists an alternative approach to relinearization [16, 36, 57], which likewise requires ring elements to be in coefficient form.

incurs $O(N^2 \cdot p \cdot \sigma)$. Since we only perform a single modulus switching after w_i levels for $w_i \in \vec{x}$, the error accumulates to $O((N^2 \cdot p \cdot \sigma)^{w_i})$. Then after modulus switching is performed, the accumulated error is reduced by a factor of $\frac{q_1 \cdot q_{\ell-i+1}}{Q}$ plus modulus switching error. □

5.3 Beyond Bootstrapping

Lastly, we briefly discuss how our new assumption can be leveraged to accelerate FHE beyond bootstrapping itself. Specifically, it can be used to accelerate regular operations for BGV, BFV, and CKKS even under the "leveled" mode (i.e., no bootstrapping involved).[8]

BGV and CKKS. We start with BGV and CKKS. The multiplications in BGV and CKKS follow the same pattern: two ciphertexts are in NTT form; a tensor product is performed between the two ciphertexts; the resulting ciphertext is transformed into the coefficient form to perform a realization step. After the multiplication, a modulus switching procedure is then performed to control noise. Under our new assumptions, this procedure can be greatly accelerated: in short, the relinearization step can be fully removed.

More specifically, Using either Binary-NTT RLWE or QH-RLWE, ciphertexts can be kept entirely in NTT form, allowing ciphertext multiplications to be performed using only four element-wise multiplications: which are simply $4N$ of $\mathbb{Z}_Q$ multiplications for ring dimension N and ciphertext modulus Q. In contrast, the costly relinearization step costs $O(\mathsf{dig} \cdot N \cdot \log(N))$ of $\mathbb{Z}_Q$ operations for dig being the number of digit decomposition (see Section 3.3.1 for details). However, to apply modulus switching and manage error growth, the resulting ciphertexts need to be key-switched to a secret key whose coefficients are short. The cost of this key-switching step is essentially identical to that of relinearization in standard BGV and CKKS multiplications.

To reduce this overhead, an approach analogous to Algorithm 3 can be employed. Specifically, one may perform w consecutive levels of ciphertext multiplications entirely in NTT form, followed by a single modulus switching operation. This amortizes the cost of key switching across multiple multiplications, at the cost of allowing the noise to grow for w levels.

Alternatively, if QH-SS-RLWE is used, modulus switching can be applied directly. In this case, since relinearization is no longer required, a single ciphertext multiplication is approximately 2-5 times faster than in standard BGV and CKKS (depending on parameters).

BFV. The BFV scheme differs slightly. Although it does not require a separate modulus switching step to manage noise growth, its multiplication procedure implicitly embeds a modulus-switching-like operation. Specifically, during a ciphertext multiplication, BFV computes $x \leftarrow a_1 a_2$, $y \leftarrow a_1 b_2 + a_2 b_1$, and $z \leftarrow b_1 b_2$. Rather than performing these operations over $\mathcal{R}_Q$, they are carried out over $\mathcal{R}$, followed by the computation $\lfloor (x, y, z)/(Q/t) \rceil \in \mathcal{R}$. This rounding step introduces an error analogous to modulus switching, with magnitude proportional to the norm of the secret key coefficients.

Consequently, when using Binary-NTT RLWE for ℓ levels of multiplications, the magnitude of x, y, z may grow to approximately $P \approx Q^{2\ell}$. To enable fast NTT-based multiplication without coefficient wrap-around, the inputs a_1, b_1, a_2, b_2 must therefore be transformed into NTT form over $\mathcal{R}_P$. As a result, the NTT/INTT operations and subsequent tensor products become roughly ℓ times more expensive, while all intermediate multiplications avoid any additional NTT/INTT.

For a computation involving L multiplications, with l_i input ciphertexts and l_o output ciphertexts, this approach yields a savings of approximately $\frac{2L}{(l_i + l_o) \cdot \ell}$.

[8] It can also be used to help BGV, BFV, and CKKS bootstrapping as well, simply because the multiplications are faster, as explained below.

Algorithm 3 Our low-depth bootstrapping

$\mathcal{D}_1$: Binary NTT Secret
$\mathcal{D}_2$: Quadratic Hint Small Secret, which outputs $\mathsf{sk}_{\mathsf{LWE}}$ and a quadratic hint (u, v)
$\mathcal{D}_3$: Ternary Secret
$q = q_1 \cdot q_2 \cdot \ldots \cdot q_\ell$ being the modulus chain
$\vec{w} := (w_0, \ldots, w_{\ell-1})$ where $\prod_i w_i = k$ ▷ WLOG k and w_i's are power-of-twos
procedure $\mathsf{KeyGen}(\mathsf{sk}_{\mathsf{LWE}}, \mathsf{sk}_{\mathsf{RLWE}})$
    $\mathsf{btk}_i \leftarrow \mathsf{RLWE}^{\mathsf{sk}}_{\mathsf{RLWE}}(\mathsf{sk}_{\mathsf{LWE}}[i]) \in \mathbb{Z}_q^{2n}$ for $i \in [n]$, in NTT form.
    if $\mathcal{D}_1$ is the distribution for $\mathsf{sk}_{\mathsf{RLWE}}$ **then** ▷ $\mathsf{sk}_{\mathsf{RLWE}}$ either from $\mathcal{D}_1$ or $\mathcal{D}_2$
        Sample $\mathsf{sk}'_{\mathsf{RLWE}} \leftarrow \mathcal{D}_3$
        $\mathsf{ksk}_1 \leftarrow \mathsf{KeySW}(\mathsf{sk}_{\mathsf{RLWE}}, \mathsf{sk}'_{\mathsf{RLWE}})$
        $\mathsf{ksk}_2 \leftarrow \mathsf{KeySW}(\mathsf{sk}'_{\mathsf{RLWE}}, \mathsf{sk}_{\mathsf{RLWE}})$
    $\mathsf{ksk} \leftarrow \mathsf{KeySW}(\mathsf{sk}_{\mathsf{RLWE}}, \mathsf{sk}_{\mathsf{LWE}})$ (where $\mathsf{sk}_{\mathsf{LWE}}$ is embedded into $\mathcal{R}$)
    Return $(\mathsf{btk}_i)_{i\in[n]}, \mathsf{ksk}, \mathsf{ksk}_1, \mathsf{ksk}_2$
procedure $\mathsf{Mult}(\mathsf{ct}_1, \mathsf{ct}_2 \in \mathbb{Z}_q^{2n})$ ▷ Both in NTT form.
    $x_1, x_2, x_3, x_4 \leftarrow \mathsf{ct}_1 \otimes \mathsf{ct}_2$ ▷ $\otimes$ denotes tensor product
    Return $(x_1 \cdot u - x_2 - x_3, x_4 - x_1 \cdot v) \in \mathbb{Z}_q^{2n}$
        ▷ If $\mathsf{sk}_{\mathsf{LWE}} \in \mathcal{D}_2$, (u, v) is the quadratic hint. If $\mathsf{sk}_{\mathsf{LWE}} \in \mathcal{D}_1$, $u = 1, v = 0$.
procedure $\mathsf{AltBoot}(\mathsf{ct} = \mathsf{LWE}_{\mathsf{msb}}(m) \in \mathbb{Z}_q^{n+1}, (\mathsf{btk}_i)_{i\in[n]}, \mathsf{ksk})$
    $P(X) := \sum_{i\in[N]} p_i \cdot X^i$ where $p_i = 1$ if $i \in [-\frac{Q}{8}, \frac{3Q}{8})$ and $p_i = 0$ otherwise.
    $\mathsf{ACC} \leftarrow (0, X^{-b} \cdot P(X)) \in \mathcal{R}_Q^2$
    $\vec{v}_0, \ldots, \vec{v}_{k-1} \leftarrow \mathsf{PBC.Ecd}([n]; r)$ with randomness r
    for $j \in [k]$ **do**
        $\mathsf{btk}_j^{(0)}(\sum_{i\in\vec{v}_j} X^{-\vec{a}[i]} \cdot \mathsf{btk}_{i,j}) + \mathsf{btk}_{j,\mathsf{dummy}}$
    $k^0 \leftarrow k$
    for $w_l \in \vec{w}(i.e., l \in [\ell])$ **do**
        for $j \in [k^l/w_l]$ **do**
            $\mathsf{btk}_j^{(l+1)} \leftarrow \prod_{i\in[j\cdot(k^l/w_l),(j+1)\cdot(k^l/w_l))} \mathsf{btk}_i^l$
                ▷ Products use Mult between two ciphertexts defined above.
        $\mathsf{INTT}(\mathsf{btk}_j^{l+1})$
        if $\mathcal{D}_1$ is the distribution for $\mathsf{sk}_{\mathsf{RLWE}}$ **then**
            $\mathsf{KeySW}(\mathsf{ksk}_1, \mathsf{btk}_j^{l+1})$
        $\mathsf{ModSW}(\mathsf{btk}_j^{l+1})$ ▷ From $q_1 \cdot \ldots \cdot q_l$ to $q_1 \cdot \ldots \cdot q_{l-1}$
        if $\mathcal{D}_1$ is the distribution for $\mathsf{sk}_{\mathsf{RLWE}}$ **then**
            $\mathsf{KeySW}(\mathsf{ksk}_2, \mathsf{btk}_j^{l+1})$
    return $\mathsf{ct}' \leftarrow \mathsf{ModSW}(\mathsf{Extract}(\mathsf{KeySW}(\mathsf{ksk}, \mathsf{ACC}), 0), q) = \mathsf{LWE}_{\mathsf{msb}}(m) \in \mathbb{Z}_q^{n+1}$

Alternatively, we can use QH-SS-RLWE instead to simply keep everything else the same, except for removing the need of relinearization. In this setting, the primary performance gain comes from eliminating relinearization altogether, which accounts for approximately 25% of the total runtime in mainstream implementations (e.g., [8, 96]).

A more detailed study on how our new assumption may help other FHE schemes, including but not limited to bootstrapping, is left for future research.

6 Understanding our New Assumptions

Since our low-depth bootstrapping scheme relies on new assumptions, we will now take a deeper look at them.

6.1 Reductions Among our Assumptions

In this subsection, we will show that all three assumptions we introduced are in fact equivalent to each other.

6.1.1 Binary-NTT RLWE implies QH-RLWE

We will reduce an instance of QH-RLWE to one of Binary-NTT RLWE. The basic idea is to apply a random linear transformation to the secret s. This will immediately result in the new secret s' obeying a random quadratic equation known to us. We then adjust (a, b) into a new pair such that if $b = as + e$, we have $b' = a's' + e$.

Theorem 5. *Let N, m, q, χ be parameters dependent on λ such that $2N \mid q-1$, q is a prime and $N \geq \lambda$. Assume that there is an adversary $\mathcal{A}$ running in time T that can distinguish between $(\{a_i\}_{i\in[m]}, u, v, \{a_i s + e_i\}_{i\in[m]})$ and $(\{a_i\}_{i\in[m]}, u, v, \{r_i\}_{i\in[m]})$ where $\{a_i\}_{i\in[m]}, \{r_i\}_{i\in[m]}, u, s \leftarrow_\$ \mathcal{R}_q$, $v = s^2 - us$ and $\{e_i\}_{i\in[m]} \leftarrow \chi$ with probability $1/2+\epsilon$. Then we can construct an adversary $\mathcal{B}$ running in time $T + O(Nm)$ who can distinguish between $(\{a_i\}_{i\in[m]}, \{a_i s + e_i\}_{i\in[m]})$ and $(\{a_i\}_{i\in[m]}, \{r_i\}_{i\in[m]})$ where $\{a_i\}_{i\in[m]}, \{r_i\}_{i\in[m]} \leftarrow_\$ \mathcal{R}_q$, $s \leftarrow \mathcal{D}^{\mathsf{Bin\text{-}NTT}}$ and $\{e_i\}_{i\in[m]} \leftarrow \chi$ with probability at least $1/2 + \epsilon$.*

Proof. To prove this theorem, we first give the algorithm for $\mathcal{B}$ in Fig. 1.

Then, we prove a few properties of $\mathcal{B}$.

Lemma 6. *If $\vec{b}$ was sampled uniformly from $\mathcal{R}_q$, the distribution of the tuple $(\vec{a}', u', v', \vec{b}')$ is identical to $\vec{a}', u', \vec{b}', s' \leftarrow_\$ \mathcal{R}_q$ and $v' = s'^2 - u's'$.*

Proof. We have

$$\Pr[a'_i = x] = \Pr[a_i u^{-1} = x] = \Pr[a_i = xu]$$

Since only a'_i depends on a_i, and a_i is sampled uniformly from $\mathcal{R}_q$, the above equations imply that each a'_i is both uniformly random and independent from the other elements of the tuple.

Similarly,

$$\Pr[b'_i = x] = \Pr[b_i + a'_i v = x] = \Pr[b_i = x - a'_i v]$$

Since only b'_i depends on b_i, and b_i is sampled uniformly from $\mathcal{R}_q$, the above equations imply that each b'_i is both uniformly random and independent from the other elements of the tuple.

Algorithm $\mathcal{B}$

Input:

- A PPT algorithm $\mathcal{A}$ for solving QH-RLWE(N, m, q, χ)
- m ring elements $\{a_i\}_{i\in[m]} \leftarrow_\$ \mathcal{R}_q$
- Another m ring elements $\{b_i\}_{i\in[m]}$ which are either sampled uniformly from $\mathcal{R}_q$, or by evaluating $b_i = a_i s + e_i$ where $s \leftarrow_\$ \mathcal{D}_{N,q}^{\text{Bin-NTT}}$, $\{e_i\}_{i\in[m]} \leftarrow_\$ \chi$

Objective: Determine which of the above distributions $\vec{b}$ is sampled from.
Algorithm:

1. Let u be a random invertible ring element. One way of sampling u is choosing a vector $\mathbf{u'} \leftarrow_\$ (\mathbb{Z}_q^*)^N$ and applying the inverse NTT transform.
2. Let $v \leftarrow_\$ \mathcal{R}_q$.
3. For all $i \in [m]$:
   (a) Let $a'_i \leftarrow a_i/u$.
   (b) Let $b'_i \leftarrow b_i + a'_i v$.
4. Let $u' \leftarrow u + 2v$.
5. Let $v' \leftarrow -v^2 - uv$.
6. Return $\mathcal{A}(\vec{a}', u', v', \vec{b}')$.

Figure 1: $\mathcal{B}$ for Theorem 5.

We have to reason about the distribution of the remaining two variables together. First, we calculate the joint distribution when these variables are sampled by $\mathcal{B}$.

$$\begin{aligned}
\Pr[u' = x, v' = y] &= \Pr[u + 2v = x, -v^2 - uv = y] \\
&= \Pr[u = x - 2v, y + v^2 + uv = 0] \\
&= \Pr[u = x - 2v, y + v^2 + (x - 2v)v = 0] \\
&= \Pr[u = x - 2v, v^2 - xv - y = 0] \\
&= \Pr\left[u = x - 2v, v = \frac{x \pm \sqrt{x^2 + 4y}}{2}\right] \\
&= \frac{1}{|\mathcal{R}_q|} \Pr\left[v = \frac{x \pm \sqrt{x^2 + 4y}}{2}\right]
\end{aligned}$$

Now, we calculate the joint distribution when they are sampled as $u', s' \leftarrow_{\$} \mathcal{R}_q$ and $v' = s'^2 - u's'$.

$$\begin{aligned}
\Pr[u' = x, v' = y] &= \Pr[u' = x, s'^2 - u's' = y] \\
&= \Pr[u' = x, s'^2 - xs' - y = 0] \\
&= \Pr\left[u' = x, s' = \frac{x \pm \sqrt{x^2 + 4y}}{2}\right] \\
&= \frac{1}{|\mathcal{R}_q|} \Pr\left[s' = \frac{x \pm \sqrt{x^2 + 4y}}{2}\right]
\end{aligned}$$

Since both v and s' are i.i.d, the above probabilities are equal. □

Lemma 7. *If $\vec{b}$ was chosen as $\vec{a} \cdot s + \vec{e}$, then $\vec{b'} = \vec{a'} \cdot s' + \vec{e}$ for some s' satisfying $s'^2 = u's' + v'$.*

Proof. This follows from simple algebra. Observe that:

$$\begin{aligned}
\vec{b'} &= \vec{b} + \vec{a'} \cdot v && \text{From Line 3b} \\
&= \vec{a} \cdot s + \vec{e} + \vec{a'} \cdot v && \text{By assumption} \\
&= \vec{a'} \cdot us + \vec{e} + \vec{a'} \cdot v && \text{From Line 3a} \\
&= \vec{a'} \cdot (us + v) + \vec{e}
\end{aligned}$$

We set $s' = us + v$. It is easy to verify that it satisfies the required quadratic equation.

$$\begin{aligned}
s'^2 &= (us + v)^2 \\
&= u^2 s^2 + 2uvs + v^2 \\
&= (u^2 + 2uv)s + v^2 && \text{Since } s^2 = s \\
&= (u + 2v)us + uv + 2v^2 - uv - v^2 \\
&= (u + 2v)(us + v) + (-v^2 - uv) \\
&= u's' + v' && \text{By definition of } u', s' \text{ and } v'
\end{aligned}$$

□

Lemma 8. *If $\vec{b}$ was chosen as $\vec{a} \cdot s + \vec{e}$, the distribution of the tuple $(\vec{a'}, u', v', \vec{b'})$ is identical to $\vec{a'}, u', s' \leftarrow_{\$} \mathcal{R}_q$, $v' = s'^2 - u's'$ and $\vec{b'} = \vec{a'} \cdot s' + \vec{e}$.*

Proof. Define $s' = us + v$. We know from Theorem 7 that $v' = s'^2 - u's'$ and $\vec{b}' = \vec{a}' \cdot s' + \vec{e}$. So all we need to show is that $\vec{a}', u', s'$ are i.i.d. variables sampled uniformly from $\mathcal{R}_q$.

This is, however, fairly easy to see directly.

$$\begin{aligned}\Pr[\vec{a}' = \vec{x}, u' = y, s' = z] &= \Pr[\vec{a} \cdot u^{-1} = \vec{x}, u + 2v = y, us + v = z] \\ &= \Pr[\vec{a} = \vec{x} \cdot u, v = (y-u)/2, s = (z-v)u^{-1}] \\ &= \left(\frac{1}{|\mathcal{R}_q|}\right)^{m+2}\end{aligned}$$

□

Proof of Theorem 5. The previous lemmas establish that corresponding to the two possible ways of sampling $\vec{b}$, we call $\mathcal{A}$ with the expected distributions for the two cases of the QH-RLWE problem. Therefore, the success probability of $\mathcal{B}$ is exactly the same as the success probability of $\mathcal{A}$. Since $\mathcal{B}$ only samples a few ring elements and does $O(m)$ ring operations before calling $\mathcal{A}$, it also has the desired time complexity. □

Remark 9. Note that for this reduction and all the following reductions, we assume that the modulus q is a prime in the rest of this section. However, in practice, a large composite modulus is often used. Note that the only place we invoke the primality of q is to establish that a randomly chosen element of $\mathcal{R}_q$ is invertible with at least constant probability (both above and below). This is still the case when q is a product of x primes, each of magnitude $\Omega(xN)$ (since any of the xN elements in the NTT domain is 0 with probability $\Omega(1/xN)$ and we can use the union bound to bound the overall probability), and our reductions work as well in the same way.

6.1.2 QH-RLWE implies Binary-NTT RLWE

We can also reduce an instance of QH-RLWE to one of Binary-NTT RLWE. The basic idea is the same as before. Given the quadratic hint, we apply an appropriate linear transformation to the secret s' so that the new secret satisfies $s'^2 = s'$. We then adjust (a, b) into a new pair such that if $b = as + e$, we have $b' = a's' + e$. This time, however, there is a chance that the new secret does not follow the Bin-NTT distribution. We show that as long as we start with a "good" quadratic hint (u, v) (whose probability is at least $1/4$), our transformation works. As a result, we can still prove an analogous theorem as before, only with a constant fraction of loss in the success probability.

Theorem 10. *Let N, m, q, χ be parameters dependent on λ such that $2N \mid q-1$ and $N \geq \lambda$. Assume that there is an adversary $\mathcal{A}$ running in time T that can distinguish between $(\{a_i\}_{i\in[m]}, \{a_i s + e_i\}_{i\in[m]})$ and $(\{a_i\}_{i\in[m]}, \{r_i\}_{i\in[m]})$ where $\{a_i\}_{i\in[m]}, \{r_i\}_{i\in[m]} \leftarrow_\$ \mathcal{R}_q$, $s \leftarrow \mathcal{D}_{N,q}^{\mathsf{Bin\text{-}NTT}}$ and $\{e_i\}_{i\in[m]} \leftarrow \chi$ with probability $1/2+\epsilon$. Then we can construct an adversary $\mathcal{B}$ running in time $T + O(Nm + N \log Nq)$ who can distinguish between $(\{a_i\}_{i\in[m]}, u, v, \{a_i s + e_i\}_{i\in[m]})$ and $(\{a_i\}_{i\in[m]}, u, v, \{r_i\}_{i\in[m]})$ where $\{a_i\}_{i\in[m]}, \{r_i\}_{i\in[m]}, u, s \leftarrow_\$ \mathcal{R}_q$, $v = s^2 - us$ and $\{e_i\}_{i\in[m]} \leftarrow \chi$ with probability at least $1/2 + \epsilon/4$.*

Proof. This proof follows the previous one quite closely. We will rigorously cover the differences and then skip over some of the repetitive details. As before, we first construct $\mathcal{B}$ as shown in Fig. 2 and prove a few properties of $\mathcal{B}$.

Lemma 11. *Square root(s) of $4v + u^2$ exist in $\mathcal{R}_q$.*

Algorithm $\mathcal{B}$

Input:

- A PPT algorithm $\mathcal{A}$ for solving Bin-NTT RLWE(N, m, q, χ)
- A tuple of ring elements $(\vec{a}, u, v, \vec{b})$ which are sampled as $(\vec{a}, u, \vec{r}, s \leftarrow_{\$} \mathcal{R}_q)$, $v = s^2 - us$, $\vec{e} \leftarrow \chi$ and $\vec{b}$ is either set to $\vec{r}$, or to $\vec{b} = \vec{a} \cdot s + \vec{e}$.

Objective: Determine which of the above distributions b is sampled from.
Algorithm:

1. Let α be a randomly sampled square root of $4v + u^2$ in $\mathcal{R}_q$.
2. If α is not invertible:
   (a) Return a random guess.
3. Let $\beta \leftarrow (\alpha - u)/2$.
4. For all $i \in [m]$:
   (a) Let $a'_i \leftarrow a_i \alpha$.
   (b) Let $b'_i \leftarrow b_i + a_i \beta$.
5. Let $a' \leftarrow a\alpha$.
6. Let $b' \leftarrow b + a\beta$.
7. Return $\mathcal{A}(\vec{a}', \vec{b}')$.

Figure 2: $\mathcal{B}$ for Theorem 10.

Proof. This follows from simple algebra. Observe that:

$$\begin{aligned}(2s-u)^2 &= 4s^2 - 4us + u^2 \\ &= 4v + u^2 && \text{By definition of } v\end{aligned}$$

□

Lemma 12. $\Pr[\alpha \textit{ is invertible}] > 1/4$

Proof. This fact is easier to see in the NTT domain. Let us denote the NTT representation of α and $2s-u$ by $\mathbf{z}$ and $\mathbf{t}$ respectively. Then, we have from Theorem 11 that

$$\forall i \in [N] \quad (\mathbf{z}[i])^2 = (\mathbf{t}[i])^2 \implies \mathbf{z}[i] = \pm\mathbf{t}[i]$$

α is invertible as long as $\mathbf{z}$ does not contain any zeroes which is true if and only if $\mathbf{t}$ has no 0 entries. However, since $u \leftarrow_{\$} \mathcal{R}_q$, so is $2s-u$, and the NTT transform of a random element of $\mathcal{R}_q$ is a random vector in $\mathbb{Z}_q^N$. Therefore, our required probability is

$$\begin{aligned}\left(\frac{q-1}{q}\right)^N &= \left(1-\frac{1}{q}\right)^N \\ &> \left(1-\frac{1}{N}\right)^N && \text{Since } 2N|q-1 \implies q > N \\ &\geq 1/4\end{aligned}$$

□

Lemma 13. *Consider the case where the last input to $\mathcal{B}$ was chosen as $\vec{b} = \vec{r}$. In case we call $\mathcal{A}$, its input $(\vec{a}', \vec{b}')$ is distributed identically to $2m$ i.i.d. random variables sampled uniformly from $\mathcal{R}_q$.*

Proof. This follows directly from the calculation below:

$$\begin{aligned}\Pr[\vec{a}' = \vec{x}, \vec{b}' = \vec{y}] &= \Pr[\vec{a}\cdot\alpha = \vec{x}, \vec{b} + \vec{a}\cdot\beta = \vec{y}] \\ &= \Pr[\vec{a} = \vec{x}\cdot\alpha^{-1}, \vec{b} = \vec{y} - \vec{a}\cdot\beta] && \text{(Since } \alpha \text{ is invertible after Line 2)} \\ &= \left(\frac{1}{|\mathcal{R}_q|}\right)^{2m}\end{aligned}$$

□

Lemma 14. *Consider the case where the last input to $\mathcal{B}$ was chosen as $\vec{b} = \vec{a}\cdot s + \vec{e}$. In case we call $\mathcal{A}$, the distribution of its input $(\vec{a}', \vec{b}')$ is identical to $\vec{a}' \leftarrow_{\$} \mathcal{R}_q$, $s' \leftarrow \mathcal{D}_{N,q}^{\textit{Bin-NTT}}$ and $\vec{b}' = \vec{a}'\cdot s' + \vec{e}$.*

Proof. Since we call $\mathcal{A}$, we can conclude α^{-1} exists.

Define $s' = (s+\beta)/\alpha$. Observe that

$$\begin{aligned}\vec{b}' &= \vec{b} + \vec{a}\cdot\beta \\ &= \vec{a}\cdot s + \vec{e} + \vec{a}\cdot\beta \\ &= (\vec{a}\cdot\alpha)(\frac{s+\beta}{\alpha}) + \vec{e} \\ &= \vec{a}'\cdot s' + \vec{e}\end{aligned}$$

Just as before, $\vec{a}'$ is distributed uniformly over $\mathcal{R}_q$ and is independent of s'. So the only thing left to show is $s' \leftarrow \mathcal{D}_{N,q}^{\text{Bin-NTT}}$.

It is easy to verify that $s'^2 = s$:

$$\begin{aligned}
s'^2 &= \left(\frac{s+\beta}{\alpha}\right)^2 \\
&= \left(\frac{s+(\alpha-u)/2}{\alpha}\right)^2 \\
&= \frac{1}{4}\left(\frac{2s-u}{\alpha}+1\right)^2 \\
&= \frac{1}{4}\left(\frac{4s^2-4su+u^2}{\alpha^2}+\frac{4s-2u}{\alpha}+1\right) \\
&= \frac{1}{4}\left(\frac{4v+u^2}{\alpha^2}+\frac{4s-2u}{\alpha}+1\right) \\
&= \frac{1}{4}\left(1+\frac{4s-2u}{\alpha}+1\right) \\
&= \frac{2\alpha+4s-2u}{4\alpha} \\
&= \frac{s+(\alpha-u)/2}{\alpha} \\
&= \frac{s+\beta}{\alpha} \\
&= s'
\end{aligned}$$

Observe that there are 2^N possible choices for α (this follows from the proof of Theorem 11). Each choice of α leads to a distinct value for s'. Also, there are exactly 2^N choices for s' that satisfies $s'^2 = s'$. These imply that choosing a random square root in Line 1 is enough to guarantee our s' is a random ring element satisfying $s'^2 = s'$, and therefore follows $\mathcal{D}_{N,q}^{\text{Bin-NTT}}$. □

Proof of Theorem 10. The previous lemmas establish that corresponding to the two possible ways of sampling $\vec{b}$, we call $\mathcal{A}$ with the expected distributions for the two cases of the QH-RLWE problem. Therefore, if we end up calling $\mathcal{A}$, the success probability of $\mathcal{B}$ is exactly the same as the success probability of $\mathcal{A}$. We call $\mathcal{A}$ whenever α is invertible, which happens with probability more than $1/4$. If α is not invertible, our success probability is exactly $1/2$ since we guess randomly. The overall success probability is therefore at least

$$(1/4)\Pr[\mathcal{A} \text{ succeeds}] + (3/4)(1/2) \geq (1/4)(1/2+\epsilon) + (3/4)(1/2) = 1/2 + \epsilon/4$$

Observe that $\mathcal{B}$ only samples a random square root and does $O(m)$ ring operations before calling $\mathcal{A}$. To sample a square root, one can perform the NTT transform, take square roots of all N coordinates in $\mathbb{Z}_p$, and then perform the INTT transform. This can be done in $O(N \log N + N \log q)$ time. Therefore $\mathcal{B}$ has the desired time complexity. □

6.1.3 QH-RLWE implies QH-SS-RLWE

This is intuitively a more surprising result than the previous two, since QH-SS-RLWE imposes a major additional constraint on the secret of QH-RLWE. We are going to show that this additional constraint does not make it any easier for an adversary to distinguish noisy linear

Algorithm $\mathcal{B}$

Input:

- A PPT algorithm $\mathcal{A}$ for solving QH-SS-RLWE(N, m, q, χ, χ)
- $2m + 4$ ring elements $(u, v, \{a_i\}_{i\in[m+1]}, \{b_i\}_{i\in[m+1]})$ which are sampled as $(\{a_i\}_{i\in[m+1]}, u, \{r_i\}_{i\in[m+1]}, s \leftarrow_\$ \mathcal{R}_q)$, $v = s^2 - us$, $\{e_i\}_{i\in[m+1]} \leftarrow \chi$ and $\{b_i\}_{i\in[m+1]}$ is either set to $\{r_i\}_{i\in[m+1]}$, or to $\{a_i s + e_i\}_{i\in[m+1]}$.

Objective: Determine which of the above distributions $\vec{b}$ is sampled from.
Algorithm:

1. If a_{m+1} is not invertible:
   (a) Return a random guess.
2. For all $i \in [m]$:
   (a) Let $a'_i \leftarrow -a_i/a_{m+1}$.
   (b) Let $b'_i \leftarrow b_i - (b_{m+1}/a_{m+1})a_i$.
3. Let $u' \leftarrow 2b_{m+1} - a_{m+1}u$.
4. Let $v' \leftarrow a_{m+1}b_{m+1}u + a^2_{m+1}v - b^2_{m+1}$.
5. Return $\mathcal{A}(u', v', \{a'_i\}_{i\in[m]}, \{b'_i\}_{i\in[m]})$.

Figure 3: $\mathcal{B}$ for Theorem 15.

samples from random elements, as long as the error distribution χ is equal to the small-secret distribution.

We follow the blueprint set in [6] to prove this. We can convert $m+1$ samples of QH-RLWE into m samples of QH-SS-RLWE such that the error term of the $m+1^{th}$ original sample becomes the new secret. This new secret is small by construction (same as the error distribution). In order to generate the quadratic hint, we just use the quadratic hint for the original secret and the linear equation $b_{m+1} = a_{m+1}s + e_{m+1}$.

Theorem 15. *Let N, q, χ be parameters dependent on λ such that $2N \mid q - 1$ and $N \geq \lambda$. Assume that there is an adversary $\mathcal{A}$ running in time T that can distinguish between the ensembles $(u, s^2 - us, \{a_i\}_{i\in[m]}, \{a_i s + e_i\}_{i\in[m]})$ and $(u, z^2 - uz, \{a_i\}_{i\in[m]}, \{r_i\}_{i\in[m]})$ where $\{a_i\}_{i\in[m]}, \{r_i\}_{i\in[m]}, u, z \leftarrow_\$ \mathcal{R}_q$ and $s, \{e_i\}_{i\in[m]} \leftarrow \chi$ with probability $1/2 + \epsilon$. Then we can construct an adversary $\mathcal{B}$ running in time $T + O(N)$ who can distinguish between the ensembles $(u, v, \{a_i\}_{i\in[m+1]}, \{a_i s + e_i\}_{i\in[m+1]})$ and $(u, v, \{a_i\}_{i\in[m+1]}, \{r_i\}_{i\in[m+1]})$ where $\{a_i\}_{i\in[m+1]}$, $\{r_i\}_{i\in[m+1]}, u, s \leftarrow_\$ \mathcal{R}_q$, $v = s^2 - us$ and $\{e_i\}_{i\in[m+1]} \leftarrow \chi$ with probability at least $1/2 + \epsilon/4$.*

Proof. We now construct $\mathcal{B}$ as in Fig. 3 and prove a few properties of $\mathcal{B}$.

Lemma 16. $\Pr[a_{m+1}$ *is invertible*$] > 1/4$

Proof. Since a_{m+1} is a randomly chosen ring element, its NTT representation is also a random vector in $\mathbb{Z}_q^N$. The element is invertible if and only if the NTT representation does not contain

any zeroes, which happens with probability

$$\begin{aligned}\left(\frac{q-1}{q}\right)^N &= \left(1-\frac{1}{q}\right)^N \\ &> \left(1-\frac{1}{N}\right)^N && \text{Since } 2N|q-1 \implies q > N \\ &\geq 1/4\end{aligned}$$

□

Lemma 17. *If $\vec{b}$ was sampled as $b_i = a_i s + e_i$, then we have:*

1. $b'_i = a'_i e_{m+1} + e_i$
2. $e^2_{m+1} = u' e_{m+1} + v'$

Proof. Both assertions follow from simple algebra. For the first part, observe that

$$\begin{aligned}b'_i &= b_i - (b_{m+1}/a_{m+1})a_i \\ &= (a_i s + e_i) - \frac{a_{m+1}s + e_{m+1}}{a_{m+1}} \cdot a_i \\ &= e_i - (e_{m+1}/a_{m+1})a_i \\ &= (-a_i/a_{m+1})e_{m+1} + e_i \\ &= a'_i e_{m+1} + e_i\end{aligned}$$

For the second part, observe that

$$\begin{aligned}u'e_{m+1} + v' &= (2b_{m+1} - a_{m+1}u)e_{m+1} + v' && \text{(By definition of } u') \\ &= (2b_{m+1} - a_{m+1}u)(b_{m+1} - sa_{m+1}) + v' && \text{(Since } b_i = sa_i + e_i) \\ &= 2b^2_{m+1} - a_{m+1}b_{m+1}u - 2a_{m+1}b_{m+1}s + a^2_{m+1}us + v' \\ &= 2b^2_{m+1} - a_{m+1}b_{m+1}u - 2a_{m+1}b_{m+1}s + a^2_{m+1}(s^2 - v) + v' && \text{(Since } s^2 = us + v) \\ &= (b^2_{m+1} - 2a_{m+1}b_{m+1}s + a^2_{m+1}s^2) + b^2_{m+1} - a_{m+1}b_{m+1}u - a^2_{m+1}v + v' \\ &= (b_{m+1} - a_{m+1}s)^2 + (b^2_{m+1} - a_{m+1}b_{m+1}u - a^2_{m+1}v) + v' \\ &= (b_{m+1} - a_{m+1}s)^2 && \text{(By definition of } v') \\ &= e^2_{m+1} && \text{(Since } b_i = sa_i + e_i)\end{aligned}$$

□

Lemma 18. *In case we call $\mathcal{A}$, its inputs follow one of the two distributions expected in the QH-SS-RLWE game depending on whether $\vec{b}$ was set to $\vec{r}$ or $\vec{a} \cdot s + \vec{e}$.*

Proof. Since we reach line 5 in the code of $\mathcal{B}$ to call $\mathcal{A}$, we can assume that a_{m+1} is invertible.

We first consider the case $\vec{b} = \vec{a} \cdot s + \vec{e}$. The a'_i are all independent and uniformly random since so are the a_i and $a'_i = -a_i/a_{m+1}$. Since $u \leftarrow_\$ \mathcal{R}_q$ so is ua_{m+1} and therefore so is $2b_{m+1} - a_{m+1}u = u'$. The new secret s' is e_{m+1}, which is sampled from χ. We know from Theorem 17 that $v' = s'^2 - u's'$ and $\vec{b'} = \vec{a'} \cdot s' + \vec{e}$. That concludes the proof for this case.

Now consider the case $\vec{b} = \vec{r}$. Just as before, u' and $\vec{a'}$ are independent and uniformly random because so are u and $\vec{a}$. Each $b'_i = b_i - (b_{m+1}/a_{m+1})a_i$ is independent of all previous

variables and uniformly random since so is $b_i = r_i$. Observe that

$$\begin{aligned}
v' &= a_{m+1}b_{m+1}u + a_{m+1}^2 v - b_{m+1}^2 \\
&= a_{m+1}b_{m+1}u + a_{m+1}^2(s^2 - us) - b_{m+1}^2 \\
&= a_{m+1}b_{m+1}u + a_{m+1}^2 s^2 - a_{m+1}^2 us - b_{m+1}^2 \\
&= a_{m+1}u(b_{m+1} - a_{m+1}s) + (a_{m+1}s + b_{m+1})(a_{m+1}s - b_{m+1}) \\
&= (b_{m+1} - a_{m+1}s)(a_{m+1}u - a_{m+1}s - b_{m+1}) \\
&= (b_{m+1} - a_{m+1}s)(2b_{m+1} - u' - a_{m+1}s - b_{m+1}) \\
&= (b_{m+1} - a_{m+1}s)(b_{m+1} - a_{m+1}s - u') \\
&= z(z - u') = z^2 - u'z, \text{ where } z := b_{m+1} - a_{m+1}s
\end{aligned}$$

Here z is uniform in $\mathcal{R}_q$ since so is b_{m+1}. That concludes the proof for this second case. □

Proof of Theorem 15. The previous lemma establishes that corresponding to the two possible ways of sampling b, we call $\mathcal{A}$ with the expected distributions for the two cases of the QH-SS-RLWE problem. Therefore, if we end up calling $\mathcal{A}$, the success probability of $\mathcal{B}$ is exactly the same as the success probability of $\mathcal{A}$. We call $\mathcal{A}$ whenever a_{m+1} is invertible, which happens with probability more than $1/4$. If a_{m+1} is not invertible, our success probability is exactly $1/2$ since we guess randomly. The overall success probability is therefore at least

$$(1/4)\Pr[\mathcal{A} \text{ succeeds}] + (3/4)(1/2) \geq (1/4)(1/2 + \epsilon) + (3/4)(1/2) = 1/2 + \epsilon/4$$

Observe that $\mathcal{B}$ only does some ring operations before calling $\mathcal{A}$. Therefore $\mathcal{B}$ has the desired time complexity. □

6.1.4 QH-SS-RLWE implies QH-RLWE

If we are given an oracle that can solve QH-RLWE, it is quite simple to use this to solve an instance of QH-SS-RLWE. An instance of the latter problem only differs from the former in the secret distribution. Therefore, all we need to do is to randomize the secret.

We can transform the secret from s to $s+s'$ by changing the samples from (a, b) to $(a, b+as')$. The quadratic hint for the new secret can also be derived via some simple algebra:

$$s^2 = us+v \implies (s+s')^2 = s^2+2ss'+s'^2 = (u+2s')s+(s'^2+v) = (u+2s')(s+s')+(v-us'-s'^2)$$

Theorem 19. *Let N, q, χ be parameters dependent on λ such that $2N \mid q-1$ and $N \geq \lambda$. Assume that there is an adversary $\mathcal{A}$ running in time T that can distinguish between the ensembles $(u, v, \{a_i\}_{i\in[m+1]}, \{a_i s+e_i\}_{i\in[m+1]})$ and $(u, v, \{a_i\}_{i\in[m+1]}, \{r_i\}_{i\in[m+1]})$ where $\{a_i\}_{i\in[m+1]}$, $\{r_i\}_{i\in[m+1]}, u, s \leftarrow_\$ \mathcal{R}_q$, $v = s^2 - us$ and $\{e_i\}_{i\in[m+1]} \leftarrow \chi$ with probability $1/2 + \epsilon$. Then we can construct an adversary running in time $T + O(N)$ who can distinguish between the ensembles $(u, s^2 - us, \{a_i\}_{i\in[m]}, \{a_i s + e_i\}_{i\in[m]})$ and $(u, z^2 - uz, \{a_i\}_{i\in[m]}, \{r_i\}_{i\in[m]})$ where $s, \{e_i\}_{i\in[m]} \leftarrow \chi$ and $\{a_i\}_{i\in[m]}, \{r_i\}_{i\in[m]}, u, z \leftarrow_\$ \mathcal{R}_q$ with probability at least $1/2 + \epsilon/4$.*

We omit a detailed proof since the techniques are almost identical to the proof of Theorem 5.

6.1.5 Search to Decision Reduction

Now that we have shown all 3 new assumptions to be equivalent, we will focus on Binary-NTT RLWE for the rest of this section. So far, we have been analyzing the decision version of this assumption. Now, we will show that the search problem can be reduced to the decision problem.

Theorem 20. *Let N, q, χ be parameters dependent on λ such that N is a power of 2, $2N \mid q-1$ and $N \geq \lambda$. Assume that there is an adversary $\mathcal{A}$ running in time T that can distinguish between the ensembles $(\{a_i\}_{i\in[m]}, \{a_i s + e_i\}_{i\in[m]})$ and $(\{a_i\}_{i\in[m]}, \{r_i\}_{i\in[m]})$ where $\{a_i\}_{i\in[m]}, \{r_i\}_{i\in[m]} \leftarrow_\$ \mathcal{R}_q$, $s \leftarrow_\$ \mathcal{D}^{\mathsf{Bin\text{-}NTT}}$ and $\{e_i\}_{i\in[m]} \leftarrow \chi$ with probability $1/2 + \epsilon$.*

Then we can construct an adversary $\mathcal{B}$ running in time $poly(T, N, m, 1/\epsilon)$ who can return the secret s given $(\{a_i\}_{i\in[m']}, \{a_i s + e_i\}_{i\in[m']})$ where $\{a_i\}_{i\in[m']} \leftarrow_\$ \mathcal{R}_q$, $s \leftarrow_\$ \mathcal{D}^{\mathsf{Bin\text{-}NTT}}$ and $\{e_i\}_{i\in[m']} \leftarrow \chi$ with probability $1 - o(1)$.

Proof. We will follow the standard search-to-decision reduction for RLWE quite closely. Because each NTT coordinate of the secret is binary, we can guess each coordinate using a constant number of queries and thus avoid the factor of q slowdown present in the standard reduction. The main technical novelty in our reduction is a way to rerandomize the secret within the $\mathsf{Bin\text{-}NTT}$ distribution, which is a necessary component of this reduction strategy.

We start by reframing the Binary-NTT RLWE problem in the NTT domain. For simplicity, we will treat N and q as fixed.

Definition 11. *We define γ to be a primitive $2N$-th root of unity modulo q if it satisfies the following properties:*

- $\gamma \in \mathbb{Z}_q$
- $\gamma^{2N} = 1 \mod q$
- *For all $i < 2N$, we have $\gamma^i \neq 1 \mod q$*

Since we assume that q is a prime such that $2N \mid q - 1$, the existence of γ follows from elementary number theory.

Definition 12. *We denote the NTT representation of a ring element $x \in \mathcal{R}_q$ by $\widehat{x}$, which is defined as the length-N vector*

$$\widehat{x} = [x(\gamma), x(\gamma^3), x(\gamma^5), \cdots, x(\gamma^{2N-1})]$$

The first step of our reduction uses a hybrid argument to *specialize* the decision oracle $\mathcal{A}$. Consider the family of distributions defined by the following sampling procedure:

Definition 13 ($\mathcal{D}_s^{m,j}$)**.**

- *Sample m ring elements $\{a_i\}$ uniformly and independently from $\mathcal{R}_q$.*
- *For all i:*
  - *Set $b_i = a_i s + e_i$ where $e_i \leftarrow_\$ \chi$.*
  - *Replace the first j elements of $\widehat{b_i}$ by uniformly random elements of $\mathbb{Z}_q$ to obtain $\widehat{c_i}$.*
- *Return $(\{a_i\}, \{c_i\})$*

Definition 14 ($\mathcal{D}^{m,j}$)**.**

- *Sample s from $\mathcal{D}^{\mathsf{Bin\text{-}NTT}}$*
- *Return a sample from $\mathcal{D}_s^{m,j}$.*

Observe that $\mathcal{A}$ can distinguish between $\mathcal{D}^{m,0}$ and $\mathcal{D}^{m,N}$ with advantage ϵ in time T. By the hybrid lemma (or a simple telescoping argument), it is easy to see that there must be some index k such that $\mathcal{A}$ can distinguish between $\mathcal{D}^{m,k-1}$ and $\mathcal{D}^{m,k}$ with advantage at least ϵ/N in time T.

Our second step is demonstrating a worst-case to average-case reduction for the decision problem. We will randomize the secret so that $\mathcal{A}$ maintains its distinguishing advantage for an *arbitrary* s.

Lemma 21. *Assume that $\mathcal{A}$ runs in time T and can distinguish between the ensembles $\mathcal{D}^{m,k-1}$ and $\mathcal{D}^{m,k}$ with probability $1/2 + \epsilon/N$.*

Then there exists an adversary $\mathcal{A}'$ running in time $T + O(Nm \log N)$ which can distinguish between $\mathcal{D}_s^{m,k-1}$ and $\mathcal{D}_s^{m,k}$ with probability $1/2 + \epsilon/N$ for any $s \in \mathcal{R}_q$ satisfying $s = s^2$.

Algorithm $\mathcal{A}'$

Input:

- A PPT algorithm $\mathcal{A}$ for solving Binary-NTT RLWE(N, m, q, χ)
- $2m$ ring elements $(\{a_i\}_{i\in[m]}, \{c_i\}_{i\in[m]})$ which are sampled from either $\mathcal{D}_s^{m,k-1}$ or $\mathcal{D}_s^{m,k}$. Here s is an arbitrary ring element satisfying $s = s^2$

Objective: Determine which of the above distributions the input is sampled from.

Algorithm:

1. Sample a random binary vector $\vec{z}$ of length N.
2. For all $i \in [m]$:
   (a) Convert a_i and c_i to their NTT representations.
   (b) For all $j \in [N]$:
      i. If $\vec{z}[j] = 0$:
         A. Let $\widehat{a_i'}[j] \leftarrow \widehat{a_i}[j]$.
         B. Let $\widehat{c_i'}[j] \leftarrow \widehat{c_i}[j]$.
      ii. Else if $\vec{z}[j] = 1$:
         A. Let $\widehat{a_i'}[j] \leftarrow -\widehat{a_i}[j]$.
         B. Let $\widehat{c_i'}[j] \leftarrow \widehat{c_i}[j] - \widehat{a_i}[j]$.
   (c) Convert $\widehat{a_i'}$ and $\widehat{c_i'}$ to their coefficient representations.
3. Return $\mathcal{A}(\{a_i'\}_{i\in[m]}, \{c_i'\}_{i\in[m]})$.

Proof. The runtime of $\mathcal{A}'$ is dominated by the cost of $O(m)$ NTT and inverse NTT transformations before it calls $\mathcal{A}$. Therefore it runs in $T + O(Nm \log N)$ time.

Let s' be the coefficient representation of the vector $\widehat{s} \oplus \vec{z}$. We will show that $(\{a_i'\}, \{c_i'\})$ are distributed according to $\mathcal{D}_{s'}^{m,k-1}$ or $\mathcal{D}_{s'}^{m,k}$ depending on whether the input was sampled from $\mathcal{D}_s^{m,k-1}$ or $\mathcal{D}_s^{m,k}$. For sake of clarity, we will only demonstrate the second case (the other case can be proved identically).

Since a_i is random in $\mathcal{R}_q$, $\widehat{a_i}$ must be a random vector in $\mathbb{Z}_q^N$. Nontrivial linear transformations of random elements in a field are themselves random, and hence, $\widehat{a_i'}$ is also a random vector in $\mathbb{Z}_q^N$. This shows that a_i' is uniformly random in $\mathcal{R}_q$.

Observe that the first k entries of $\widehat{c_i}$ are completely random. The corresponding entries of $\widehat{c'_i}$ are set to either $\widehat{c_i}[j]$ or $\widehat{c_i}[j] - \widehat{a_i}[j]$ and are therefore also random in $\mathbb{Z}_q$. The remaining entries of $\widehat{c_i}$ are the same as that of $\widehat{b_i} = \widehat{a_i s + e_i}$. Because addition and multiplication in the NTT domain are componentwise, we have the relation

$$\widehat{c_i}[j] = \widehat{b_i}[j] = \widehat{a_i s + e_i}[j] = \widehat{a_i}[j] \cdot \widehat{s}[j] + \widehat{e_i}[j] \quad \forall j > k$$

Since $s = s^2$, $\widehat{s}[j]$ is either 0 or 1.

$$\begin{aligned}
\vec{z}[j] = 0 \implies \widehat{c'_i}[j] &= \widehat{c_i}[j] \\
&= \widehat{a_i}[j] \cdot \widehat{s}[j] + \widehat{e_i}[j] \\
&= \widehat{a'_i}[j] \cdot (\widehat{s}[j] \oplus \vec{z}[j]) + \widehat{e_i}[j] \\
&= \widehat{a'_i s' + e_i}[j] \\
\vec{z}[j] = 1 \implies \widehat{c'_i}[j] &= \widehat{c_i}[j] - \widehat{a_i}[j] \\
&= \widehat{a_i}[j] \cdot \widehat{s}[j] + \widehat{e_i}[j] - \widehat{a_i}[j] \\
&= \widehat{a_i}[j] \cdot (\widehat{s}[j] - 1) + \widehat{e_i}[j] \\
&= \widehat{a'_i}[j] \cdot (1 - \widehat{s}[j]) + \widehat{e_i}[j] \\
&= \widehat{a'_i}[j] \cdot (\vec{z}[j] \oplus \widehat{s}[j]) + \widehat{e_i}[j] \\
&= \widehat{a'_i s' + e_i}[j]
\end{aligned}$$

By construction, the NTT representation of s' is a completely random binary vector (since $\vec{z}$ is random). Therefore, $s' \leftarrow_\$ \mathcal{D}^{\mathsf{Bin\text{-}NTT}}$, and the input to $\mathcal{A}$ in line 3 is distributed according to either $\mathcal{D}^{m,k-1}$ or $\mathcal{D}^{m,k}$. The lemma statement now follows. □

Our third step is to convert $\mathcal{A}'$ into a search oracle that can return the k-th entry of $\hat{s}$. This can be done by adapting the standard guess-and check technique.

Algorithm $\mathcal{B}'$

Input:

- $2m$ ring elements $(\{a_i\}_{i\in[m]}, \{b_i\}_{i\in[m]})$ where $\{a_i\} \leftarrow_\$ \mathcal{R}_q$, $s \leftarrow_\$ \mathcal{D}^{\mathsf{Bin\text{-}NTT}}$, $\{e_i\} \leftarrow_\$ \chi$ and $b_i = a_i s + e_i$.

Objective: Find $\hat{s}[k]$

Algorithm:

1. For all $i \in [m]$:
   (a) Set $a'_i \leftarrow a_i$ and $b'_i \leftarrow b_i$.
   (b) Convert a'_i and b'_i to their NTT representations.
   (c) For all $j < k$:
      i. Sample $\widehat{b_i}[j] \leftarrow_\$ \mathbb{Z}_q$
   (d) Sample $\widehat{a'_i}[k] \leftarrow_\$ \mathbb{Z}_q$.
   (e) Convert $\widehat{a'_i}$ and $\widehat{b'_i}$ to their coefficient representations.
2. If $\mathcal{A}'(\{a'_i\}_{i\in[m]}, \{b'_i\}_{i\in[m]})$ returns $\mathcal{D}^{m,k}$, output 1.
3. Otherwise, output 0.

Lemma 22. *Assume that $\mathcal{A}'$ is a PPT algorithm that runs in time T' and can distinguish between $\mathcal{D}_s^{m,k-1}$ and $\mathcal{D}_s^{m,k}$ with probability $1/2 + \epsilon/N$ for any $s \in \mathcal{R}_q$ satisfying $s = s^2$.*

Then $\mathcal{B}'$ is a PPT algorithm with runtime $T' + O(Nm \log N)$ and it correctly returns $\hat{s}[k]$ with probability $1/2 + (\epsilon/N)$.

Proof. $\widehat{a'_i}$ is clearly uniformly random in $\mathbb{Z}_q^N$, and therefore $a'_i \leftarrow_\$ \mathcal{R}_q$. Let us consider the NTT representation $\widehat{b'_i}$. The first $k-1$ entries are uniformly random by construction. For the k-th entry, we have

$$\begin{aligned}
\widehat{b'_i}[k] &= \widehat{b_i}[k] \\
&= \widehat{a_i}[k]\widehat{s}[k] + \widehat{e_i}[k] \\
&= \widehat{a'_i s + e_i}[k] + (\widehat{a'_i}[k] - \widehat{a_i}[k])\hat{s}[k] \\
&= \begin{cases} \widehat{a'_i s + e_i}[k] & \text{if } \hat{s}[k] = 0 \\ \text{uniformly random in } \mathbb{Z}_q & \text{if } \hat{s}[k] = 1 \end{cases}
\end{aligned}$$

The remaining entries of $\widehat{b'_i}$ are equal to those of $\widehat{a'_i s + e_i}$. This shows that $\mathcal{A}'$ receives a sample from $\mathcal{D}^{m,k-1}$ if $\hat{s}[k] = 0$ and a samples from $\mathcal{D}^{m,k}$ if $\hat{s}[k] = 1$. The success probability of $\mathcal{B}'$ should therefore be exactly $1/2 + \epsilon/N$.

$\mathcal{B}'$ has the correct runtime since it only does $O(m)$ NTT transforms and some algebra for each coordinate of each input before calling $\mathcal{A}'$. □

The last step in our reduction is to use $\mathcal{B}'$ to extract the other coordinates of $\hat{s}$. We can do this by repeatedly rotating the arrays $\widehat{a_i}$ and $\widehat{b_i}$ before calling $\mathcal{B}'$, which imposes the same rotation on $\hat{s}$.

Lemma 23. *Assume that $\mathcal{B}'$ is a PPT algorithm that runs in time T' and returns $\hat{s}[k]$ with probability $1/2 + (\epsilon/N)$ on input $\{a_i, a_i s + e_i\}$ where $\{a_i\} \leftarrow_\$ \mathcal{R}_q$, $s \leftarrow_\$ \mathcal{D}^{\mathsf{Bin\text{-}NTT}}$ and $\{e_i\} \leftarrow_\$ \chi$.*

Then there exists a PPT algorithm $\mathcal{B}$ with runtime $O(T' N^3 \log N/\epsilon^2)$ which returns s given the same inputs with probability $1 - o(1)$.

Proof. The final algorithm $\mathcal{B}$ takes as input a much larger number of samples than the previous oracles. We set $m' = mN^3 \log N/\epsilon^2$. It extracts each coordinate of $\hat{s}$ in order. To obtain $\hat{s}[j]$, it uses $mN^2 \log N/\epsilon^2$ samples. $\mathcal{B}$ repeatedly takes a batch of m samples $(\{a_i\}_{i\in[m]}, \{b_i\}_{i\in[m]})$, rotates the corresponding NTT representations $k-j$ spaces to the right, and passes the resulting ring elements to $\mathcal{B}'$. We then take the majority output from the $N^2 \log N/\epsilon^2$ queries and let it be our guess for $\hat{s}[j]$. Once we have guessed all N coordinates, we simply perform a NTT transform and return our guess for s.

Algorithm $\mathcal{B}$

Input:

- $2m'$ ring elements $(\{a_i\}_{i\in[m']}, \{b_i\}_{i\in[m']})$ where $\{a_i\} \leftarrow_\$ \mathcal{R}_q$, $s \leftarrow_\$ \mathcal{D}^{\mathsf{Bin\text{-}NTT}}$, $\{e_i\} \leftarrow_\$ \chi$ and $b_i = a_i s + e_i$.

Objective: Find s
Algorithm:

1. Set $\hat{s} \leftarrow \mathbf{0}^N$
2. For all $j \in [N]$:
   (a) Set $G \leftarrow \mathbf{0}^{N^2 \log N/\epsilon^2}$.
   (b) For all $\ell \in [N^2 \log N/\epsilon^2]$:
      i. For all $i \in [m]$:
         A. Define $i' \leftarrow i + \ell m + jmN^2 \log N/\epsilon^2$
         B. Set $x_i \leftarrow a_{i'}(X^{2j+1})$
         C. Set $y_i \leftarrow b_{i'}(X^{2j+1})$
      ii. Set $G[\ell] \leftarrow \mathcal{B}'(\{x_i\}, \{y_i\})$
   (c) If G has more 0s than 1s, set $\hat{s}[(2kj + k + j) \mod N] \leftarrow 0$
   (d) Else, set $\hat{s}[(2kj + k + j) \mod N] \leftarrow 1$
3. Perform INTT on $\hat{s}$ to get s and output it.

If the coefficient vector of $e_{i'}$ be $\vec{c}$, we can expand $e_{i'}(X^{2j+1}) = \sum_{d=0}^{N-1} c_d X^{(2j+1)d}$. Since N is a power of 2, $2j+1$ is coprime to $2N$ and therefore multiplying by $2j+1$ and taking mod N is a permutation of $[N]$. If $(2j+1)d = d' \mod N$, we have $c_d X^{(2j+1)d}$ equal to either $c_d X^{d'}$ or $-c_d X^{d'}$ modulo $X^N + 1$. Therefore, the coefficients of $e_{i'}(X^{2j+1})$ can be obtained by permuting the coefficients of $e_{i'}$ and negating some of them. Recall that each coefficient of $e_{i'}$ is sampled iid from χ, where χ is symmetric around 0. This implies that $e_{i'}(X^{2j+1})$ has the same distribution as $e_{i'}$.

Define $s' = s(X^{2j+1})$. It is easy to see that $y_i = x_i s' + e_{i'}(X^{2j+1})$ as well as $s'^2 = s'$. We therefore call $\mathcal{B}'$ with a valid set of inputs every time. Since each of the original m' tuples are sampled independently except for the shared secret, the calls to $\mathcal{B}'$ each have a success probability $1/2 + \epsilon/N$ independent of each other.

For a fixed j, we call $\mathcal{B}'$ a total of $N^2 \log N/\epsilon^2$ times and take the majority. By the Chernoff bound, this gives us $\widehat{s'}[k]$ with probability $1 - 1/N^2$. Observe that

$$\begin{aligned}
\widehat{s'}[k] &= s'(g^{2k+1}) && \text{where } g \text{ is a primitive } 2N^{th} \text{ root of unity modulo } q \\
&= s(X^{2j+1})(g^{2k+1}) \\
&= s(g^{4kj+2k+2j+1})
\end{aligned}$$

Let $x = (2kj + k + j) \mod N$

$$\begin{aligned}
&= s(g^{2x+1}) && \text{since } g^{2N} = 1 \\
&= \hat{s}[x]
\end{aligned}$$

Also, each value of $j \in [N]$ produces a distinct x since

$$\begin{aligned} 2jk + j + k &= 2j'k + j' + k \mod N \\ \implies (2k+1)(j-j') &= 0 \mod N \\ \implies j &= j' \mod N \qquad \text{since } N \text{ is a power of 2} \end{aligned}$$

Therefore, we populate each entry of s' correctly with probability $1 - 1/N^2$. By the union bound, the entire vector is reconstructed correctly with probability at least $1 - 1/N = 1 - o(1)$, as desired.

We call $\mathcal{B}'$ a total of $N^3 \log N/\epsilon^2$ times. Apart from that, we only do a little bit of algebra which is dominated by the INTT operation at the end. The total runtime is therefore $O(T'N^3 \log N/\epsilon^2)$.

□

Proof of Theorem 20. The theorem statement follows directly from chaining the previous 3 lemmas. □

6.2 Security against known attacks

Let us now consider how one might attempt to break the search version of the Binary-NTT RLWE assumption. Recall that we are given ring elements $\{a_i, b_i\}_{i\in[m]}$. Let $s = \sum_{j=0}^{N-1} s_j X^j \in \mathcal{R}_q$ be the secret. As before, we will denote the NTT representation of s as $\hat{s}$.

Note that one can use any and all attacks against LWE or RLWE[2, 93, 82, 26, 35] to solve instances of this problem. However, to our knowledge, these attacks do not leverage the algebraic structure of our new assumptions. Thus, for these attacks, we can choose the parameters according to the LWE-estimator [1] as other schemes Below, we will focus on how one might take advantage of the additional structure in our problem beyond standard RLWE.

6.2.1 Local Attacks

6.2.2 Binary-secret LWE-based Attack

Observe that we can write the product sa_i as $\text{INTT}(\text{NTT}(s) \odot \text{NTT}(a_i))$ where $\odot$ denotes componentwise multiplication. Define INTT_x to be the matrix obtained by multiplying the i-th row of the INTT matrix by x_i for all $i \in [N]$. We can then write $b_i = \hat{s} \cdot \text{INTT}_{\text{NTT}(a_i)} + e_i$. Here $\hat{s}$ is a binary vector since $s = s^2$ and each entry of e_i is i.i.d sampled from χ. We can therefore concatenate the $\text{INTT}_{\text{NTT}(a_i)}$ matrices and the b_i vectors to obtain an instance of binary-secret LWE. The public matrix has a certain algebraic structure, but similar to standard RLWE, it is unclear how one can take advantage of it. Concretely, we choose our parameters such that we have λ-bit security against off-the-shelf binary-secret LWE attacks [20, 85, 91, 1, 94, 2].

6.2.3 Algebraic Attacks

This class of attacks represent the problem as a system of equations. We describe two ways to achieve that for Binary-NTT RLWE. In the following, we denote the coefficient vectors of ring element x by $\vec{x}$. For any vector, define $\vec{x}[-j] := -\vec{x}[j]$.

1. Observe that the secret is statistically unique given just one sample (a, b). We have

$$\begin{aligned} b - as = e &\implies \left(\frac{b-e}{a}\right) = s \\ &\implies \left(\frac{b-e}{a}\right)^2 = \left(\frac{b-e}{a}\right) \end{aligned}$$

Expanding this equation in terms of the coefficients of e, we get N quadratic equations in N variables. Since each of these variables is sampled i.i.d from χ, we can assume they all lie in $[-B, B]$ for some small bound B that depends on the width of error distribution χ. Therefore we can write the following equation for each coefficient of e:

$$\prod_{k=-B}^{B} (\vec{e}[i] - k) = \vec{e}[i] \cdot \prod_{k=1}^{B} \left(\vec{e}[i]^2 - k^2\right) = 0 \tag{1}$$

We thus end up with N additional single variable equations of degree $2B + 1$

2. We can also use multiple samples to generate a larger system of equations. Note that we have two kinds of information regarding s.

   First, as in RLWE, we have $e_i = b_i - a_i s$ where each e_i is a small norm polynomial. In particular, since each coefficient of e_i is sampled independently from χ, we can expand the above equations in terms of the coefficients of s and end up with a total of mN linear expressions in $\{\vec{s}[j]\}_{j\in[N]}$ each of which must be small . A typical such expression would look like

   $$\vec{e_i}[j] = \vec{b_i}[j] - \prod_{k=0}^{N-1} \vec{a_i}[k]\vec{s_i}[j-k].$$

   As before, we assume that each $\vec{e_i}[j]$ is in $[-B, B]$ for some small B and write equations as Eq. (1). This gives us mN equations of the above form, each of degree $2B + 1$.

   Second, we have the single additional piece of information that $s^2 = s$. This expands to a set of N quadratic equations in $\{\vec{s_j}\}_{j\in[N]}$. A typical such equation looks like

   $$\vec{s_i}[j] = \prod_{k=0}^{N-1} \vec{s_i}[k]\vec{s_i}[j-k].$$

We can use different techniques to solve these high-degree systems.

- Arora-Ge Style Linearization: We can simply treat each monomial as its own variable and treat it as a linear system. This blows up the number of variables. If we have fewer equations than variables after this stage, we repeatedly multiply the smallest degree equations by all possible degree 1 monomials to get additional equations; this process increases the number of equations faster than the number of variables. This is the approach taken in [7].

  Observe that the fact that $s = s^2$ only yields N quadratic equations in N variables in both of the above systems. Since the highest degree is $2B + 1$ in both systems, the linearization process creates an additional $O(N^{2B+1})$ equations. Therefore, this approach can not perform better against Binary-NTT RLWE with m samples than against RLWE with $m + O(N^{2B+1})$ samples. In the LWE-estimator [1], we set the number of samples to be infinity, which thus captures this sample increase.

- Multivariate Quadratic attacks: There is a rich literature behind solving a system of quadratic equations in multiple variables. We can try to make use of those algorithms if we convert our equations into quadratic ones. This can be done by introducing a new variable for all high degree monomials and introducing a quadratic equation for each such variable representing it as a product of 2 other monomials. (As an example, one can replace $f(xyz) = 0$ by introducing variables u, v and equations $u = xy, v = uz, f(v) = 0$.)

	Boolean Gates (ms)				**Arb. Func.** (ms)
	STD_128		MEDIUM		STD_128
Implementation	No Prep.	With Prep.	No Prep.	With Prep.	No Prep.
TFHE (OpenFHE) [29, 8]	36.5	–	30.1	–	391.5
Ours (sparse binary)	7.9	–	5.8	–	53.5*
Ours (structured sparse)	8.9	5.7	6.5	4.5	58.2*

Table 2: Runtime comparison. Ours (sparse binary) and (structured sparse) use Algorithm 2 with general binary secrets and structured secrets, respectively (see Section 4.2). Prep. refers to the preprocessing technique in Section 4.3, which enhances runtime at the cost of larger bootstrapping keys. "–" means preprocessing does not improve the runtime with the current proof of concept implementation. "*" stands for estimated runtime using microbenchmarks with our current implementation. See Section 7.1 for details.

Applying this transformation to the first system, we get $N(B+2)$ equations in $N(B+1)$ variables. The second system yields $mN(B+2)+N$ equations in $mN(B+1)+N$ variables. Since the number of equations is much smaller than the square of the number of variables, all known Multivariate Quadratic (MQ) algorithms require exponential time to solve this system [37, 38, 31, 60, 98, 59].

Note that some such attacks (e.g., [38]) work directly with a high degree and do not need to transform the equations to have degree 2, but such attacks similarly take exponential time due to the number of equations we have.

7 Evaluation

7.1 FHEW/TFHE Bootstrapping with Sparse Keys

Methodology. We choose parameters according to the MEDIUM security (103-bit according to [1]) and STD_128 security (120-bit according to [1]) in OpenFHE[9] used for TFHE in OpenFHE [8], and guarantee the same correctness probability (by making the error at being at most the amount of error for the TFHE implementation of OpenFHE). To this end, we choose the lattice parameters as in Table 3 for general sparse binary secret keys. Furthermore, we choose PBC (Definition 5) with $c = 3$ and $k = h + 3$, which guarantees correctness $> 1/2$ for parameters we use, according to the same tests as in prior works [5, 3]. We believe that this is sufficient, since during the key generation, the user can simply reject sampling the PBC randomness if PBC fails, which leaks < 1 bit of information (since the failure probability is $< 1/2$), and as discussed in Section 3.2, entropic LWE is as secure as regular LWE, so we conjecture that this $\ll 1$-bit information loss does not affect concrete security by more than 1 bit.[10]

We also choose parameters for structured binary secrets (our stepping stone construction). Since there is no standard way to estimate the security for such a secret, we use the entropy: we use the assumption that the same entropy of structured binary secrets has the same concrete security as the general sparse binary secrets. In other words, for structured secrets with secret

[9]Note that while OpenFHE parameter selection was designed for 128-bit security, as the lattice-estimator advances, the concrete security reduces from 128 bits to 120 bits. We choose the same level of security for fair comparisons.

[10]One could alternatively use $k = 1.5h$ as used in prior works [5, 3] to guarantee negligible failure probability, but it reduces or concrete runtime by a factor of $\sim$1.5 for our constructions.

			LWE Parameters			RLWE Parameters		
Parameters	**Functionality**	**Security**	n	q	h	N	Q	Q'
Param 1	Gate Boots	`STD_128`	1024	512	43	1024	2^{28}	2^{12}
Param 2	Gate Boots	`MEDIUM`	1024	512	31	1024	2^{28}	2^{12}
Param 3	Arb Func	`STD_128`	2048	2^{12}	70	2048	2^{54}	2^{35}

Table 3: Parameters for general sparse binary secret keys. Q' is an intermediate ciphertext modulus used by OpenFHE. The rest of the parameters are the same as OpenFHE, so we omit the repetition.

			LWE Parameters			RLWE Parameters		
Parameters	**Functionality**	**Security**	n	q	h	N	Q	Q'
Param 1	Gate Boots	`STD_128`	1024	512	64	1024	2^{28}	2^{12}
Param 2	Gate Boots	`MEDIUM`	990	512	45	1024	2^{28}	2^{12}
Param 3	Arb Func	`STD_128`	2040	2^{12}	102	2048	2^{54}	2^{35}

Table 4: Parameters for structured binary secret keys.

length n' hamming weight h', we make sure $(n'/h')^{h'} \geq \binom{n}{h}$ for general sparse binary key with length n hamming weight h. Note that if this assumption is too aggressive, one should consider using the general sparse binary secret parameters in Table 3. We give the parameters in Table 4.

Performance. Then, we show the runtime comparisons of our constructions compared to TFHE implemented by OpenFHE in Table 2. As shown, using sparse secret keys, our scheme can achieve $\sim$4.5–7.5$\times$ runtime improvement compared to TFHE, for boolean gate bootstrapping and arbitrary function evaluation. For arbitrary function evaluation, the runtime improvement is more significant since the LWE secret key length is larger, making the gap between the sparsity and secret key length a lot larger. Using our preprocessing technique in Section 4.3, the runtime for structured sparse secret keys can be further improved, since we can save h of NTTs. However, since the key size is extremely large ($>$ 10 GB), the memory read becomes the main performance bottleneck when performing additions. Thus, for general sparse binary secret keys, this preprocessing does not improve the runtime. In Appendix B, we show the runtime for using ternary keys, which has similar gains as binary keys.

7.2 Tree-shaped Bootstrapping with Sparse Keys

This section gives the estimated runtime of our Algorithm 3.

Methodology. We first discuss the parameter choice. We conjecture that our assumptions in Section 3.2 is as hard as RLWE with binary secrets (in its coefficient form) and choose the parameters accordingly. For LWE parameters, we choose $n = 1450$, $h = 29$, $q = 512$, $\sigma = 3.2$ (standard deviation of the noise distribution for both key-switching key and LWE ciphertexts), and RLWE parameters, we choose $N = 4096$, $Q \approx 2^{55+50}$, $d = 4$ (where d is the number of digits for key-switching), and $\sigma' = 0.75$. We additionally use $Q' = 2^{12}$ for intermediate modulus switching key-switching from RLWE to LWE, and a modulus $Q'' \approx 2^{50}$, which will be detailed below. These parameters guarantee $>$ 128-bit of computational security according to lattice-

	Operation Counts			Circuit Depth	
Scheme	**Int. Additions**	**Int. Mults**	**Improv.**	**NTT Depth**	**Improv.**
TFHE [29, 8]	30,912,512	15,460,352	—	503	—
Algorithm 2	31,768,576	3,305,472	∼4×	46	∼10×
Algorithm 3	80,101,376	5,128,192	∼1.5× (1×)	3	∼100×

Table 5: Performance complexity. Comparison of operation counts and circuit depth between TFHE and proposed algorithms. The parentheses of improvement for Algorithm 3 is the estimated improvement adjusted due to the use of 64-bit machine word size instead of 32-bit. See Section 7.2 for details.

estimator [1]. Then, for the underlying PBC, we again choose $c = 3, k = h+3$. This means that when organized as a tree, we have $\log(29+3) = 5$ levels of multiplication. Note that we choose two different secret keys, one with uniform binary coefficients and one with uniform binary in NTT form. To perform these 5 levels of multiplication, we first use a binary NTT secret key to multiply for 3 levels *without* modulus switching, followed by a modulus switching from a 105-bit Q to a 50-bit Q''. Then, we perform two levels of multiplication *with* modulus switching using a binary coefficient secret key.[11]

To estimate the runtime, we calculate the number of 64-bit integer operations needed. For an x-bit Q, a $\mathbb{Z}_Q$ operation takes $\lceil x/64 \rceil$ operations.

Our algorithm (Algorithm 3) can be estimated using the following steps: (1) PBC-based summation (line 21-22), which takes $N \cdot n \cdot c \cdot \lceil \log(Q)/64 \rceil \cdot 2$ additions; (2) INTT, which takes $k \cdot N \cdot \log(N) \cdot \lceil \log(Q)/64 \rceil$ multiplications; (3) binary NTT multiplication for $k-3$ multiplications (no key switching or modulus switching needed), which takes $(k-3) \cdot 4 \cdot N \cdot \lceil \log(Q)/64 \rceil$ multiplications; (4) key-switching for 4 ciphertexts, where each key-switching is dominated by $2 \cdot (d+1)$ NTT/INTTs, which therefore takes $4 \cdot 2 \cdot (d+1) \cdot N \cdot \log(N)$ multiplications; (5) regular BV-like multiplications for 3 multiplications, which is dominated by 3 key-switchings, thereby taking $3 \cdot 2 \cdot (d+1) \cdot N \cdot \log(N)$ multiplications.

Then, we use a similar way to estimate the number of operations for TFHE in OpenFHE [8] and for our construction in Algorithm 2 with Param 1 and a general sparse binary key, and compare the results in Table 5.

Runtime estimation for a single CPU core. As shown in Table 5, Algorithm 3 achieves $> 2\times$ reduction in terms of multiplications, while having $< 4\times$ increase in additions. To estimate the overall improvement, we rely on our micro-benchmarks using OpenFHE [8] for our Algorithm 2, which indicates the operation count improvement is ∼1.5× compared to TFHE [29, 8]. Compared to our Algorithm 2, the reduction in the number of operations is not as significant (which matches our proof of concept implementation tests shown in Table 2).

Note that, however, for TFHE [29, 8] and Algorithm 2, the underlying integer operations are only 32-bit. Thus, their per-integer operation is ∼1.5× faster than Algorithm 3. Overall, the single-core CPU runtime of Algorithm 3 is expected to be roughly the same as TFHE [29, 8] and 4-5× slower than Algorithm 2.

Circuit depth and GPU-friendliness. The most essential advantage of Algorithm 3 is its circuit depth. As shown in Table 5, its NTT/INTT depth is only 3, compared to 46 for

[11]Note that this step would require a Q''' satisfying $Q' < Q''' < Q''$ as an intermediate modulus. However, since it does not affect the runtime, we omit Q''' for simplicity.

Algorithm 2, and 503 for TFHE [8]. As discussed in Section 5.2, NTT/INTT is the performance bottleneck for GPU: For parameters used in TFHE and our constructions, the GPU runtime for a single NTT/INTT is essentially the same as a CPU runtime [106, 12]. Thus, due to our high parallelizability, we estimate that using GPU, our speed-up can be close to two orders of magnitude.

References

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

[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] A. Ali, T. Lepoint, S. Patel, M. Raykova, P. Schoppmann, K. Seth, and K. Yeo. Communication–Computation trade-offs in PIR. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021.

[4] J. Alperin-Sheriff and C. Peikert. Practical bootstrapping in quasilinear time. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 1–20, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[5] S. Angel, H. Chen, K. Laine, and S. T. V. Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2018.

[6] B. Applebaum, D. Cash, C. Peikert, and A. Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In *Annual International Cryptology Conference*, pages 595–618. Springer, 2009.

[7] S. Arora and R. Ge. New algorithms for learning in presence of errors. In *International Colloquium on Automata, Languages, and Programming*, pages 403–415. Springer, 2011.

[8] 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.

[9] L. Bergerat, A. Boudi, Q. Bourgerie, I. Chillotti, D. Ligier, J.-B. Orfila, and S. Tap. Parameter optimization and larger precision for (T)FHE. *Journal of Cryptology*, 36(3), July 2023.

[10] L. Bergerat, I. Chillotti, D. Ligier, J.-B. Orfila, A. Roux-Langlois, and S. Tap. New secret keys for enhanced performance in (t)FHE. Cryptology ePrint Archive, Paper 2023/979, 2023.

[11] L. Bergerat, J.-B. Orfila, A. Roux-Langlois, and S. Tap. Accelerating TFHE with sorted bootstrapping techniques. In G. Hanaoka and B.-Y. Yang, editors, *ASIACRYPT 2025, Part VII*, volume 16251 of *LNCS*. Springer, Singapore, Dec. 2025.

[12] F. Boemer, S. Kim, G. Seifu, F. D.M. de Souza, and V. Gopal. Intel hexl: Accelerating homomorphic encryption with intel avx512-ifma52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[13] D. Boneh, S. Eskandarian, L. Hanzlik, and N. Greco. Single secret leader election. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, AFT '20, page 12–24, New York, NY, USA, 2020. Association for Computing Machinery.

[14] 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.

[15] F. Bourse, M. Minelli, M. Minihold, and P. Paillier. Fast homomorphic evaluation of deep discretized neural networks. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*. Springer, Cham, Aug. 2018.

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

[17] Z. Brakerski and N. Döttling. Hardness of LWE on general entropic distributions. In *EUROCRYPT 2020, Part II*, LNCS, May 2020.

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

[19] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Annual cryptology conference*, pages 505–524. Springer, 2011.

[20] J. Buchmann, F. Göpfert, R. Player, and T. Wunderer. On the hardness of lwe with binary error: Revisiting the hybrid lattice-reduction and meet-in-the-middle attack. In *International Conference on Cryptology in Africa*, pages 24–43. Springer, 2016.

[21] A. Burton, S. J. Menon, and D. J. Wu. Respire: High-rate PIR for databases with small records. ACM CCS 2025, 2024.

[22] 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.

[23] H. Chen and K. Han. Homomorphic lower digits removal and improved FHE bootstrapping. In J. B. Nielsen and V. Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, Apr. / May 2018.

[24] H. Chen, Z. Huang, K. Laine, and P. Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In *ACM CCS 2018*. ACM Press, Oct. 2018.

[25] H. Chen, K. Laine, and P. Rindal. Fast private set intersection from homomorphic encryption. In *ACM CCS 2017*. ACM Press, Oct. / Nov. 2017.

[26] H. Chen, K. Lauter, and K. E. Stange. Attacks on the search rlwe problem with small errors. *SIAM Journal on Applied Algebra and Geometry*, 1(1):665–682, 2017.

[27] 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.

[28] 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.

[29] 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.

[30] 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 *ACM CCS 2021*. ACM Press, Nov. 2021.

[31] N. Courtois, A. Klimov, J. Patarin, and A. Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 392–407. Springer, 2000.

[32] A. Dalvi, A. Jain, S. Moradiya, R. Nirmal, J. Sanghavi, and I. Siddavatam. Securing neural networks using homomorphic encryption. In *2021 International Conference on Intelligent Technologies (CONIT)*, pages 1–7, 2021.

[33] K. Duan, H. Li, D. Liu, and G. Ma. Large-plaintext functional bootstrapping in FHE with small bootstrapping keys. Cryptology ePrint Archive, Paper 2025/1717, 2025.

[34] 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.

[35] Y. Elias, K. E. Lauter, E. Ozman, and K. E. Stange. Ring-lwe cryptography for the number theorist. In *Directions in Number Theory: Proceedings of the 2014 WIN3 Workshop*, pages 271–290. Springer, 2016.

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

[37] J.-C. Faugere. A new efficient algorithm for computing gröbner bases (f4). *Journal of pure and applied algebra*, 139(1-3):61–88, 1999.

[38] J. C. Faugère. A new efficient algorithm for computing gröbner bases without reduction to zero (f5). In *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation*, ISSAC '02, New York, NY, USA, 2002. Association for Computing Machinery.

[39] J. Feng, B. Wu, D. Lin, and B. Xiang. Accelerating NTRU-based bootstrapping with block key distributions. ISC'25, 2025.

[40] B. Fisch, A. Lazzaretti, Z. Liu, and C. Papamanthou. Thorpir: Single server pir via homomorphic thorp shuffles. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, New York, NY, USA, 2024. Association for Computing Machinery.

[41] B. Fisch, Z. Liu, E. Tromer, and Y. Wang. UnifOMR: Oblivious message retrieval with near-optimal concrete efficiency. ACM CCS 2026, 2026.

[42] R. Geelen and F. Vercauteren. Bootstrapping for bgv and bfv revisited. *J. Cryptol.*, 36(2), mar 2023.

[43] 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.

[44] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, Aug. 2013.

[45] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *CRYPTO 2013, Part I*, LNCS, Aug. 2013.

[46] S. Goldwasser, Y. T. Kalai, C. Peikert, and V. Vaikuntanathan. Robustness of the learning with errors assumption. In *ICS 2010*. Tsinghua University Press, Jan. 2010.

[47] A. Guimarães, E. Borin, and D. F. Aranha. Revisiting the functional bootstrap in TFHE. *IACR TCHES*, 2021(2), 2021.

[48] 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`.

[49] S. Halevi and V. Shoup. Bootstrapping for HElib. *Journal of Cryptology*, 34(1), Jan. 2021.

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

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

[52] A. Henzinger, E. Dauterman, H. Corrigan-Gibbs, and N. Zeldovich. Private web search with tiptoe. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 396–416, New York, NY, USA, 2023. Association for Computing Machinery.

[53] D. Hong and Y. Lee. Optimizing FHEW-like homomorphic encryption schemes with smooth performance-failure trade-offs. Cryptology ePrint Archive, Paper 2025/1892, 2025.

[54] W. jie Lu, Z. Huang, C. Hong, Y. Ma, and H. Qu. PEGASUS: Bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2021.

[55] M. Joye and P. Paillier. Blind rotation in fully homomorphic encryption with extended keys. In *Cyber Security, Cryptology, and Machine Learning: 6th International Symposium, CSCML 2022, Be'er Sheva, Israel, June 30 – July 1, 2022, Proceedings*, Berlin, Heidelberg, 2022. Springer-Verlag.

[56] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In W. Enck and A. P. Felt, editors, *USENIX Security 2018*. USENIX Association, Aug. 2018.

[57] A. Kim, Y. Polyakov, and V. Zucca. Revisiting homomorphic encryption schemes for finite fields. In *ASIACRYPT 2021*. Springer, 2021.

[58] 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`.

[59] A. Kipnis, J. Patarin, and L. Goubin. Unbalanced oil and vinegar signature schemes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 206–222. Springer, 1999.

[60] A. Kipnis and A. Shamir. Cryptanalysis of the hfe public key cryptosystem by relinearization. In *Annual International Cryptology Conference*, pages 19–30. Springer, 1999.

[61] C. Lee, S. Min, J. Seo, and Y. Song. Faster TFHE bootstrapping with block binary keys. In J. K. Liu, Y. Xiang, S. Nepal, and G. Tsudik, editors, *ASIACCS 23*. ACM Press, July 2023.

[62] D. Lee, S. Min, and Y. Song. Functional bootstrapping for packed ciphertexts via homomorphic LUT evaluation, 2024.

[63] J.-W. Lee, H. Kang, Y. Lee, W. Choi, J. Eom, M. Deryabin, E. Lee, J. Lee, D. Yoo, Y.-S. Kim, and J.-S. No. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access*, 10:30039–30054, 2022.

[64] J.-W. Lee, E. Lee, Y.-S. Kim, and J.-S. No. Rotation key reduction for client-server systems of deep neural network on fully homomorphic encryption. In J. Guo and R. Steinfeld, editors, *Advances in Cryptology – ASIACRYPT 2023*, pages 36–68, Singapore, 2023. Springer Nature Singapore.

[65] 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*. EUROCRYPT 2021, 06 2021.

[66] K. Lee and Y. Yeo. SophOMR: Improved oblivious message retrieval from SIMD-aware homomorphic compression. Cryptology ePrint Archive, Paper 2024/1814, 2024.

[67] K. Lee and J. W. Yoon. Discretization error reduction for high precision torus fully homomorphic encryption. In A. Boldyreva and V. Kolesnikov, editors, *PKC 2023, Part II*, volume 13941 of *LNCS*. Springer, Cham, May 2023.

[68] 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.

[69] 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.

[70] Z. Li, X. Lu, Z. Wang, R. Wang, Y. Liu, Y. Zheng, L. Zhao, K. Wang, and R. Hou. Faster NTRU-based bootstrapping in less than 4 ms. *IACR TCHES*, 2024(3), 2024.

[71] H. Liang, Z. Liu, E. Tromer, X. Xie, and Y. Yu. InstantOMR: Oblivious message retrieval with low latency and optimal parallelizability. USENIX Security 2026, 2026.

[72] H. Liang, Z. Liu, Y. Wang, and F. Z. Xiang Xie and, Yu Yu and. Relect: Single secret leader election via fhe with reduced computation and communication and transparent setup. ACM CCS 2026, 2026.

[73] C. Lin, Z. Liu, and T. Malkin. XSPIR: Efficient symmetrically private information retrieval from ring-LWE. In V. Atluri, R. Di Pietro, C. D. Jensen, and W. Meng, editors, *ESORICS 2022, Part I*, volume 13554 of *LNCS*, Sept. 2022.

[74] 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.

[75] 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.

[76] 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*. Springer-Verlag, 2023.

[77] Z. Liu, K. Sotiraki, E. Tromer, and Y. Wang. Snake-eye resistant PKE from LWE for oblivious message retrieval and robust encryption. In S. Fehr and P.-A. Fouque, editors, *EUROCRYPT 2025, Part III*, volume 15603 of *LNCS*, pages 126–156. Springer, Cham, May 2025.

[78] Z. Liu, E. Tromer, and Y. Wang. Group oblivious message retrieval. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4367–4385, 2024.

[79] Z. Liu, E. Tromer, and Y. Wang. PerfOMR: Oblivious message retrieval with reduced communication and computation. Usenix Security'24, 2024.

[80] Z. Liu and Y. Wang. Amortized functional bootstrapping in less than 7 ms, with $\tilde{\mathrm{O}}(1)$ polynomial multiplications. In J. Guo and R. Steinfeld, editors, *ASIACRYPT 2023, Part VI*, volume 14443 of *LNCS*. Springer, Singapore, Dec. 2023.

[81] Z. Liu and Y. Wang. Relaxed functional bootstrapping: A new perspective on BGV/BFV bootstrapping. In K.-M. Chung and Y. Sasaki, editors, *ASIACRYPT 2024, Part I*, volume 15484 of *LNCS*. Springer, Singapore, Dec. 2024.

[82] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 1–23. Springer, 2010.

[83] S. Ma, T. Huang, A. Wang, and X. Wang. Accelerating bgv bootstrapping for large p using null polynomials over $\mathbb{Z}_{p^e}$. Eurocrypt 2024, 2024. `https://eprint.iacr.org/2024/115`.

[84] 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*. IEEE Computer Society Press, May 2022.

[85] D. Micciancio. On the hardness of learning with errors with binary secrets. *Theory of Computing*, 14(1):1–17, 2018.

[86] D. Micciancio and Y. Polyakov. Bootstrapping in FHEW-like cryptosystems. Cryptology ePrint Archive, Paper 2020/086, 2020.

[87] D. Micciancio and Y. Polyakov. *Bootstrapping in FHEW-like Cryptosystems*. Association for Computing Machinery, 2021.

[88] D. Micciancio and V. Vaikuntanathan. Sok: learning with errors, circular security, and fully homomorphic encryption. In *IACR International Conference on Public-Key Cryptography*, pages 291–321. Springer, 2024.

[89] 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.

[90] 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`.

[91] N. Nolte, M. Malhou, E. Wenger, S. Stevens, C. Li, F. Charton, and K. Lauter. The cool and the cruel: separating hard parts of lwe secrets. In *International Conference on Cryptology in Africa*, pages 428–453. Springer, 2024.

[92] H. Okada, R. Player, and S. Pohmann. Homomorphic polynomial evaluation using galois structure and applications to bfv bootstrapping. Asiacrypt 2023. `https://eprint.iacr.org/2023/1304`.

[93] C. Peikert. A decade of lattice cryptography. *Foundations and Trends$\hat{W}$ in Theoretical Computer Science*, 10(4):283–424, 2016.

[94] R. Player. *Parameter selection in lattice-based cryptography*. PhD thesis, Royal Holloway, University of London, 2018.

[95] R. L. Rivest, L. Adleman, M. L. Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

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

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

[98] E. Thomae and C. Wolf. Solving underdetermined systems of multivariate quadratic equations revisited. In *International workshop on public key cryptography*, pages 156–171. Springer, 2012.

[99] H. Wang, M. Luo, H. Xia, M. Wang, and H. Hou. Accelerating FHEW-like bootstrapping via new configurations of the underlying cryptosystems. Cryptology ePrint Archive, Paper 2025/1711, 2025.

[100] R. Wang, J. Bai, Y. Liu, X. Zhang, X. Lu, L. Zhao, K. Wang, and R. Hou. Bootstrapping over free $\mathcal{R}$-module. Cryptology ePrint Archive, Paper 2025/1753, 2025.

[101] Y. Wang and F. Zhang. Qelect: Lattice-based single secret leader election made practical. Usenix Security 2025, 2025.

[102] B. Xiang, J. Zhang, Y. Deng, Y. Dai, and D. Feng. Fast blind rotation for bootstrapping FHEs. Cryptology ePrint Archive, Paper 2023/1564, 2023.

[103] K. Yeo. Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. In H. Handschuh and A. Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*. Springer, Cham, Aug. 2023.

[104] M. Zhou, W.-K. Lin, Y. Tselekounis, and E. Shi. Optimal single-server private information retrieval. In C. Hazay and M. Stam, editors, *EUROCRYPT 2023, Part I*, volume 14004 of *LNCS*, pages 395–425. Springer, Cham, Apr. 2023.

[105] T. ZHOU, X. YANG, L. LIU, W. ZHANG, and Y. DING. Faster bootstrapping with multiple addends. IEEE Access, 2018.

[106] A. Şah Özcan and E. Savaş. Two algorithms for fast GPU implementation of NTT. Cryptology ePrint Archive, Paper 2023/1410, 2023.

A Additional Preliminary

A.1 Overview of Basic FHE Operations

We give a more detail overview on the homomorphic operations we recalled in Section 3.3.2.

Again, for all the operations below, unless otherwise stated, the secret key s and ciphertext modulus q remain unchanged.

Additions.

Interface: $\mathsf{Enc}(m_1) + \mathsf{Enc}(m_2) \to \mathsf{Enc}(m_1 + m_2)$

Input: $\mathsf{Enc}(m_1)$, $\mathsf{Enc}(m_2)$ where $\mathsf{Enc} \in \{\mathsf{LWE}, \mathsf{LWE}_{\mathsf{msb}}, \mathsf{RLWE}, \mathsf{RLWE}_{\mathsf{msb}}, \mathsf{RLWE}_{\mathsf{lsb}}, \mathsf{RLWE}', \mathsf{RLWE}'_{\mathsf{msb}}, \mathsf{RGSW}, \mathsf{RGSW}_{\mathsf{msb}}\}$

Output: $\mathsf{Enc}(m_1 + m_2)$ where Enc is the same scheme as the inputs

Algorithm: The algorithm is simply adding two ciphertexts coordinate-wise. The correctness is easy to see and the error grows is also simply the two errors added.

Cost: A LWE ciphertext is a vector of length $n+1$, so it takes $n+1$ $\mathbb{Z}_q$ additions to add them homomorphically. For the other schemes, assume the ring dimension used is N. Since all the variants of RLWE ciphertexts have 2 ring elements, it takes $2N$ $\mathbb{Z}_q$ additions to add them homomorphically. RLWE' ciphertexts require $2N \cdot \mathsf{dig}$ $\mathbb{Z}_q$ additions because they are composed of dig RLWE ciphertexts we must add individually. Similarly, RGSW ciphertexts require $4N \cdot \mathsf{dig}$ $\mathbb{Z}_q$ additions.

Monomial multiplication.

Interface: $X^c \times \mathsf{Enc}(m) \to \mathsf{Enc}(m \cdot X^c)$

Input: $c \in [N]$, $\mathsf{Enc}(m)$ where $\mathsf{Enc} \in \{\mathsf{RLWE}, \mathsf{RLWE}_{\mathsf{msb}}, \mathsf{RLWE}_{\mathsf{lsb}}, \mathsf{RLWE}', \mathsf{RGSW}\}$

Output: $\mathsf{Enc}(m \cdot X^c)$ where Enc is the same scheme as the inputs

Algorithm: For a RLWE ciphertext, we simply multiply both a and b by X^c. It is easy to see that this results in a ciphertext for $m \cdot X^c$ with error $e \cdot X^c$. Since $X^n = -1$ in our ring, this multiplying by a monomial does not change the magnitude of the coefficients, and hence the new error follows the same distribution as long as χ is symmetric around 0. RLWE$'$ and RGSW ciphertexts are simply vectors of RLWE ciphertexts, and we can perform monomial multiplication with them by just applying the above procedure to each component.

Cost: We need to efficiently multiply ring elements by monomials for this procedure to be fast. If we represent ring elements by coefficients vectors $\in \mathbb{Z}_q^N$, we have to perform an array rotation followed by changing the sign of c elements, which takes $O(N)$ time. However, by using a smarter representation (such as the $2N-1$ sized vector $[-a_{N-2}, -a_{N-3}, \ldots, -a_1, -a_0, a_{N-1}, a_{N-2} \ldots, a_1, a_0]$ and a start pointer) we can bring down this cost to $O(1)$ machine operations. RLWE' and RGSW ciphertexts will both require $O(\mathsf{dig})$ machine operations.

Plaintext multiplication.

Interface: $c \odot \mathsf{RLWE}'(m) \to \mathsf{RLWE}(c \cdot m)$

Input: $c \in \mathcal{R}_q$ in the coefficient representation and $\mathsf{RLWE}'(m)$ in the NTT or coefficient representation.

Output: $\mathsf{RLWE}(c \cdot m)$ in the NTT or coefficient representation

Algorithm: We first decompose c as $(c_0, \ldots, c_{k-1})$ satisfying $c = \sum_{i \in [k]} c_i \cdot B^i$. All of the c_i have infinite norm bounded by B. We then apply the NTT transform to each c_i (and the input ciphertext if in coefficient form) and compute the following in NTT domain

$$\begin{aligned}
\mathsf{RLWE}(c \cdot m) &\leftarrow \sum_{i \in [k]} c_i \cdot \mathsf{RLWE}'(m)[i] \\
&= \sum_{i \in [k]} c_i \cdot \mathsf{RLWE}(B^i m) \\
&= \sum_{i \in [k]} c_i \cdot (a_i,\ a_i \cdot s + B^i m + e_i) \\
&= \left(\sum_{i \in [k]} a_i c_i,\ s \cdot \sum_{i \in [k]} a_i c_i + m \cdot \sum_{i \in [k]} c_i B^i + \sum_{i \in [k]} c_i e_i \right) \\
&= \left(a,\ a \cdot s + c \cdot m + \sum_{i \in [k]} c_i e_i \right) \qquad \text{where } a = \sum_{i \in [k]} a_i c_i
\end{aligned}$$

Afterwards, an INTT is necessary if the output needs to be in coefficient representation. The resulting error has norm $O(\mathsf{dig} \cdot B \cdot |e| \cdot \sqrt{N})$ where e is the input error. This allows correct decryption for $c \cdot m$ with a more manageable error growth. Note that the analysis looks very similar if we use $\mathsf{RLWE}_{\mathsf{msb}}$ or $\mathsf{RLWE}_{\mathsf{lsb}}$ encodings instead.

Cost: If the input and output ciphertexts are in coefficient representation, the cost is essentially $3\mathsf{dig} + 2$ NTT/INTTs (where dig of NTTs for c and $2\mathsf{dig}$ of NTTs for RLWE' and 2 INTTs to transform the result back to its coefficient form). Otherwise, the cost is essentially dig of NTT/INTTs. Both of these ignore the coordinate-wise multiplications, since it takes little time compared to NTT/INTT.

BV-like ciphertext multiplication. We can multiply ciphertexts by following the recipe of [19]. This strategy requires an additional input called an evaluation key which is defined as $\mathsf{evk} := \mathsf{RLWE}'(s^2)$ where s is the secret key. The evaluation key is always represented in the NTT domain. Below, we present two variants of this protocol corresponding to the two encodings presented.

Interface: $\mathsf{RLWE}_{\mathsf{lsb}}(m_1) \times \mathsf{RLWE}_{\mathsf{lsb}}(m_2) \to \mathsf{RLWE}_{\mathsf{lsb}}(m_1 m_2)$

Input: $\mathsf{RLWE}_{\mathsf{lsb}}(m_1) = (a_1, b_1)$, $\mathsf{RLWE}_{\mathsf{lsb}}(m_2) = (a_2, b_2)$ and $\mathsf{evk} = \mathsf{RLWE}'(s^2)$ (which is implicitly taken in the interface), each of them using NTT representation

Output: $\mathsf{RLWE}_{\mathsf{lsb}}(m_1 m_2)$ in NTT representation

Algorithm: We first compute $(x = a_1 a_2, y = a_1 b_2 + a_2 b_1, z = b_1 b_2) \in \mathcal{R}_q^3$. Observe that

$$\begin{aligned} xs^2 - ys + z &= a_1 a_2 s^2 - a_1 b_2 s - a_2 b_1 s + b_1 b_2 \\ &= (a_1 s - b_1)(a_2 s - b_2) \\ &= (m_1 + e_1 t)(m_2 + e_2 t) \\ &= m_1 m_2 + t \cdot (e_1 m_2 + e_2 m_1 + e_1 e_2 t) \\ &= m_1 m_2 + t e' \end{aligned}$$

where

$$\begin{aligned} |e'| &= |e_1 m_2 + e_2 m_1 + e_1 e_2 t| \\ &\leq |e_1 m_2| + |e_2 m_1| + t \cdot |e_1 e_2| \\ &\approx O(\sqrt{N} \cdot |e_1||m_2|) + O(\sqrt{N} \cdot |e_2||m_1|) + t \cdot O(\sqrt{N} \cdot |e_1||e_2|) \end{aligned}$$

Note that $|m_1|$, $|m_2|$ are at most t since they both belong in $\mathcal{R}_t$

$$= \sqrt{N} \cdot t \cdot O(|e_1| + |e_2| + |e_1 e_2|)$$

Here $\sqrt{N}$ is the ring expansion factor (see [57]).[12] We now relinearize the ciphertext as per [57]. Observe that (y, z) is a $\mathsf{RLWE}_{\mathsf{lsb}}$ ciphertext encrypting $m_1 m_2 - xs^2$ (with error $t \cdot e'$). We compute $x \odot \mathsf{evk}$ to obtain $\mathsf{RLWE}(xs^2)$. As observed before, additive homomorphism of the underlying scheme implies that simply adding these two ciphertexts will yield $\mathsf{RLWE}_{\mathsf{lsb}}(m_1 m_2) = (y, z) + x \odot \mathsf{evk}$.

Cost: The cost is dominated by $\mathsf{dig} + 1$ of NTT/INTTs (transforming x back to its coefficient representation, decompose it, and transform it into its NTT representation).

For $\mathsf{RLWE}_{\mathsf{msb}}$ ciphertexts, it is slightly more complicated [16, 36]. Since it is less relevant to our work, we omit the details.

GSW-like multiplication. The RGSW ciphertexts can be used for a different style of ciphertext multiplication [44]:

Interface: $\mathsf{RLWE}_{\mathsf{msb}}(m_1) \times \mathsf{RGSW}(m_2) \to \mathsf{RLWE}_{\mathsf{msb}}(m_1 m_2)$

Input: $\mathsf{RLWE}_{\mathsf{msb}}(m_1) = (a, b)$ in coefficient representation and $\mathsf{RGSW}(m_2) = (\mathsf{RLWE}'(-s \cdot m_2), \mathsf{RLWE}'(m_2))$ in NTT representation or coefficient representation. Here $m_1 \in \mathcal{R}_t$ but $m_2 \in \mathcal{R}_2$ (but lifted to $\mathcal{R}_q$).

[12] Note that in the worst case, the expansion factor is N since N different elements are added up, but in practice, it is $\sqrt{N}$ as used by most works, so we use $\sqrt{N}$ here as well for simplicity.

Output: $\mathsf{RLWE}_{\mathsf{msb}}(m_1 m_2)$ in coefficient representation.

Algorithm: We calculate

$$\begin{aligned}
&a \odot \mathsf{RLWE}'(-s \cdot m_2) + b \odot \mathsf{RLWE}'(m_2) \\
=&\mathsf{RLWE}(-asm_2) + \mathsf{RLWE}(bm_2) \\
=&\mathsf{RLWE}((b - as) \cdot m_2) \\
=&\mathsf{RLWE}(\Delta \cdot m_1 m_2 + e_1 m_2) \\
=&(a',\ a's + \Delta \cdot m_1 m_2 + e_1 m_2 + e')
\end{aligned}$$

Since $|m_2| \leq 1$, the term $e_1 m_2 + e'$ is small and the above expression is a valid $\mathsf{RLWE}_{\mathsf{msb}}$ ciphertext.

Cost: The total cost is dominated by the two $\odot$ operations for a total of $2\mathsf{dig}$ NTT/INTTs if the both the input RGSW ciphertext and output are in NTT form. Otherwise, the total cost is $6\mathsf{dig} + 4$ NTT/INTTs.

Key switching. We can change the secret key used for a ciphertext as described in [57, Appx B]. This requires an additional input called a key switching key which encrypts the old secret key s_1 using the new secret key s_2 as follows: $\mathsf{ksk} := \mathsf{RLWE}'(s_1)$. The key switching key is always represented in the NTT domain.

Interface: $\mathsf{KeySW}(\mathsf{ksk}, \mathsf{Enc}^{s_1}(m)) \to \mathsf{Enc}^{s_2}(m)$

Input: $\mathsf{Enc}^{s_1}(m) = (a_1, b_1)$ and $\mathsf{ksk} = \mathsf{RLWE}'^{s_2}(s_1)$, each of them using NTT representation. Here $\mathsf{Enc} \in \{\mathsf{RLWE}, \mathsf{RLWE}_{\mathsf{msb}}, \mathsf{RLWE}_{\mathsf{lsb}}\}$.

Output: $\mathsf{Enc}_{s_2}(m)$ in NTT representation.

Algorithm: We calculate $-a_1 \odot \mathsf{ksk} = \mathsf{RLWE}^{s_2}(-a_1 s_1) = (a_2, b_2)$. Assuming $\mathsf{Enc} = \mathsf{RLWE}$, we have

$$\begin{aligned}
b_1 &= a_1 s_1 + m + e_1 \\
b_2 &= a_2 s_2 - a_1 s_1 + e_2 \\
\implies b_1 + b_2 &= a_2 s_2 + m + e_1 + e_2 \\
\implies (a_2, b_1 + b_2) &= \mathsf{RLWE}_{s_2}(m)
\end{aligned}$$

The analysis for the two other encryption schemes are the same.

Cost: Similar to the BV-like multiplication above, the cost is dominated by $\mathsf{dig}{+}1$ of NTT/INTTs.

Modulus switching.

Interface: $\mathsf{ModSW}(\mathsf{Enc}^{q}(m)) \to \mathsf{Enc}^{q'}(m)$

Input: $\mathsf{Enc}(m) = (a, b)$ in coefficient form (where $\mathsf{Enc} \in \{\mathsf{RLWE}_{\mathsf{msb}}, \mathsf{RLWE}_{\mathsf{lsb}}\}$) and a new modulus $q' < q$. Observe that $m \in \mathcal{R}_t$ and $(a, b) \in \mathcal{R}_q^2$

Output: $\mathsf{Enc}(m) = (a', b')$ where $(a', b') \in \mathcal{R}_{q'}^2$, where the output noise is $O(q'e/q + ||s||_2)$ for input noise e and secret s.

Algorithm: The new ciphertext is obtained as

$$(a', b') = \frac{q'}{q}(a - t\lfloor a/t \rceil_{q/q'}, b - t\lfloor b/t \rceil_{q/q'}) \mod q'$$

where $\lfloor \cdot \rceil_{q/q'}$ standards for computing the flooring and then mod q/q' (and for simplicity, assume q/q' being an integer for the $\mathsf{RLWE}_{\mathsf{lsb}}$ case) if $\mathsf{Enc} = \mathsf{RLWE}_{\mathsf{lsb}}$, and see [18] for a detailed correctness analysis.

The new ciphertext is obtained as

$$(a', b') = \lfloor \frac{q'}{q}(a, b) \rceil$$

if $\mathsf{Enc} = \mathsf{RLWE}_{\mathsf{msb}}$. See [18, 16] for a detailed correctness analysis.

Cost: The cost is dominated by $2N$ real number multiplications and rounding.

Extract. Sometimes we want to extract specific coefficients of a message $m \in \mathcal{R}_t$ encoded using a ring-based scheme into LWE ciphertexts. This can be done really efficiently.

Interface: $\mathsf{Extract}(\mathsf{RLWE}^s_{\mathsf{msb}}(m)) \to \mathsf{LWE}^{\vec{s}}_{\mathsf{msb}}(m_i)$

Input: $\mathsf{RLWE}^s_{\mathsf{msb}}(m) = (a, b)$ in coefficient representation, and an index $i \in [N]$.

Output: $\mathsf{LWE}^{\vec{s}}_{\mathsf{msb}}(m_i)$ where m_i is the coefficient of X^i in the polynomial m, and $\vec{s}$ is the vector of all the coefficients of s.

Algorithm: We show the algorithm when $i = 0$. It can be modified to work for arbitrary indices fairly easily. We know that $b = as + m\Delta + e$. We view each element as a polynomial and focus on the constant coefficient. Clearly, $b_0 = (as)_0 + m_0\Delta + e_0$. The constant term of the product as is $\sum$

$$\begin{aligned} b_0 &= (as)_0 + m_0\Delta + e_0 \mod q \\ &= a_0 s_0 + \sum_{j=1}^{N-1} (-s_j a_{N-j}) + m_0\Delta + e_0 \mod q \qquad (\text{since } X^N = -1) \end{aligned}$$

We define the vectors $\vec{s} := [s_0, s_1, \ldots, s_{N-1}]$ and $\vec{a}' := [a_0, -a_{N-1}, -a_{N-2}, \ldots, -a_1]$

$$\begin{aligned} &= \langle \vec{a}', \vec{s} \rangle + \Delta \cdot m_0 + e_0 \mod q \\ &\implies (\vec{a}', b_0) = \mathsf{LWE}_{\mathsf{msb}}(m_0) \end{aligned}$$

This works for RLWE and $\mathsf{RLWE}_{\mathsf{lsb}}$ ciphertexts the same way.

Cost: This operation simply requires copying the coefficients of a and negating some of them. This takes about N machine operations and is almost free compared to the rest of the procedures described here.

			LWE Parameters			RLWE Parameters		
Parameters	**Functionality**	**Security**	n	q	h	N	Q	Q'
Param 1	Gate Boots	STD_128	1024	512	38	1024	2^{28}	2^{12}
Param 2	Gate Boots	MEDIUM	1024	512	27	1024	2^{28}	2^{12}
Param 3	Arb Func	STD_128	2048	2^{12}	65	2048	2^{54}	2^{35}

Table 6: Parameters for general sparse ternary secret keys. Q' is an intermediate ciphertext modulus used by OpenFHE. The rest of the parameters are the same as OpenFHE, so we omit the repetition.

			LWE Parameters			RLWE Parameters		
Parameters	**Functionality**	**Security**	n	q	h	N	Q	Q'
Param 1	Gate Boots	STD_128	1008	512	56	1024	2^{28}	2^{12}
Param 2	Gate Boots	MEDIUM	880	512	40	1024	2^{28}	2^{12}
Param 3	Arb Func	STD_128	2046	2^{12}	93	2048	2^{54}	2^{35}

Table 7: Parameters for structured ternary secret keys.

B Additional Evaluation for Ternary Keys

For completeness, we also show the performance of our constructions with ternary keys.

Methodology and parameters. We follow the exact same methodology as Section 7.1 and give the following parameter sets for general sparse ternary secrets (Table 6) and structured sparse ternary secrets (Table 7).

Performance. As shown in Table 8, we similarly achieve ∼4× to ∼7× improvement over the TFHE implementation in OpenFHE [8] when using sparse keys. The difference comes from the fact that for ternary keys, the number of additions doubles, while the number of multiplications is slightly smaller, according to [1].

	Boolean Gates (ms)				**Arb. Func.** (ms)
	STD_128		MEDIUM		STD_128
Implementation	No Preproc.	With Preproc.	No Preproc.	With Preproc.	No Preproc.
TFHE (OpenFHE) [29, 8]	36.5	–	30.1	–	391.5
Ours (sparse ternary)	8.5	–	5.8	–	68.0*
Ours (structured sparse)	8.5	7.1	6.3	5.3	58.8*

Table 8: Runtime comparison. Ours (sparse ternary) and (structured sparse) use Algorithm 2 with general ternary secrets and structured ternary secrets, respectively (see Section 4.2). Preproc. refers to the preprocessing technique in Section 4.3, which enhances runtime at the cost of larger bootstrapping keys. Note that "–" means preprocessing does not improve the runtime with the current proof of concept implementation. "*" stands for estimated runtime using microbenchmarks with our current implementation. See Section 7.1 for details.